August 2017

Improving Pattern Recognition and Neural Network Algorithms With Applications to Solar Panel Energy Optimization

Ernesto Zamora Ramos
University of Nevada, Las Vegas, sidezr@gmail.com

IMPROVING PATTERN RECOGNITION AND NEURAL NETWORK ALGORITHMS WITH APPLICATIONS TO SOLAR PANEL ENERGY OPTIMIZATION

by

Ernesto Zamora Ramos

Bachelor of Science – Computer Science
University of Nevada, Las Vegas
2013

A dissertation submitted in partial fulfillment of
the requirements for the

Doctor of Philosophy in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2017

UNLV | GRADUATE COLLEGE

Dissertation Approval

The Graduate College
The University of Nevada, Las Vegas

May 17, 2017

This dissertation prepared by

Ernesto Zamora Ramos

entitled

Improving Pattern Recognition and Neural Network Algorithms With Applications to Solar Panel Optimization.

is approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science
Department of Computer Science

Evangelos Yfantis, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Hal Berghel, Ph.D.
Examination Committee Member

Laxmi Gewali, Ph.D.
Examination Committee Member

Dr. Andreas Stefik, Ph.D.
Examination Committee Member

Dr. Sarah Harris, Ph.D.
Graduate College Faculty Representative

Abstract

Artificial Intelligence is a big part of automation and with today's technological advances, artificial intelligence has taken great strides towards positioning itself as the technology of the future to control, enhance and perfect automation. Computer vision includes pattern recognition and classification and machine learning. Computer vision is at the core of decision making and it is a vast and fruitful branch of artificial intelligence. In this work, we expose novel algorithms and techniques built upon existing technologies to improve pattern recognition and neural network training, initially motivated by a multidisciplinary effort to build a robot that helps maintain and optimize solar panel energy production.

Our contributions detail an improved non-linear pre-processing technique to enhance poorly illuminated images based on modifications to the standard histogram equalization for an image. While the original motivation was to improve nocturnal navigation, the results have applications in surveillance, search and rescue, medical imaging enhancing, and many others.

We created a vision system for precise camera distance positioning motivated to correctly locate the robot for capture of solar panel images for classification. The classification algorithm marks solar panels as clean or dirty for later processing. Our algorithm extends past image classification and, based on historical and experimental data, it identifies the optimal moment in which to perform maintenance on marked solar panels as to minimize the energy and profit loss.

In order to improve upon the classification algorithm, we delved into *feedforward neural networks* because of their recent advancements, proven universal approximation and classification capabilities, and excellent recognition rates. We explore state-of-the-art neural network training techniques offering pointers and insights, culminating on the implementation of a complete library with support for modern deep learning architectures, multilayer percepterons and convolutional neural networks.

Our research with neural networks has encountered a great deal of difficulties regarding hyperparameter estimation for good training convergence rate and accuracy. Most hyperparameters, including architecture, learning rate, regularization, trainable parameters (or weights) initialization, and so on, are chosen via a trial and error process with some educated guesses. However, we developed

the first quantitative method to compare weight initialization strategies, a critical hyperparameter choice during training, to estimate among a group of candidate strategies which would make the network converge to the highest classification accuracy faster with high probability. Our method provides a quick, objective measure to compare initialization strategies to select the best possible among them beforehand without having to complete multiple training sessions for each candidate strategy to compare final results.

Acknowledgements

I would like to offer my sincere gratitude to my mentor and advisor Dr. Evangelos A. Yfantis for his continuous support and encouragement to keep moving forward, his patience and experience. I deeply appreciate his disposition to share his knowledge, teaching me something new at every step of the way. His guidance helped me throughout the creation, research, experiments and writing of this document. I cannot thank him enough for helping me be where I am today.

I would also like to extend my gratitude to the members of my committee: Dr. Hal Berghel, Dr. Laxmi Gewali, Dr. Andreas Stefik, and Dr. Sarah Harris for their support and encouragement that incented me to widen my research views and try new technologies.

Special thanks also go to Mr. Rick Hurt for his assistance and for allowing me access to the solar lab and equipment to collect data and to conduct experiments.

Last, but not least, I would like to thank my family, especially my mother, Fidelia, for their unconditional love and support through the duration of my studies, during both difficult times and good times.

Ernesto Zamora Ramos

University of Nevada, Las Vegas
August 2017

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Theory and applications of computer vision are quite prevalent today, mainly because the increase in computing power and developments in parallel computing have made possible to achieve the high volume of computations required by old and new techniques to simulate, and, in some cases, surpass, human-like behavior in recognizing and classifying signals such as images or sounds. Once computers are programmed and learn to perform successful recognition optimally, massive amounts of information can be reasonably mined, classified and logically arranged to free humanity of such tedious tasks. This allows people to focus more on creative and scientific endeavors.

In this research, we develop novel algorithms and techniques built upon existing technologies to improve pattern recognition and neural network training, specifically applied to signal and image processing. The initial motivation was to improve pre-processing, segmentation and classic pattern classification methods to apply them to a research project funded by the National Science Foundation. The research project focused on a multidisciplinary effort to create a robot-server architecture to clean and maintain solar installations. The results extend to applications beyond the focus of the project and we developed enhanced techniques for software night vision for both grayscale and color images. We developed techniques that allow spatial positioning for image capture, applied classic pattern classification methods on captured images to detect the cleanliness of solar panels and predict the optimal time to clean them and minimize the energy loss due to production drop from dirty photovoltaic cells. We delved into feedforward neural networks because of their recent advancements, proven universal approximation and classification capabilities, and excellent recognition rates, in particular, Convolutional Neural Networks as the state-of-the-art signal classifiers. We studied the mathematics behind neural networks' capability to learn patterns from examples and the internal working of network structures.

Current research on neural networks focuses on incremental improvement in network convergence

accuracy and time to learn or convergence rate (in number of epochs and number of samples per epoch needed to converge) without sacrificing optimization. Our research with neural networks encountered a great deal of difficulties regarding hyperparameter estimation for good training convergence rate and accuracy. Most hyperparameters, including architecture, learning rate, regularization, trainable parameters (or weights) initialization, and so on, are chosen via a trial and error process with some educated guesses. One of the hyperparameters we studied was the weight parameter initialization as a critical step during training and one of the most difficult choices to boost accuracy, or even to start learning. Empirical evidence exists that some researchers have been able to train some large networks better using one initialization technique over another, and while there is a consensus on different techniques to initialize the weights, there is no formal evidence of one method being better than another. Our contribution as part of this research is the development of the first quantitative method to compare weight initialization strategies based on experiments designed to estimate, among a group of candidate strategies, which would make the network converge to the highest classification accuracy faster with high probability. Our method provides a quick, objective measure to compare initialization strategies to select the best possible among them beforehand without having to complete multiple training sessions for each candidate strategy to compare final results.

Instead of experimenting on existing neural network and deep learning libraries, we created our own neural network library in C++ using modern techniques of computation such as multi-core computing, mass matrix and vector machine optimization libraries and coprocessor offloading to Many Integrated Cores (MICs) and Graphic Processing Units (GPUs), with the purpose of understanding the internal functionality and details and to fulfill our need to be able to monitor and tweak the internal functionality of a neural network to fit our experiments at will. Our library was able to stand on par with state-of-the-art libraries in comparable situations such as Google's TensorFlow and Intel's Deep Learning libraries. We deployed our library to the National Supercomputer Institute Cherry-Creek cluster to speed up computation on our experiments allowing us to complete computations in hours instead of days.

In the next chapter we survey important previous work on image pre-processing, dark image enhancement and histogram equalization variants. We continue with a survey on photogammetry, solar panel technology and maintenance. We conclude the chapter with a preliminary survey of neural networks and an in-depth look at their weight initialization strategies.

After the literature review, we dive into our findings regarding enhancement of poorly illuminated images along with comparisons with the standard method of histogram equalization, ending the discussion with striking results when our proposed algorithm is applied to color images.

We follow the pre-processing treatment with our unveiling of the vision system for precise camera distance positioning that was part of the robot Helios' kit along with the night vision system provided by our enhanced histogram equalization technique. We discuss the mathematics behind the model followed by an overview of the solar panel cleanliness classification system. We end the chapter by explaining the details of our maintenance schedule algorithm, tied to the classification system in order to minimize costs and loss of energy.

The details that went into the construction of our neural network library are explained in the next chapter. We survey modern techniques and state-of-the-art enhancements to train multilayer perceptrons and deep architectures of convolutional neural networks along with our own implementation details, improvements and optimizations with the goal of creating a worthy library that stands on par with popular neural network libraries. The resulting code is listed in the appendices.

The motivation for the creation of our own neural network library was to be able to train a network to improve upon the classification for the solar panel maintenance project. However, the time-consuming trial and error approach to hyperparameter selection to train the networks led us to research and formulate the quantitative comparison method explained in the last chapter for the selection of parameter initialization strategy in order to improve neural network accuracy while saving crucial training time and resources.

Chapter 2

Literature Review

2.1 Non-linear Equalization for Poorly Illuminated Images

Histogram equalization is an image processing technique often used to enhance contrast. It works by expanding the histogram of an image to cover the whole dynamic range.

Even though standard Histogram Equalization (HE) offers a direct alternative for the enhancement of dark images, in many occasions the resulting images are too bright, or washed out, thus, several researchers have proposes improvements upon this technique in general, and for the purpose of night vision.

Several variations to the HE algorithm exist. Many of these try to increase the contrast and dynamic range of an image while preserving the brightness or expected intensity value with the purpose of applying it to images with low contrast, but not specifically for poorly illuminated images.

In their work "Contrast Enhancement Using Brightness Preserving Bi-Histogram Equalization" [34], the authors explain a popular variation of standard HE, called Bi-Histogram Equalization (Bi-HE). This is one of several variations to HE that aims to maintain the image average intensity while expanding the dynamic range. Bi-HE seeks to divide the image histogram into two histograms separated by the mean value. Standard HE is then performed on each sub-histogram. Methods like this are introduced in an attempt to make HE viable in consumer electronics by lessening the artifact of brightness change in an image after equalizing it.

Chen and Ramli [10] propose a variation of Bi-HE where they apply Bi-HE recursively. Each new section of histogram delimited by its bounds or means is subjected to Bi-HE once again, and so on.

Another variation divides the image into regular regions of equal area and applies HE to each

region so that the variance of the intensity is smaller in the locality and does not affect the result as drastically as the global variance [86]. The result, however, is a patchy image with unnaturally bright areas.

While histogram equalization methods that preserve brightness could be useful to enhance images with poor contrast, they may not be as appropriate to utilize for poorly illuminated images because the intention is to increase their contrast as well as to brighten them up. These methods also tend to suffer from the same downside as standard HE when applied to dark images, still providing washed out results.

However, histogram equalization has only been used in a limited amount of work as a night vision alternative. Many authors have also realized that standard HE and variations used in other types of images need to be modified in order to provide better results for dark images.

In his thesis work [69], Teo realizes that HE, or one of its variants, can be applied to night vision images to enhance their contrast and improve further on the quality of visuals. The author, however, applies the technique to images already captured with night vision or thermal imaging devices, so, the effect of HE on these images is not as pronounced as when applied directly to the original dark images.

In their paper "New image enhancement algorithm for night vision," [83] the authors propose a combination of HE and contrast enhancement to improve upon standard HE when used for dark images after realizing the unnatural increase in brightness in resulting images when applied for night vision.

Sapkota [63] shows in his work the application of the concept, already explaining the possibility of building a capture device that utilizes HE to provide night vision for low light environments. He goes on to explain Incremental Histogram Equalization to look for the optimal upper bound of dynamic range expansion of the histogram where the resulting image would more closely resemble the well illuminated version. The author, however, still uses the standard HE algorithm to obtain the night vision result, just varying the upper bound intensity to find the peak of signal to noise ratio.

In most of these papers, the variations to the standard HE provide some improvement on the resulting images, however, almost none addressed the issue that HE, in fact, does not expand the histogram to the whole dynamic range for poorly illuminated images as it does for better lit images.

2.2 Vision System and Optimization for Solar-Panel-Cleaning Robot Architecture

Solar power plants are currently growing in number across the globe. They are a source of renewable, clean energy that can be used to significantly reduce the ecological impact and increase the efficiency of production of electrical energy.

Solar power plants incorporate large arrays of solar panels. However, today, many individuals have access to solar panels that can be used to produce enough electricity to power a house. Research in the area is abundant right now in pursuit of more efficient ways to collect the solar energy.

Solar panels are collections of interconnected solar cells (also called photovoltaic cells) that absorb the energy of incident light, converting it into an electric current through a phenomenon called "photovoltaic effect."

The photovoltaic (PV) effect is directly related to the photoelectric effect. In summary, the electrons on certain materials can be excited by incident light. Semiconductor materials, such as silicon, are usually used. When an electron in the valence band of a crystal's atom absorbs enough energy from incident photons, it jumps to the conductive band and becomes free, ionizing the source atom with a positive charge. Under the presence of an electric field, the separated electrons and ions are attracted to the opposite charged plates, creating an electromotive force. If a circuit is connected to these plates, an electric current flows. As light continues to excite the material, the ionization is maintained and the electricity continues to flow [5].

Clearly, if more light reaches and gets absorbed by a PV cell, then, more atoms get ionized in the crystal and more electrons become free. As a result, the potential of the electromotive force created by the separation of more negative and positive charges increases as well as the electricity flowing through the circuit.

The basic structure of a PV cell is designed to allow the maximum light possible to reach the excitable material, maximizing the absorption of photons and minimizing reflection. Solar cells rely on a layer of antireflection coating on the front of the cell to reduce reflection of the incident light. On simple cells, light rays enter through the front surface and, if not absorbed, leave through the rear. More sophisticated designs extend the path of light inside the cell to improve absorption through a process called "light trapping" [51].

Sunlight is comprised of ultraviolet, visible, and infrared light. While red light has the longest wavelength in the visible spectrum, infrared light is even longer. Therefore, infrared and red light, both having the larger wavelengths compared to the other components of the incident light, penetrate the glass more readily and produce the most amount of electricity. Meanwhile, blue and violet light

suffer the most absorption by the coating of the cell, not the cell itself, and offer little contribution to the production of electric energy in comparison.

As we said earlier, the amount of energy generated by PV cells is directly proportional to the amount of light absorbed. And the more light directly illuminating the cell, more photons reach the material, and more light can be absorbed.

Now, the amount of light striking the solar cells on a solar panel array is dependent on many factors, including the month of the year, day of the month, time of day, weather conditions, and other locationdependent circumstances. Most weather conditions that can limit the amount of light, and thus, the amount of electricity generated, cannot be avoided. Other causes, however, such as light obstruction due to other objects, broken cells and overall cleanliness or dirtiness of the panels, can be dealt with in order to maximize the amount of light reaching the solar cells.

Many solar power plants are established in areas with arid climates due to the low humidity and clear skies year round. Dust and sand storms are common in these climates and the dust gradually settles on the glass surface of solar panels, slowly decreasing the amount of light that reaches the solar cells. The loss of light energy depends on the amount, size, and chemical composition of the dust [62] [49]. In terms of time, trees are sparse in arid climates, and during migration in the fall and spring, birds use solar farms as rest areas; therefore, the solar panels become dirty with bird excrement. In general, this is a problem throughout the entire year. Bird droppings are worse than dust, because no light passes through them.

Deposits on a dirty panel reflect, scatter and obstruct the incident light, reducing the amount of photons that can penetrate through and reach the PV cells, consequently decreasing the amount of electricity produced [62]. To position the capture system in order to properly obtain images of solar panels and to determine whether they are clean or dirty, we applied some photogrammetric techniques.

Photogrammetry (photo-light, gram-drawing, metry-measuring) has a Greek derivation, and is the practice of determining the geometric properties of objects from photographic images. It is dated back to nineteenth century when film photography started. The process is as simple as getting the distance between two points on a plane parallel to the photographic image plane. Work in stereo photogrammetric image enhancement, image processing, and stereo vision started in the latter part of the last century.

Our work in this area pertains to a robot vision system using some photogrammetric techniques [87][73], initial classification algorithm utilizing Mahalanobis distance to detect dirty solar cells [87][52][85] and a photovoltaic cell output optimization algorithm that estimates the ideal time to clean solar panels to minimize impact in energy production [47].

2.3 Neural Network Details

Feed-Forward Artificial Neural Networks have been around since the 1960s. However, recent advancements in computing power, parallelization and increasing computer memory have allowed for the scarce old techniques to be implemented, improved upon and be quite successful in pattern recognition and classification, permitting deeper networks and new architectures to appear and thrive.

Classic fully connected, feed-forward artificial neural networks present an input layer or retina with, usually, no processing steps other than to prepare and organize the input data for the the rest of the network. The input layer is followed by, at least, one hidden layer of non-linear units, also called neurodes or neurons due to their mathematical model and simulation of real-life neuron cell behavior. The hidden layers are followed by the output layer which produces the final output of the network in whatever desired format, usually by applying similar processing and some non-linearity to its input, much like the hidden layers operate. The non-linear activation function in classic networks is usually set to the log-sigmoid or hyperbolic tangent as continuous, differentiable alternatives to the original McCulloch-Pitts nodes [50] in order to facilitate training [59][56]. Modern architectures, however, are not limited to fully connected layers or to sigmoid activation units.

While fully connected layers are often integral parts of the newer architectures, modern networks possess layers with specific connections. Convolutional Neural Networks such as LeNet [42], GoogLeNet [31], Inception Network [68] and others contain sparsly connected layers known as convolutional layers that act as feature filters. These networks are considered deep architectures because instead of having a couple of fully connected hidden layers, they figure from 5 to 14 or even more combination of layers, from convolutional to pooling to fully connected layers. Recurrent Neural Networks, also considered deep networks, present another type of architecture with feedback loops that form directed cycles inside the network, allowing it to show temporal memory behavior, such as the Long Short Term Memory based networks [28], a variant of the Recurrent Neural Networks that aims to reduce or eliminate the vanishing gradient problem [35].

It has been shown that neural networks with continuous non-linear and non-polynomial activation units in, at least, one hidden layer possess the universal approximation property [29][45]. While classic units feature sigmoid or hyperbolic tangent functions as non-linear activations, other activation functions have been researched over the years and applied successfully to deep architectures. Rectified Linear Units (ReLU) use the identity function for positive input values and zero for negative input values as the non-linear activation [21]. Variants to avoid dead ReLU units (units that stop learning) such as leaky ReLU [46] and the softplus function [17] have shown improved results. Recent studies have brought Exponential Linear Units (ELU) as another alternative in attempts to

fight some of the drabacks of ReLU units [12]. Other seminal activation functions that appeared as alternatives to the sigmoid were the Radial Basis Function [57] and an alternative non-sigmoidal activation proposed by Chandra, Ghose and Sood that behaves similarly to the Radial Basis Function, but involves no exponential calculation, potentially helping on optimization of the training process [9].

The most commonly used method for training neural networks is the back propagation algorithm. Back propagation is a variant of gradient descent optimization that uses first order derivatives to compute the error between a target function and the approximation. The error is utilized to change the parameters of the approximation using the steepest slope to steer it towards the known function [61][60][40]. The majority of today's state-of-the-art neural networks use this algorithm or some variant or extension with modifications because of its ease of computational speed and space optimization and proven results [31][68].

Other methods have been proposed as learning algorithms, especially second order methods to increase convergence rate [2][7][58]. In terms of convergence rate, it has been shown that second order methods converge faster than gradient descent based training and are among the algorithms with higher convergence rates [43]. Another appeal to second order methods, other than their proven superior convergence rates is the fact that many second order methods do not have the learning rate hyperparameter and adjust themselves depending on the shape of the error surface. A popular method for second order training is Newton's method and variations such as quasi-Newton methods or conjugate methods [24][48][3].

Despite boasting superior convergence rates, second order method are rarely used for practical applications in neural network training. Their drawback is that they are not practical for large neural networks trained in batch mode [8]. During back propagation, the explicit computation of the Hessian matrix and its inverse are quite expensive, reaching over 3 TB of memory needed to hold these matrices for a 1 million weights network. All of this already piles on the actual computational load of the matrices' elements. So, while the number of epochs needed by second order methods is less than stochastic gradient descent, the computational complexity, time and memory required can become prohibitive. The work done in "Large Scale Distributed Deep Networks" [14] concludes that second order methods can outperform stochastic gradient descent with AdaGrad adaptive learning rate procedure given enough computing power, but not for much. However, for modest systems with less than 2000 cores, stochastic gradient decent is the only practical solution.

2.3.1 Neural Network Weight Initialization

Despite all these new variants of the classic approach to fully connected, feed-forward neural networks, the challenge of selecting a good distribution of initialization parameters that would yield the best learning results persists. Various techniques are used to initialize different architectures and units depending on their non-linearity. Based on the activation function, the initialization strategy varies and there are different strategies for the same activation and architecture.

Most texts for beginners today indicate that a "good enough" weight initialization for sigmoid activation neurons (such as log-sigmoid or hyperbolic tangent functions) is a uniformly distributed random value in the interval $(-1, 1)$ [70][59][56]. Further details explain that better results are obtained by initializing each weight for the connection from neuron j to neuron i as:

$$w_{ij} = \text{uniform}(-1, 1) \cdot \alpha \tag{2.1}$$

where the coefficient $\alpha \in (0, 1)$ is called the scaling coefficient. It is usually recommended to pick small coefficient values. The closer this value is to zero seems to translate into better convergence rates. Typical coefficient values range between 0.01 and 0.5.

The SCAWI method [16][13] proposes a per-layer initialization for networks with a single hidden layer where:

$$\begin{aligned} \alpha^{(1)} &= \frac{1.3}{\sqrt{2}} \\ \alpha^{(2)} &= \frac{1.3}{\sqrt{1 + 0.3 \cdot fan\_in_i}} \end{aligned} \tag{2.2}$$

where $\alpha^{(l)}$ is the scaling coefficient for layer l and fan_in_i is the number of inputs to neuron i.

Nguyen and Widrow [55] suggest that since each hidden node computes a slice of the final function that the network tries to approximate, a good approach is to initialize the weights of a node to a uniform random interval dependent on the magnitude of the vector of weights for the node over the input space. That is, given the number of hidden nodes H and the vector or weights for node i is W_i, then the input interval should be divided in portions of size equal to:

$$\alpha = H^{1/fan\_in_i} \tag{2.3}$$

where each weight should be initialized to a uniform random number in their portion. The bias should be initialized to $\text{uniform}(-1, 1) \cdot \alpha$.

Other sources [41] [20] have found that better convergence rates result if the scaling coefficient is set as:

$$\alpha = fan\_in_i^{-1/2} \tag{2.4}$$

or

$$\alpha = 4\sqrt{6\left(fan\_in_i + fan\_out_i\right)^{-1}} \tag{2.5}$$

where fan_out_i is the number of neurons that use the activation of neuron i as input. The idea is based on the nature of sigmoid-like activation functions $f(x)$, such as the hyperbolic tangent and the sigmoid $f(x) = \dfrac{1}{1+e^{-x}}$, where e is Euler's constant, that saturate (i.e. $f(x)$ approaches an asymptote) for $|x| > 1$, while approaching a linear function elsewhere. Neurons with larger number of inputs should have smaller weights to avoid saturation when, at least, half of the inputs are set [4].

In their work, Sodhi, Chandra, and Tanwar [67] propose a variation on the Nguyen and Widrow method to initialize sigmoid units in which they also section the weights for the i-th unit in layer l with a total of $N^{(l)}$ hidden neurons by computing the sections as such:

$$w_{ij} = \text{uniform}\left(-\alpha + M(i-1) - \frac{M}{2}, -\alpha + M(i-1) + \frac{M}{2}\right) \tag{2.6}$$

where $M = \dfrac{2\alpha}{N^{(l)} - 1}$, with $\alpha \in (0, 1)$ as the scaling coefficient. This results in weights for a single unit in the range $(-\alpha, \alpha)$, but each weight is initialized in its own non-overlapping portion of the range.

Another method insists on initializing the weights using a random Gaussian distribution as in:

$$w_{ij} = \text{gaussian\_random}(\mu, \sigma) \tag{2.7}$$

where gaussian_random $\sim \text{n}\left(\mu, \sigma\right)$ is a normally distributed random value, μ is the mean and σ is the standard deviation. The suggested values are $\mu = 0$ and $\sigma \in (0, 1]$ [56], with smaller values of standard deviation offering better results [54].

In his book [56], Nielsen offers an intuitive explanation on why smaller standard deviations have better convergence rates and reduced error. He suggested that if we have fan_in_i inputs to the neuron i with a sigmoid activation function, where each weight was initialized as $w_{ij} =$ gaussian_random$(0, 1)$, assuming that, at least, half the connected neurons fire with a value of 1, then, the value of the local receptive field $S_i = \sum_{j=0}^{n_i} w_{ji} x_j$ (ignoring the bias) has a normal distribution of the form $\text{n}\left(0, \sqrt{fan\_in_i}\right)$. This means that about 68% of the values of S_i (input to the sigmoid) will be in the range $\left(-\sqrt{fan\_in_i}, \sqrt{fan\_in_i}\right)$ with $fan\_in_i >> 1$, and even worse, about 27% will be greater than $\sqrt{fan\_in_i}$. In general, more than half the time, the input to the sigmoid will be much greater than 1 making $f'(S) \approx 0$ as depicted in Figure 2.1, saturating the neuron. The

back propagation learning algorithm depends greatly on the value of the derivatives to compute how weights should change. Since the derivative of a saturated neuron evaluates close to zero, the resulting weight change will also be close to zero. Applying these small changes to the neuron's weights will have little effect on the neuron activation, thus slowing down learning.

Nielsen [56] and Glorot [20] suggest that to ensure $S_i \sim \mathrm{n}(0,1)$, a better approach is to make the standard deviation in the Gaussian initialization:

$$\sigma = \frac{1}{\sqrt{fan\_in_i}} \tag{2.8}$$

This method is commonly known as Xavier initialization [20] .

The goal in all these studies is the same: to avoid the saturation of sigmoid neurons to improve learning speed and convergence rates.

Initializing parameters for non-sigmoid units also has several variants and as many challenges as sigmoid units with specific attention to details. ReLU and variants to ReLU have been found to benefit from Xavier initialization technique in equation 2.8 with the added measure of initializing the bias of the unit to 1 to avoid early death of the neuron [26]. Simonyan and Zisserman [66] explain how they had to train shallower versions of their deep network and use the obtained weights to initialize deeper versions in a progressive manner.

Other parameter initialization methods try to perform a coarse approximation to the final solution and then use the approximation result as the initialization values for the neural network instead of randomly set the initial weights [59]. One such method is initial weight selection with genetic

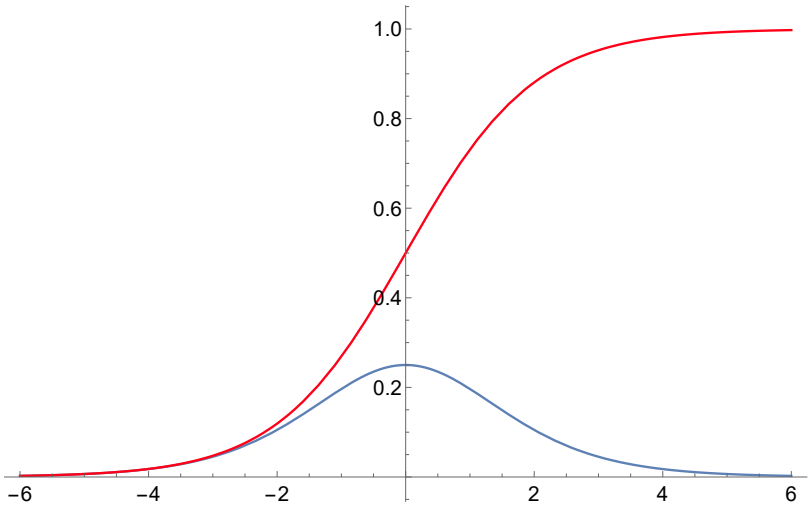


Figure 2.1: Sigmoid function and its derivative.

(Red) Sigmoid function; (red) derivative of the sigmoid. Notice that as the sigmoid approaches its asymptotes $y = 1$ and $y = 0$, the derivative approaches $y = 0$.

algorithms.

Genetic algorithms are a class of search and optimization algorithms based on an analogy to natural evolutionary mechanisms. The basics of a genetic algorithm is "survival of the fitest" where generations of results are compared using some cost function that decides which "individuals" are closest to the actual result and thus should continue existing. Along with some rarely introduced "mutations," the longest-living individuals live long enough to pass on their traits to the new generations in hopes that the combination of good enough approximations will result in better "offspring," eventually converging to a global maximum (or minimum, depending on the specific problem) guided by the process of "natural selection" enforced by the cost function [59][53][15].

There are many details for a genetic algorithm, but, theoretically, an initial approximation of the neural network weights using a genetic algorithm would steer the hyperplanes defined by each neuron on its input hyperspace towards the global optima. One main disadvantage of the algorithm is the nature of the running process that involves the level of approximation. The genetic algorithm needs a large amount of computation and storage space to evaluate a large population, thus running the algorithm for too long may hinder the whole training performance by attempting to approximate the solution too much. This overuse steps over the neural network training that is being initialized, and takes almost as much time as the neural network training itself. On the other hand, not enough approximation may result in an initialization that may be not good enough and even lead to divergence during training [59].

Chapter 3

Improved Non-linear Equalization for Night Vision

It is well known that histogram equalization (HE) is a method utilized in digital image processing to enhance the contrast of an image. It works by expanding the dynamic range of the image histogram.

In the histogram for an image with poor contrast, it can be seen that all pixels are clustered close together around a few intensities. After applying histogram equalization to the image, it is observed how the pixels are no longer clumped together, but their intensities have been spread, trying to expand over the whole range of the histogram. This increased distance between intensities translates in an increased contrast for the image as it can be seen in Figure 3.1.

Note that for this discussion, it is assumed, unless otherwise specified, that digital images have been converted to grayscale using the intensity value of each pixel, also known as luma or Y component in the YUV color space, computed from the RGB color space as standardized by International Telecommunications Union in BT.601-7.

HE accomplishes its goal by applying a non-linear transformation to the image in question. It computes the cumulative distribution function (CDF) of the histogram of the image, and then uses it as a look up table for the new pixel intensity values in the resulting image. Algorithm 1 shows the definition of the classic HE algorithm [22].

Now, notice that the histogram for images captured with commercial digital cameras under poor illumination conditions is similar to that of an image with low contrast, but the intensities are closer to zero (see Figure 3.2).

If HE is applied to poorly illuminated images, the result is a clearer picture. The little energy captured is amplified, lighting up the original image, and making the shapes visible to the human

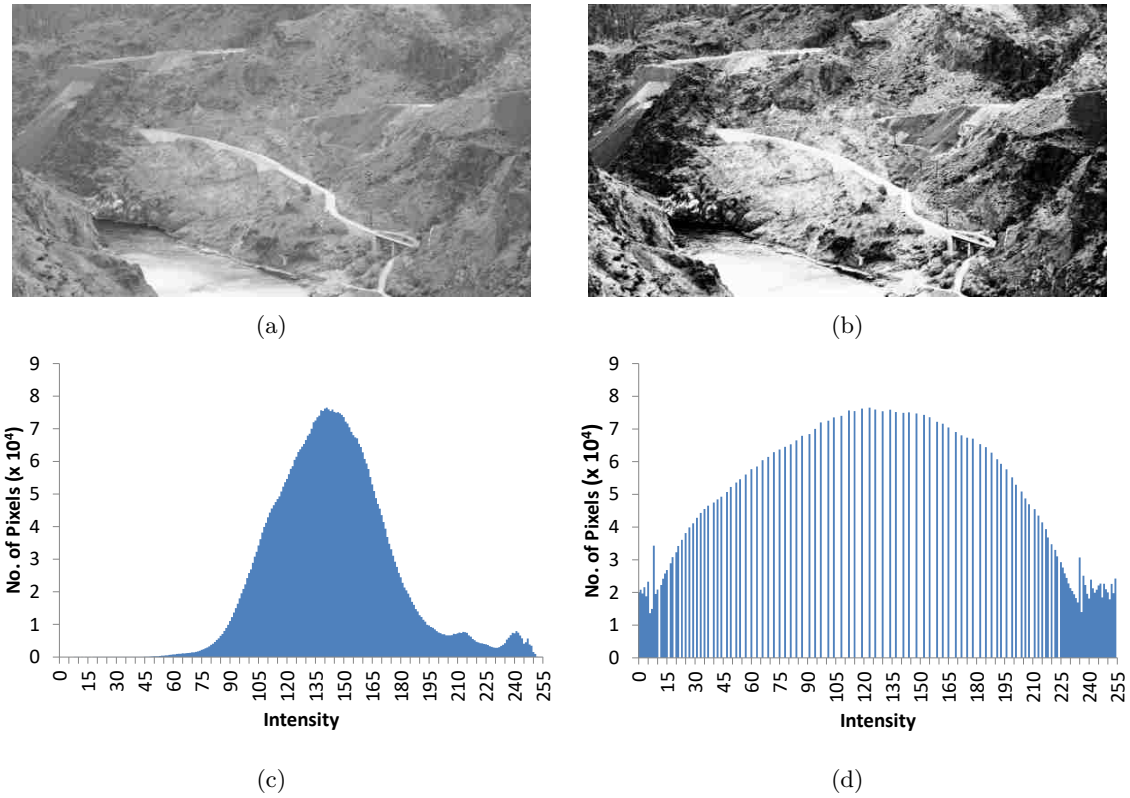


Figure 3.1: Poor contrast image before and after equalization.

(a) original image; (b) image after histogram equalization; (c) histogram of image a (notice how most pixels are clustered around a peak intensity);. (d) histogram of image b (notice how the intensities have been spread apart).

eye (see Figure 3.3).

It is clear that if image capture devices are equipped with means of detecting low illumination and small dynamic ranges in the histogram of the capture, the subsequent captures can be subject to some variant of HE to improve contrast and effectively perform night vision via software. Such devices can be used as an alternative option to current and more expensive night vision devices such as infrared or thermal cameras and to improve current camera technology, ubiquitous in automobiles [64][65], cell phones and other wearables. These cameras can also be very useful for surveillance, nocturnal observations such as wildlife and deep ocean exploration, medical imaging enhancement, search and rescue, and even space exploration.

However, after several experiments, it can be noticed that HE fails to expand the histogram to the whole dynamic range in dark images. Observe that an undesired artifact of applying HE as an alternative to enhance poorly illuminated scenes is the washed out, overly bright nature of the result. The reasons for this behavior were determined and a variation of the algorithm was developed that

Algorithm 1 Classic Histogram Equalization Algorithm

1: Let $\mathbf{I}$ be the original image.
2: Let $I(i)$ be the intensity of pixel i in the image. $0 \leq I(i) \leq X_h$. X_h is the upper bound of the intensity. Typically $X_h = 255$.
3: Let N be the number of pixels in $\mathbf{I}$.
4: Let $\mathbf{H}$ be the histogram of $\mathbf{I}$, i.e. $H(x)$ is the number of pixels in $\mathbf{I}$ such that $I(i) = x$.
5: Let $f(x) = \frac{H(x)}{\underset{x}{N}}$, be the probability function for a pixel in $\mathbf{I}$ to have intensity x.
6: Let $F(x) = \sum_{n=0} f(n)$ be the CDF for $f(x)$.
7: Then, the resulting image $\mathbf{I}'$ is defined as $I'(i) = X_h \cdot F(I(i))$.

results in sharper images with better contrast where the whole range of the histogram is utilized.

3.1 Intensity Weighed Histogram Equalization

Observe how the result from applying HE to a dark image usually looks washed out, and overall, too bright (see Figure 3.3), making some details close to the higher intensities practically indistin-

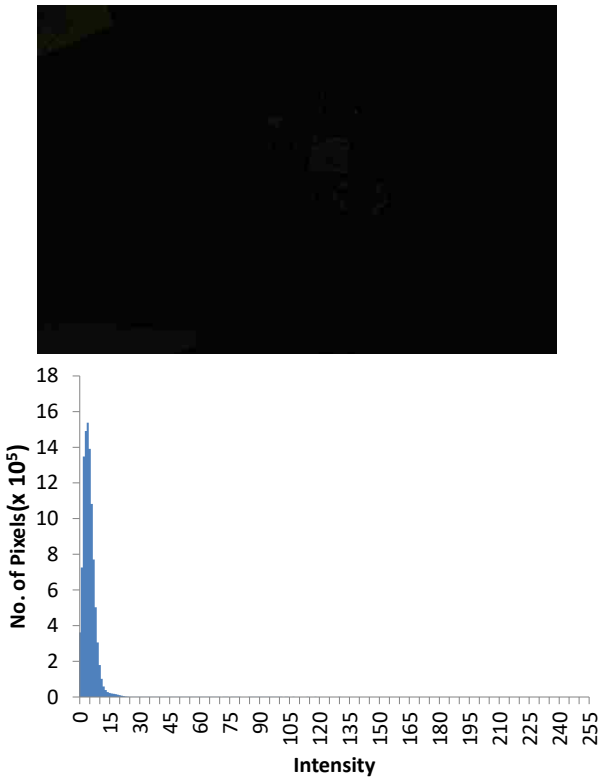


Figure 3.2: Image captured with poor illumination and its corresponding histogram.
Notice how the intensities are clustered together, akin to an image with poor contrast.

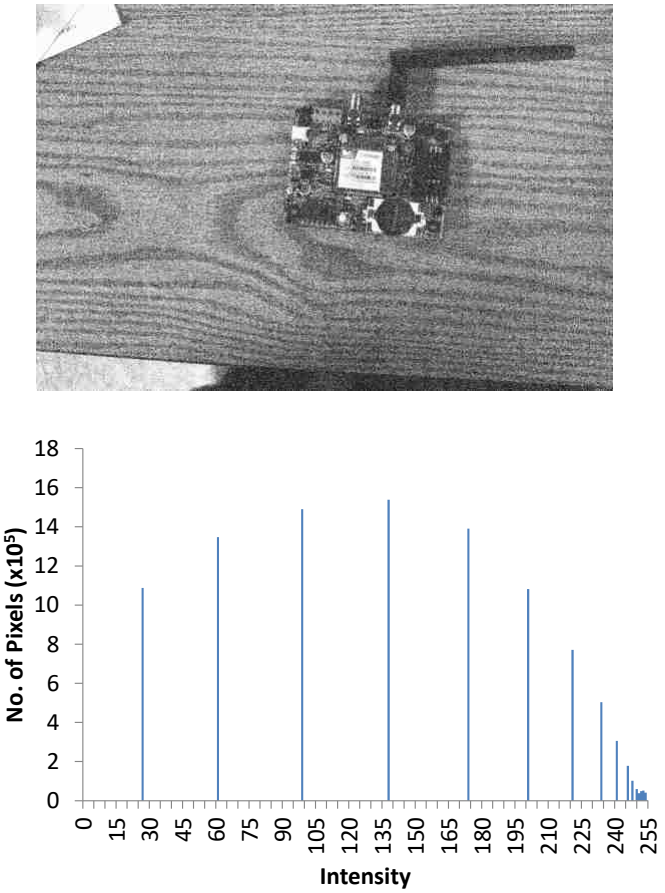


Figure 3.3: Resulting image after applying Standard Histogram Equalization.
This is the result of applying HE to the poorly illuminated image from Figure 3.2.

guishable. This is one of the undesired artifacts introduced by the equalization process. Studying the histograms of resulting images closely, the cause of this undesired effect can be identified and the technique can be adjusted accordingly.

On dark images, the zero and close to zero intensities are predominant, and their effect is what causes the phenomenon highlighted in the histogram in Figure 3.4. Applying classic HE to most low contrast, poorly illuminated images will cause this effect. Notice from the definition of Algorithm 1 that if there are pixels with zero intensity, then $F(0) = u > 0$. So, in the new image all pixels will be $I'(i) \geq u \cdot X_h$. This means that the gap highlighted in Figure 3.4 is of magnitude u intensities. The main problem with this incident is that classic HE will not map the pixels successfully to the whole dynamic range of the histogram when applied to dark images. Also, no matter what, pixels that would actually have zero intensity, will be given an artificial value.

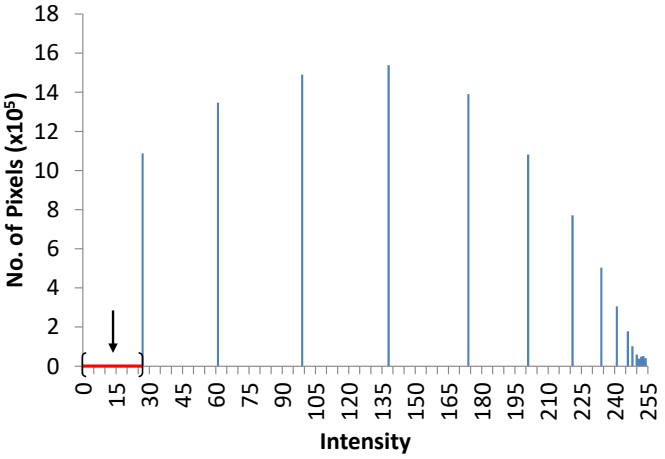


Figure 3.4: Lapse in intensities before the lowest intensity appears.

This lapse in in the histogram for an equalized poorly illuminated image using classic HE causes the standard algorithm not to map successfully the pixels to the whole dynamic range of the histogram.

Intensity Weighted Histogram Equalization (IWHE) is the proposed solution to this problem. It is a variation of the standard HE, and it is also a global equalization method (i.e. it operates over the whole image instead of smaller regions). The main modification in IWHE consists of replacing the computation of the resulting image with the formula:

$$I'(i) = X_h \cdot \min\left[F(I(i))\frac{I(i)}{limitY}, \quad 1\right] \tag{3.1}$$

Where $0 < limitY \leq X_h$ is the intensity value where a desired percentage of the pixels have already appeared.

Experimental results suggest that the best quality images are obtained when the value for $limitY$ satisfies:

$$0.9 < \frac{1}{N}\sum_{n=0}^{limitY} H(n) \leq 0.99 \tag{3.2}$$

That is, $limitY$ is the intensity value in the histogram where 90% to 99% of all pixels in the original image have been accounted for. The smallest intensity where 100% of pixels have been counted is ideal to spread all pixel intensities across the whole dynamic range $(0, X_h)$, but the image may still be a little dark because there is usually a small amount of pixels spread among the intensities after 99% of pixels have been counted. The outlying 1% of the pixels should be set to maximum intensity while spreading the rest.

The parameter $limitY$, within reasonable values, can be used, in effect, to control the brightness

of the resulting image. It inversely affects the overall brightness. Larger values offer more spread of the intensities, but reducing the brightness. Smaller values increase the brightness, but can cause loss of information because too many pixels will be moved to full intensity.

The expression $\frac{I(i)}{limitY}$ used in the new formula to compute the resulting intensity, considers the normalized weight of the pixel intensity when expanding the dynamic range of the histogram. The result, as seen in Figure 3.5, is an image that looks more natural, no longer washed out, more detailed, and arguably less noisy than its classic histogram equalized counterpart. Even the text and bar codes engraved on the chip are visible and readable. Observe that the resulting histogram has the values spread over the whole range instead of starting with a constant gap, and the pixels are distributed over more intensities. The new equalization also ensures that pixels with originally zero intensity, remain at zero. Black pixels had no energy captured by the camera, and thus it is

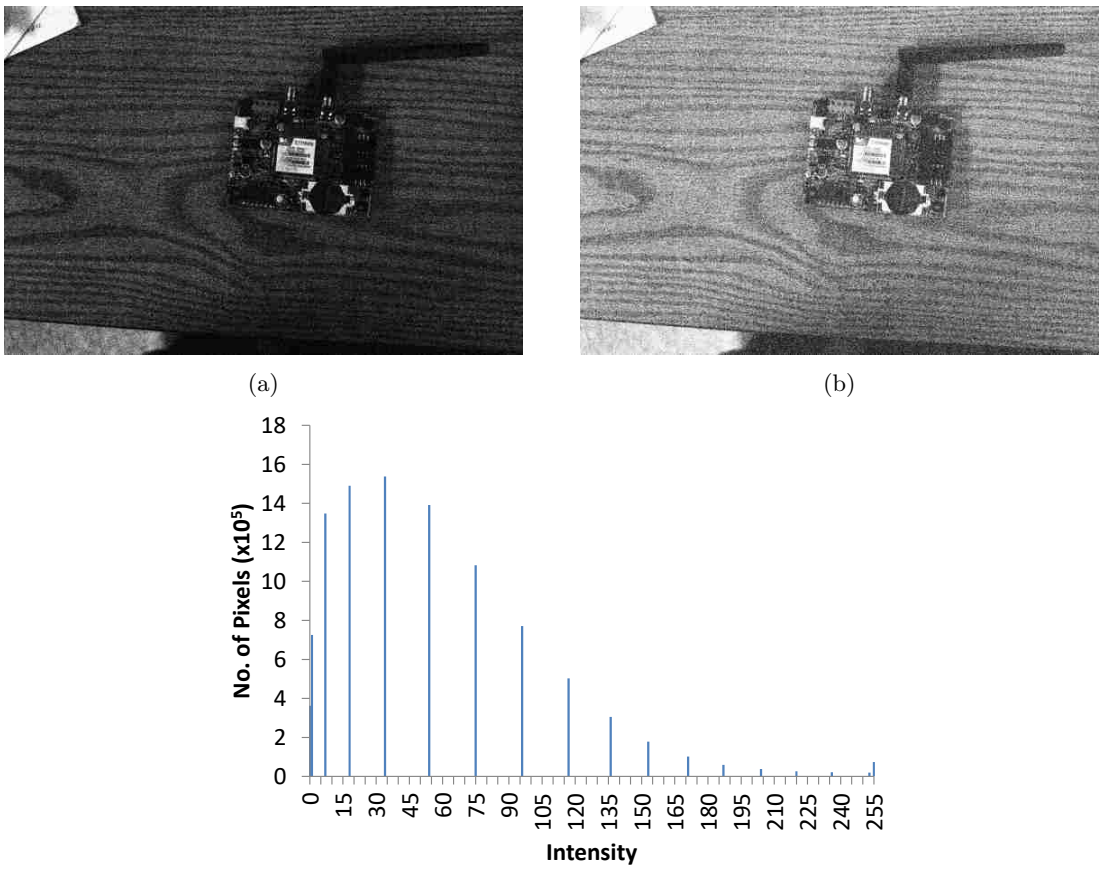


Figure 3.5: Intensity Weighted Histogram Equalization.
(a) Intensity Weighted Histogram Equalization of poorly illuminated image from Figure 3.2; (b) classic histogram equalization of same image for comparison; (c) histogram for image *a*.

artificial to give them values too high. The the new histogram maintains a familiar shape when compared to the original dark image as well. This result was achieved after applying IWHE with a $limitY$ that satisfies the equation $\frac{1}{N}\sum_{n=0}^{limitY} H(n) = 0.98$

3.2 Enhanced Intensity Weighed Histogram Equalization

As shown in Figure 3.5, when a poorly illuminated image is enhanced with IWHE, the resulting histogram has expanded to take over the whole dynamic range and it retains a shape similar to the original image histogram, thus giving the resulting image a more natural look.

Note that the resulting histogram is still a comb containing gaps between intensity values giving the image sharp jumps in intensity among pixel regions. To improve upon this, the pixels can be distributed around their representative intensity value in a normal-distribution-like pattern.

Buckets are created around each individual intensity in the resulting histogram of IWHE. For each bucket, the left limit is defined to be the value halfway between the representative intensity and the neighbor intensity to the left. The right limit is defined in a similar manner. Afterwards, the limits of each bucket are extended based on its size to overlap with adjacent buckets (i.e., larger buckets receive a larger extension to each side). See Figure 3.6.

The N_8 vicinity of a pixel p is the collection of pixels (including p) in the 3×3 matrix of pixels centered at p. The IWHE result is passed through a filtering mechanism where a new intensity value

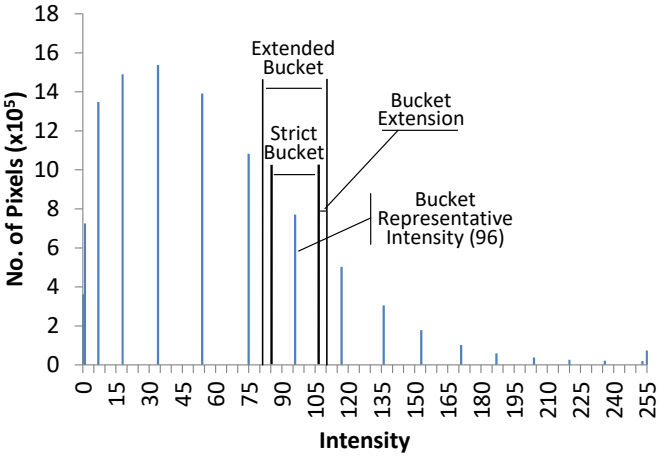


Figure 3.6: Depiction of a bucket around a representative intensity.

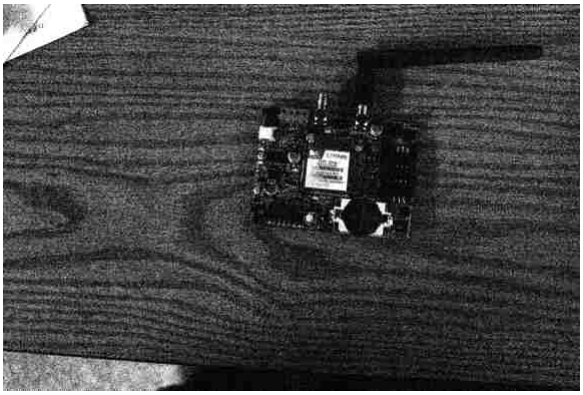
Strict bucket limits are the halfway values between intensities. The extended bucket is the final bucket and is computed by adding the bucket extension to each side of the strict bucket, extending the boundaries to achieve overlap between adjacent buckets. The representative intensity in this example is 96.

$I_1(p)$ is computed based on the intensity of the pixels in the N_8 vicinity of p (for the experiments, the average of the intensities was used, but any other filter that can spread the values in the bucket as defined works as well):

$$I_1(p) = f\left(I(N_8(p))\right) \tag{3.3}$$

Then, the range $\Re$ delimited by $\min(I(p_i))$ and $max(I(p_i))$ is mapped linearly to the range $\Re'$ for the bucket corresponding to the intensity of p. The intensity value for p in the final image, $I'(p)$, is the value $I_1(p)$ mapped from $\Re$ to $\Re'$:

$$I'(I_1(p)) : \Re \rightarrow \Re' \tag{3.4}$$



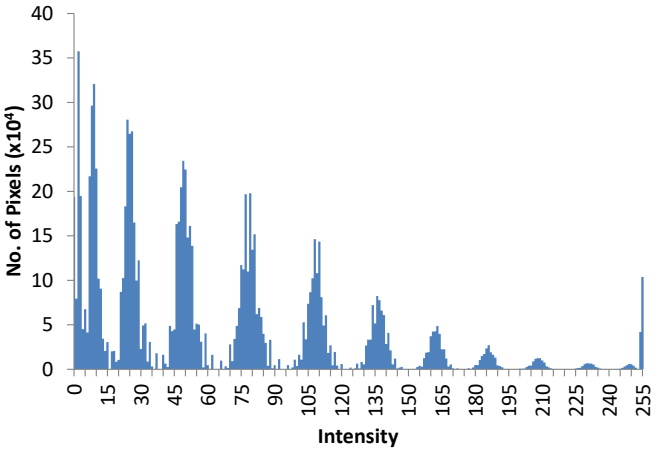


Figure 3.7: Result of applying enhanced IWHE to the image in Figure 3.2.
Notice how the histogram distribution has values over the whole range without holes. The image looks even clearer than the non-enhanced result.

Figure 3.7 shows the final result and histogram obtained after applying this method to the dark image in Figure 3.2. The images enhanced with this method present reduced noise on areas where uniform intensity is expected due to the attenuation of high frequency regions.

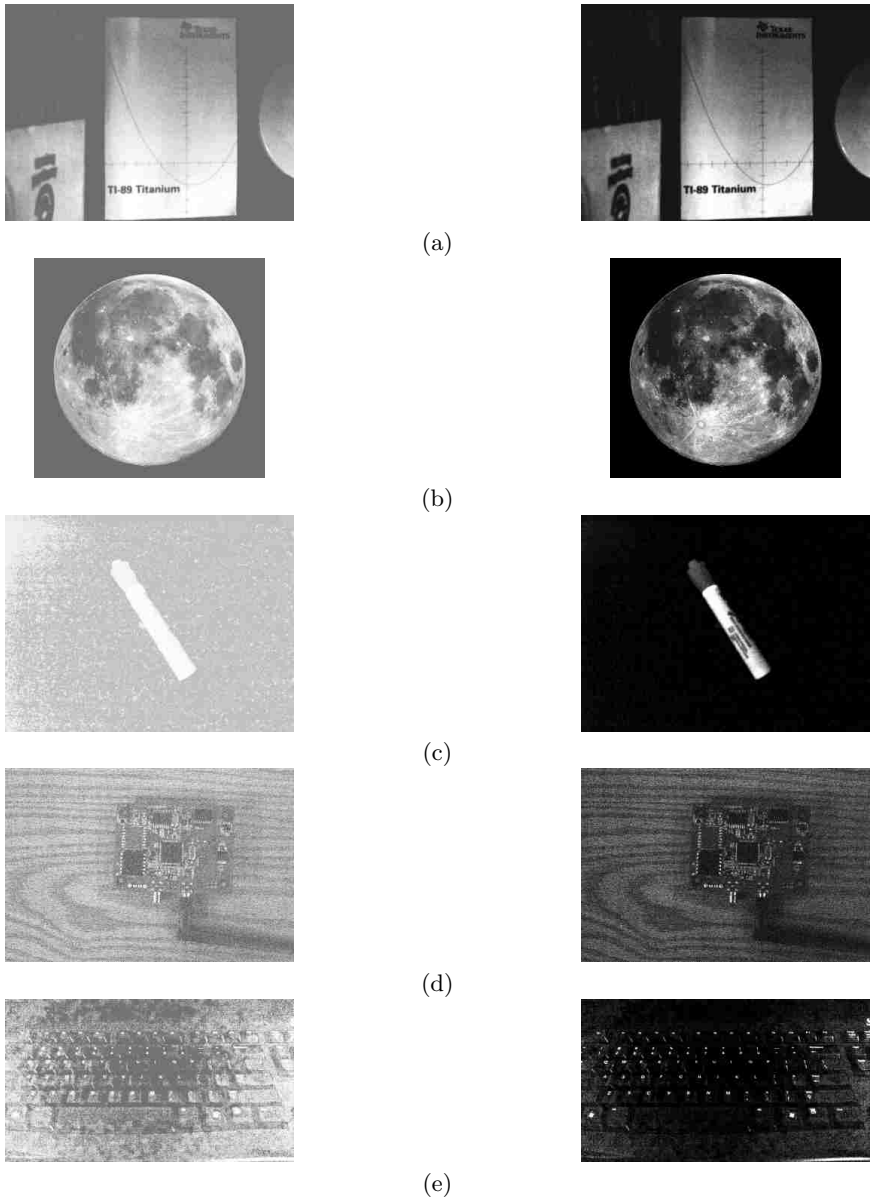


Figure 3.8: Visual comparison between standard HE and Enhanced IWHE.

Images are arranged in pairs where the left image of the pair was obtained through the application of standard HE to a poorly illuminated scene and the right image was obtained through the application of Enhanced IWHE to the same image. (a) book; (b) the moon (original photo obtained from NASA website); (c) marker; (d) circuit board; (e) computer keyboard.

3.3 Experimental Outcomes

A collection of pictures was captured with two different commercially available cameras with no night vision capabilities under poor light conditions without applying flash. The variety of images included night time landscapes as well as objects captured in a closed room without illumination.

Figure 3.8 shows a visual comparison between the application of HE and enhanced IWHE to improve some of the poorly illuminated images captured. The original images are omitted because their poor illumination offers no meaningful detail. With both cameras, the obtained pictures were nothing more than dark images much like Figure 3.2. Notice how images reconstructed with enhanced IWHE retain a more realistic look and details than their washed out, surreal HE counterparts.

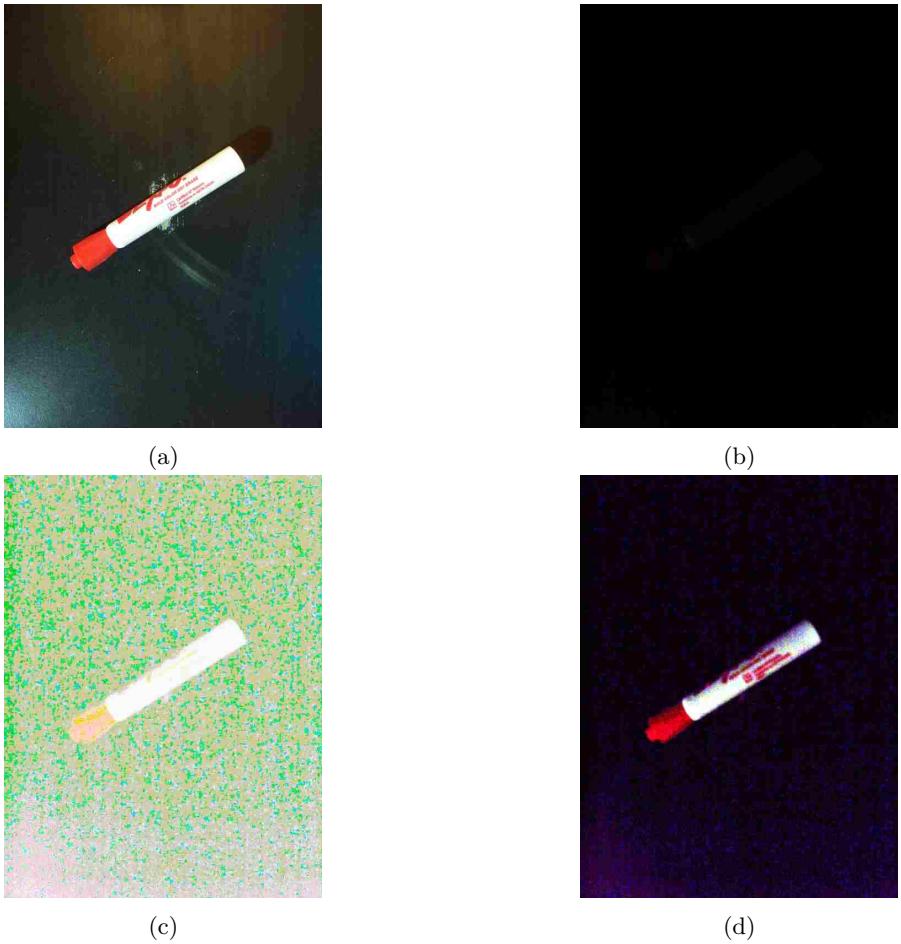
3.3.1 Enhanced IWHE and Color Images

While advanced studies can be conducted for color images regarding equalization and enhancement, the results of applying standard HE and enhanced IWHE to color images are striking. Figure 3.9 shows the comparison between the two methods. The image in (a) shows the captured scene with lights turned on. Image (b) is the same scene with lights turned off resuling in very little energy captured. The histogram of (b) looks very similar to the histogram of Figure 3.2. Image (c) was obtained after reconstructing (b) using the standard HE method. Image (d) was constructed with our enhanced IWHE. Notice how the details, shape and colors of the pen and the whole scene look more natural and closer to the illuminated image after reconstruction using enhanced IWHE instead of the washed out and artifficially colored result of standard HE.

To obtain these images, we applied each discussed technique to each color channel, then the resulting channel intensity was merged with the other resulting intensities to form the final image. For example, from the original image with color channels RGB, we extracted the red intensities for each pixel in R and applied the equalization algorithm, obtaining a new intensity map R'. We repeated this process to obtain the green G' and blue B' results. The new image was formed by combining the computed channels $R'G'B'$.

3.4 Results

Histogram equalization is an inexpensive method to increase the contrast in an image that can be used to enhance visibility in poorly illuminated scenes, effectively providing night vision capabilities. Although, dark images receive improved lighting when equalized, the results look washed out and unnatural, due to the concentration of pixels with zero intensities, and their histogram shows a comb-like shape with large gaps between intensities. The introduction and application of enhanced



(a) (b) (c) (d)

Figure 3.9: Comparison between standard HE and enhanced IWHE for color images.
(a) well illuminated color scene for comparison; (b) scene *a* captured with poor illumination; (c) poorly illuminated color scene after standard HE reconstruction; (d) poorly illuminated color scene after enhanced IWHE reconstruction.

Intensity Weighted Histogram Equalization as a global equalization technique tackles these problems by considering the normalized weight of each pixel when equalizing the image, expanding the histogram to utilize the whole dynamic range, producing crispier, more natural looking, more detailed and less noisy results than standard histogram equalization for dark images.

This technology can be embedded in capture devices that engage Enhanced IWHE when a small dynamic range is detected on the input image histogram, with values clustered close to lower intensities, as a less expensive alternative to other night vision technologies, such as infrared cameras. This will provide enhanced images from captures in low light environments, useful in cameras employed for surveillance, driving, search and rescue, observation of wild life at night or deep ocean, medical imaging enhancement, and many others.

Refer to appendix A for the code implementation of enhanced Intensity Weighted Histogram Equalization in C#.

Chapter 4

Vision System and Optimization for Solar-Panel-Cleaning Robot Architecture

The energy produced by solar panels keeps increasing yearly. Silicon solar panels produce by Panasonic have 22.8% efficiency, making solar panels an economically viable alternative to traditional power. Companies like First Solar it has converted 22.1% of the sunlight energy into electricity using experimental cells made from cadmium telluride. New semiconductor technologies based on Gallium Arsenide and Indium Gallium Nitride (inGan) promise a major improvement over silicon solar panels. Also multi-junction solar panels have higher production of electric energy. Gallium nitride Solar panels are relatively inexpensive, they last for a considerable amount of time, and every year we see new large scale installations as well as smaller for houses and commercial buildings. Many of these large scale installations are in desert environments, where strong winds blow sand and dust onto the panels inhibiting the energy productions. In addition to that birds migrating from colder to warmer climates choose the solar panel sites as a resting place and the bird droppings on the panels inhibit the energy production. Also small animals with sharp teeth roaming on the solar panel site at night, or using the space under the panels to protect themselves from the hot summer days and the cold winter nights cut the cables with their sharp teeth thus disabling the panels. Finally vandals throwing stones and other objects through the fence could damage the glass or other parts of the panels.

Having a maintenance crew to look after the panels is an expensive proposition which introduces human problems.

The robot "Helios" is part of the research project funded by the National Science Foundation. Its purpose is to inspect every panel, as well as its cable connections, and decide if the panel needs cleaning or not.

All the pertinent information, which includes the panel ID, and the details related to the panel status are transmitted wireless to a file server and stored in the data base. For each panel needed cleaning the robot returns during the night when the panel does not produce any energy and cleans the panel.

The robot (Figure 4.1), consists of a mechanical component, an electro-mechanical component, an electronics system, and a software suite.

The mechanical and electromechanical parts consist of an all-terrain vehicle, two electric brushless motors, a telescopic vision system, and telescopic cleaning system with a brush, stepper motors controlling the telescopic vision system, and the telescopic vacuum system, and a small vacuum system with a brushless electric motor. The vacuum system traces the solar panel from top to bottom and cleans it.

Figure 4.1: Prototype of the robot Helios.
The robot features the folding telescopic support of the vision system consisting of four cameras in a cross configuration and a noninvasive laser.

The software consists of a scalable operating system, an intelligent vision system with pattern recognition, a communication software system, and an intelligent navigation system.

Our research pertains to the creation of the robot's vision system, classification algorithm and optimization algorithm to determine the optimal time to perform cleaning on the solar panels to minimize the impact in energy production.

4.1 Robot-Server System Overview

The Robot architecture consists of the hardware and the software. The majority of the effort is in the software, making the robot to be a specialized computer on wheels. The hardware consists of three parts.

The mechanical component comprises the vehicle which is an all-terrain using an army-tank-like continuous track, with two brushless motors, one in each front wheel. Each one of the motors is controlled by a separate electronic speed controller. The main reason for this is to enable the robot to make turns. Thus in order for the robot to turn left; we increase the speed on the right motor and decrease the speed on the left. The mechanical part also includes a telescopic vision system which provides the input to the intelligent software that understands a panel's boundaries and decides if a panel is clean or needs to be cleaned. The mechanical part also includes the vacuum and a brush used to loosen material on the panel in order for the vacuum to clean the panel.

The electromechanical system includes the two brushless motors of the vehicle part of the robot, the stepper motors of the telescopic vision system, as well as the stepper motors of the telescopic vacuum system, and the brushless motor of the vacuum system.

The electronic part consists of a printed circuit board (PCB) connected to the four cameras via four BNC connections, having a number of sensors used as part of the navigation system, GPS, accelerometer, magnetometer, a DSP that takes as input the images obtained by the four cameras via the BNC connections, stores the images in four memory chips on board, performs the classification algorithm and makes a decision if the panel is clean or needs cleaning. The decision is passed to the transceiver on board to transmit it wireless to the file server. The PCB board also contains a control system that uses an ARM chip to compress the images obtained by the four cameras, passes them to the transceiver on board which transmits them to the file server. A transceiver, a voltage amplifier that amplifies the voltage from 1.5V to 12V, and an antenna. The PCB board is connected to a computer board via a PCI express connection.

The software consists of the classification algorithm, that takes as input the images of the panel obtained by the four cameras, applies the classification algorithm and decides if the panel needs cleaning. The vision system also inspects the electric cables underneath the panel and decides if all

connections are good or not.

4.2 The Vision System

The vision system is a critical part of the robot and it consists of four identical cameras and a non-invasive laser. Figure 4.2 depicts the schematics of our system. The cameras are mounted on a frame having a cross configuration. Each leg of the cross is telescopic having the ability to increase or decrease the distance of the camera from the laser so that will decrease or eliminate occlusions. The distance of each camera from the center of the cross is controlled by a stepper motor and it is always known. The noninvasive laser is in the middle of the cross and is equidistant to the four cameras. When distances are to be resolved the noninvasive laser is activated and its light is registered by each one of the four cameras. The cameras are parallel to one another and also parallel to the laser. During calibration for every pixel in the image space registering the laser light dot on the object, the angle between the line defined by the image center and the pixel, and the line defined by the pixel and the laser dot on the object, is computed and stored in a lookup table. Thus during the focus on a panel if for a camera the laser dot is registered by a certain pixel then we know the angle formed by the line between the pixel and the laser dot and the line between the pixel and the camera center.

Figure 4.3 shows each of the four cameras with each own local coordinate system. In the default state the laser and the cameras are parallel and the distance of each camera from the laser is fixed.

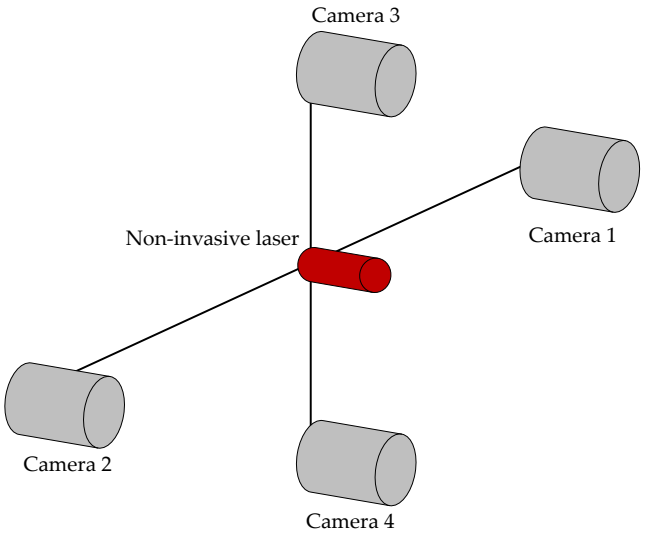


Figure 4.2: The schematics of the vision system.
The vision system consists of four identical cameras and a noninvasive laser. The purpose of the system is to resolve distances, and enable the creation of 3D vision.

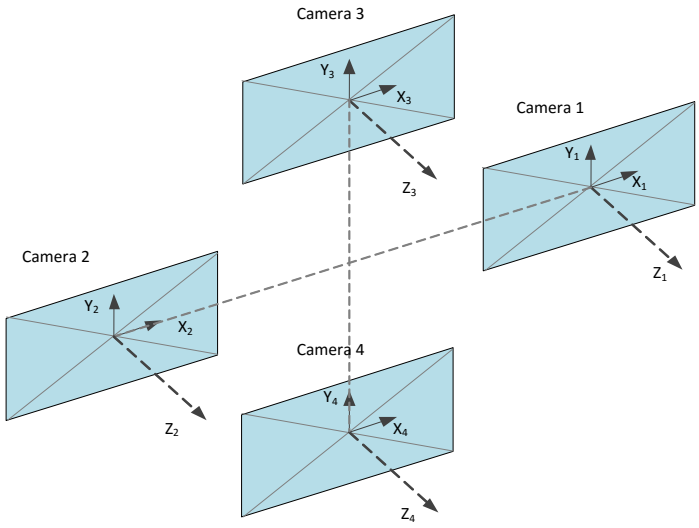


Figure 4.3: Camera coordinate system.

Each one of the cameras has a local coordinate system (X_i, Y_i, Z_i), and an image space coordinate system (x_i, y_i), $i \in \{1, 2, 3, 4\}$. The local coordinates can easily be transformed to a global coordinate system via translation and rotation.

The default state is the one we use in order to position the camera system at a fixed distance from the panel. In this state the laser dot has exactly the same Z coordinate for each one of the four local camera coordinate systems.

The Z coordinate for a pinhole camera (Figure 4.4), is given by equation 4.1. If B is the known distance between the focal points of cameras 1 and 2 then equation 4.4 gives an estimate of the distance of the laser focal point to the laser dot on the panel. In a similar way we can obtain another estimate of Z from the cameras 3 and 4. Two estimates of Z can be obtained from cameras 1 and 3, another two from cameras 2 and 3, another two estimates of Z from cameras 1 and 4, and finally another two estimates of Z from cameras 2 and 4. An estimate of Z can be obtained from camera 1 and the laser, similarly another from camera 2 and the laser, from camera 3 and the laser, and from camera 4 and the laser.

Thus, 14 estimates of the distance of the laser focal point from the laser dot on the panel can be obtained. All these estimates are slightly different due to the noise in the system. the average $\bar{Z}$ of these fourteen estimates is a more accurate estimate of the distance of the vision system from the panel. The DSP of the PCB board computes this distance relatively fast and positions the vision system at a fixed distance above the panel. This distance is the same at every inspection of every panel. The details of the geometry and formulas are given below.

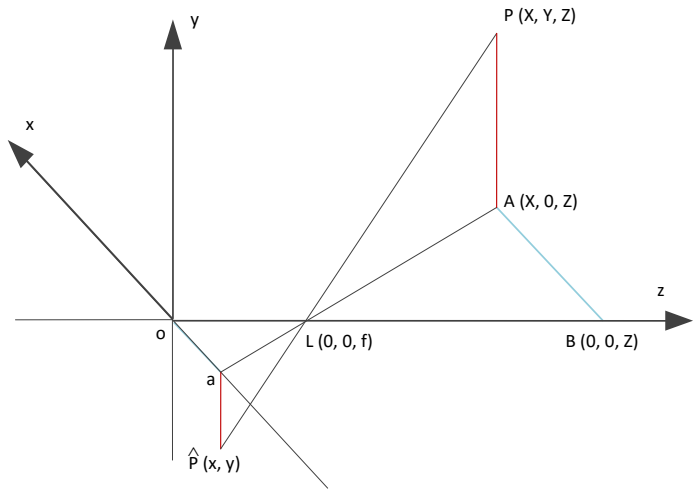


Figure 4.4: Pinhole camera model.

L is the camera pinhole or the center of the lens. O is the center of the imager, as well as the origin of the local coordinate system (X, Y, Z), and the image coordinate system (x, y). The point $P(X, Y, Z)$ is projected to the point $\widehat{P}(x, y)$.

The pinhole camera model can also represent the modern CCD or CMOS cameras with the chip replacing the film and the center of the lens replacing the pinhole. In Figure 4.4, O is the center pixel of the imager chip, and also the center of the local coordinate system. $L(0, 0, f)$ is the lens center, $P(X, Y, Z)$ is a point in the space projected to the point $\widehat{P}(x, y)$ on the imager. Then from the similar triangles $\triangle ALB$ and $\triangle OaL$, if $\lambda = \overline{LO}$, $\overline{Oa} = x$, $\overline{LB} = Z - \lambda$, we have:

$$-\frac{Z-\lambda}{\lambda} = \frac{X}{x}$$

or

$$Z = \lambda\left(1 - \frac{X}{x}\right) \tag{4.1}$$

From equation 4.1 we have:

$$Z = \lambda\left(1 - \frac{X_1}{x_1}\right) \tag{4.2}$$

$$Z = \lambda\left(1 - \frac{X_2}{x_2}\right) = \lambda\left(1 - \frac{X_1 + B}{x_2}\right) \tag{4.3}$$

Thus from 4.2 and 4.3 we get:

$$Z = \lambda\left(1 - \frac{B}{x_2 - x_1}\right) \tag{4.4}$$

Due to the noise in the system we obtain four different estimates of Z from the four laser-camera geometries: two estimates using the geometry of two horizontal and two vertical cameras. Eight different estimates of Z can be also obtained using the geometry of any two adjacent cameras. Each one of these estimates is a random variable with mean Z_t, the true value of the distance, and variance σ_i^2, $i \in \{1, 2, ..., 14\}$. According to the central limit theorem the average $\bar{Z}$ of these estimates of the distance is normally distributed with mean Z_t, where Z_t is the true distance, and variance

$$\sigma_{\bar{Z}}^2 = \frac{\sum_{i=1}^{14} \sigma_i^2}{14}.$$

Let S_Z be an estimate of the standard deviation of the random variable Z based on the estimates of the true distance, and if we denote by Z_t the true value of Z then the statistic:

$$\frac{\bar{Z} - Z_t}{\frac{S_Z}{\sqrt{14}}}$$

has the t distribution with 13 degrees of freedom. Therefore

$$\bar{Z} - \frac{S_Z}{\sqrt{14}} t_{1-\frac{\alpha}{2},13} < Z_t < \bar{Z} + \frac{S_Z}{\sqrt{14}} t_{1-\frac{\alpha}{2},13} \tag{4.5}$$

with probability $1 - \alpha$.

Formula 4.5 provides a measure of how accurate the distances estimated by our system are. For example if $\bar{Z} = 100$cm and $S_Z = 3$cm, then with probability 0.95 (95%) the true distance Z_t , is $98.27\text{cm} < Z_t < 101.73\text{cm}$.

4.3 Overview of the Classification Algorithm

The original classification algorithm is based on spectral decomposition as suggested in "Using Spectral Decomposition to Detect Dirty Solar Panels and Minimize Impact on Energy Production" [87].

The energy produced by solar panels declines in proportion to the amount of light blocked by the deposits of dust and other contaminants accumulating on the surface of the photovoltaic cells.

The classification system employed by the robot to determine if a panel is clean or not is based on the Mahalanobis distance, which is the relative, statistical measure of the data point's distance from a common point. The classifier we developed for the system recognizes the state of the panel with an accuracy above 90%.

To evaluate the accuracy of the classification system we obtained sample images from solar panels using a camera configuration similar to what will be part of the finalized vision system of the robot.

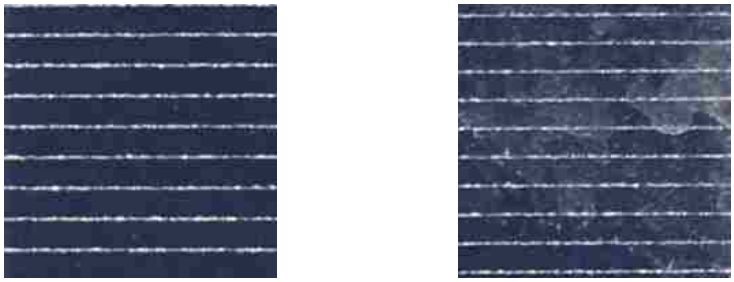


Figure 4.5: Solar panel samples.
Left) clean panel; right) dirty panel [87].

We decided to employ three groups of training data, where each group contains one clean sample set and one dirty sample set.

1. The first group contains data from the same panel.
2. The second group builds upon the first group by incorporating data from another panel with similar photovoltaic cell structure.
3. The third group does not build upon the first and second group. Instead, it incorporates data from two panels of similar characteristics, but with a lighter shade of blue than the panels from the other groups.

See Figure 4.5 for two samples taken by the vision system of clean and dirty solar panels.

Then we utilized the jackknifing technique to estimate the precision of the classifier through the formula:

$$Accuracy = \frac{TN + TP}{TN + TP + FN + FP} \tag{4.6}$$

where TN (true negative)/FN (false negative) are the number of samples correctly/incorrectly classified as clean, and TP (true positive)/FP (false positive) are the number of samples correctly/incorrectly classified as dirty [87].

Table 4.1 [87] shows the experimental results of equation 4.6 from applying Jackknife test on the sample data.

Table 4.1: Results of applying Jackknife test on all three groups of sample data.

Group	TN	FN	TP	FP	Accuracy (%)	Misclassification Error
1	12	0	12	0	100	0
2	17	3	19	1	90	0.10
3	17	2	19	0	94.4	0.0526

4.4 Cleaning Schedule Optimization Algorithm

When the vision and classification system determines that a solar panel is dirty, that information is stored in the server instead of having the panel cleaned right away. The ideal moment to perform the cleaning depends on the cost of maintenance vs. the loss of energy production, accounting for variables such as frequency of rainfall.

To compute the energy generated in a time interval we used the trapezoid variation of the Riemann Sum:

$$E = \frac{1}{2} \sum_{i} (f(x_{i+1}) + f(x_i)) \cdot \Delta x_i \tag{4.7}$$

where $f(x_i)$ is the power measured at sample x_i. Δx_i is the time lapse between measurements x_i and x_{i+1} which is constant for all measurements since sampling was taken at uniform intervals during the experiment. Our Power (P) sample was computed from measured solar panel output given by Voltage (ΔV) and Current (I) using Ohm's Law variant:

$$P = \Delta V \cdot I \tag{4.8}$$

See Figure 4.7 for an example of captured data. In this example, the energy generated measured from the sampled data on May 21^{st}, 2012 from 0 hours to 2359 hours using equation 4.7 was $E = 7.20kWh$.

We collected weather data from Clark County Regional Flood Control District (Flood Control)

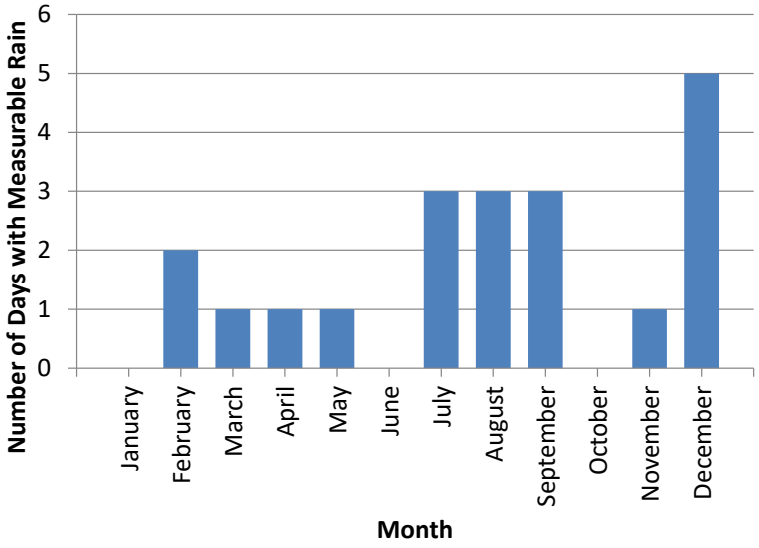
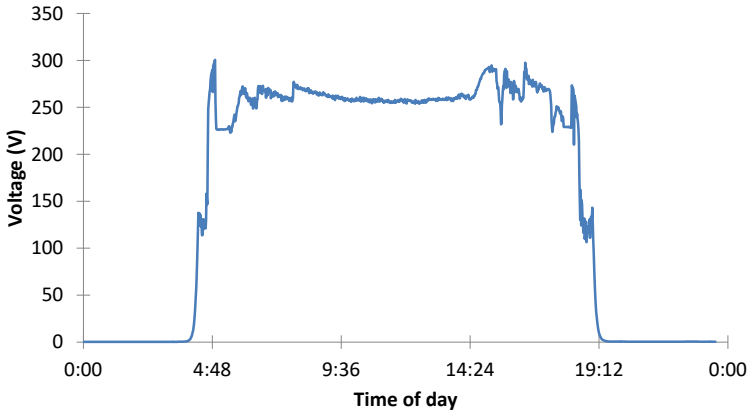
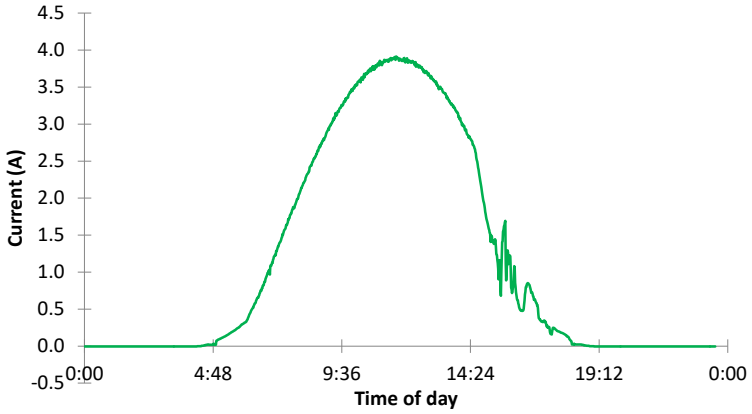


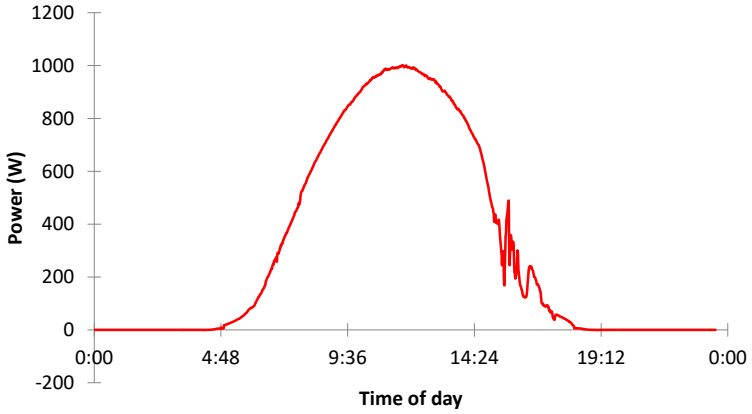
Figure 4.6: Number of days with measurable rainfall per month on 2014.
Measurements according to NOAA at Henderson station, Mojave Desert, Nevada.



(a)



(b)



(c)

Figure 4.7: Solar panel data collected during a typical sunny day in May.
(a) Voltage (in Volts) vs. Time; (b) Current (in Ampere) vs. Time; (c) Power (in Watts) vs. Time.

and National Oceanic and Atmospheric Administration (NOAA) to determine the frequency of measurable rainfall on arid zones, such as the Mojave Desert. We decided the measurable amount based on how much water was necessary to have an effect on dust and particle accumulation on a solar panel surface.

We arranged the rainfall data per month and computed the number of days per month with a measurable rainfall throughout 5 years (from 2011 to 2015). Using a best of fit test on the collected weather pattern data , we determined that the number of days with measurable rainfall can be modeled using the Poisson distribution:

$$P(X = x) = \frac{\lambda^x e^{-\lambda x}}{x!}, \qquad x \in \{0, 1, 2, ...\} \tag{4.9}$$

where λ is the mean and variance of the distribution. The parameter λ depends on the particular location as well as on the Sun spot activity on said location.

The dust accumulation on the panel in a day is a random variable. Because power loss is directly related to accumulation of dust and particles on the panel surface we can model the power loss (in USD) also as a random variable. After experimentation, we found that the power loss in a day follows an exponential distribution with parameter α and has the form:

$$f(x) = \alpha e^{-\alpha x}, \qquad x > 0 \tag{4.10}$$

where the parameter α can be estimated from the data in a controlled environment. This function measures the probability that the power loss is $x_1 \leq x \leq x_2$.

If X_i is a random variable and $X_i \sim$ Exponential(α), then $\sum_{i=1}^{k} X_i \sim$ Erlang(k, α). Therefore, provided no measurable rain has fallen in k days, then, the sum of power loss in k days follows an Erlang distribution with parameters k and α:

$$g(x) = \frac{x^{k-1}\alpha^k e^{-\alpha x}}{\Gamma(k)}, \qquad \alpha > 0, k \in \{1, 2, ...\} \tag{4.11}$$

where $g(x)$ measures the probability that the power loss is $x_1 \leq x \leq x_2$ after k days of no measurable rainfall.

Finally, if C is the cost of cleaning a panel, then the expected power loss (in USD) in k days after the panel was cleaned, and no rainfall has occurred is:

$$E(x) = \int_0^{+\infty} \frac{x^k \alpha^k e^{-\alpha x}}{\Gamma(k)} dx = \frac{k}{\alpha}, \qquad \alpha > 0, k \in \{1, 2, ...\} \tag{4.12}$$

This means that if $C \leq \frac{k}{\alpha}$ or $k \geq C\alpha$ for the first time (minimum k) then, it is the optimal time to clean the panels. As the cost C decreases, the k decreases and the power increases.

4.5 Results

The described architecture of the Robot-Server system allows for the proper positioning of the robot cameras used to capture solar panel images. The classification algorithm recognizes if solar panels are dirty based on statistical pattern recognition. Dirty panels and marked and stored in the server. The system is able to detect whether the loss of energy is above a critical value in which case, maintenance and cleaning crews, or better yet, the robot itself is dispatched to clean the marked panels. Testing the system in the solar panel lab offered excellent results minimizing power loss successfully when compared to a regular cleaning schedule regardless of cleanliness and energy loss.

Chapter 5

Details on Deep Learning and Artificial Neural Networks

In this chapter we discuss the basics of modern Artificial Neural Networks. We survey techniques employed to improve convergence rate and speed up neural network learning as well as explain the inner workings of Convolutional Neural Networks (CNN) and implementation details. The main goal in this chapter is to replicate the implementation of current state-of-the-art networks using our own implementation to test our knowledge about the modern techniques that went into those networks. Knowing the internal works and details of modern neural networks allows us to build upon their architecture to identify areas that need improvement or that can be built upon to either perfect these designs or to produce our own architectures.

In the following sections we offer an in-depth overview of the fundamental concepts and techniques necessary to understand the details of current state-of-the-art neural networks and deep learning research.

We will cover necessary concepts regarding Fully Connected (vanilla) Feedforward Neural Networks, their architecture and training using stochastic gradient descent along with a plethora of methods to improve convergence rate. These concepts are expanded to the more recent CNNs especially applied today to image and signal processing and classification, giving birth to modern buzz terms in the area such as "deep learning." We give details and pointers to adapt vanilla network methodologies to the more involved CNNs.

5.1 Fully Connected Feedforward Neural Networks

A *Feedforward Neural Network* (FNN) is an artificial neural network in which its basic computational units do not form cycles. Artificial neural networks are graph-like computational models inspired in the biology of the neural cortex, used in classification and approximation problems. A fully connected FNN is also known as a multilayer perceptron.

5.1.1 Basic Computational Unit

The basic computational unit of the neural network is the neurode and it is represents the simplified mathematical model of a biological neuron. The terms *computational unit*, *neuron* or *neurode* are used interchangeably to refer to the artificial neuron or neurode. Figure 5.1 shows the model of a specific neuron i.

The incoming n arrows have weights $w_0, w_1, ..., w_n$, and represent weighted connections from other neurons. The vector of incoming weights is $\mathbf{w}$. These are the parameters that get modified during training of a neural network.

The arrows labeled $x_0, x_1, ..., x_n$, represent the activation value of the connected neurons that serve as input for this neuron. The vector of inputs is $\mathbf{x}$.

The activation value for this neuron is z_i. It is computed as $z_i = f_i(S_i)$, where $f_i(x)$ is referred to as the *activation function* or *non-linearity*. While there are many activation functions, for the purpose of back propagation (in a later section) f_i is often a non-linear, non-polynomial, continuous, differentiable, function due to the shown fact that neural networks with continuous non-linear and non-polynomial activation units in, at least, one hidden layer possess the universal approximation property [29][45].

The parameter S_i is the result of the function $\mathbf{S}_i$ and has no formal name in literature. It is

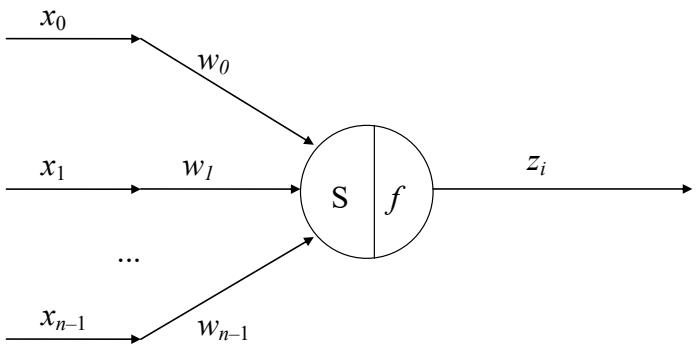


Figure 5.1: Model of a neurode, basic computational unit of a neural network.

often expanded and included as part of the activation function, but we like to call it *weighted (or total) input* or *local receptive field* as suggested by LeCun *et al.* [42] because its task is to combine the incoming weights and inputs somehow for the activation function. The most common weighted input function is the linear combination or dot product:

$$S_i = \mathbf{S}_i(\mathbf{w}, \mathbf{x}) = \sum_{j=0}^{n} w_j x_j + bias_i \tag{5.1}$$

where $bias_i$ is an independent parameter for each neuron i, called *bias*. It is used to reposition the linear combination in the n-dimensional space to better represent the distribution of the input. The bias is often considered a weight for a constant input of 1 because it is another parameter that is modified during training.

Different neurons in a neural network may present different weighted input functions and different activation functions.

While dot product is the most common, it is not the only weighted input function. Other commonly used functions include the maximum or average of either the inputs or the product of inputs and weights. This is another reason why we like to have a separate term for this part of the activation function.

The mathematical function of a neuron is to divide the n-dimensional hyper space with the hyperplane defined by $\mathbf{S}_i(\mathbf{w}, \mathbf{x})$ and to compute a new output (activation) based on how far is the input vector from the defined hyperplane.

A collection of neurons acting over an input vector is able to divide the hyperspace into complex regions. This behavior allows for exact approximation or for generalization of functions with strict subdivisions or relaxed subdivision respectively.

Common Activation Functions

While the classic activation functions used in early neuron models called McCulloch-Pitts nodes were the step functions of the form [50]:

$$f(x) = \begin{cases} 1, & \text{if } x \geq t \\ 0, & \text{otherwise} \end{cases}$$

where t is some threshold value, differentiable activation functions are preferred as activations in order to be able to perform training using back propagation with gradient descent or second order methods.

The most common activation functions used today are the logistic sigmoid (often referred to as log-sigmoid or just sigmoid), hyperbolic tangent [59][56], radial basis function (RBF) [57], rectifier

[21] and softmax.

The sigmoid and hyperbolic tangent functions are known as sigmoidal or sigmoid-like functions, having an "S" shape. These functions were used initially to substitute step functions because of their similarity, but with the added benefit of being continuous and differentiable.

The sigmoid is defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5.2}$$

The hyperbolic tangent behaves much like the sigmoid, but the inflection point is more constrained:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{5.3}$$

Figure 5.2 shows the plot of the activations for the sigmoidal functions.

The RBF was studied as a variant to the sigmoid activations. It has a bell shaped curve similar to a Gaussian (from which RBF draws inspiration) and is defined as:

$$f(x) = e^{-\beta(x-\mu)^2} \tag{5.4}$$

where μ is the value where the curve peaks and β controls the opening of the curve. Training radial basis networks is a little more involved than sigmoid networks because of the extra parameters μ and β that need to be either trained, or decided before training.

An alternative to RBF was proposed to potentially help with optimization of the training process

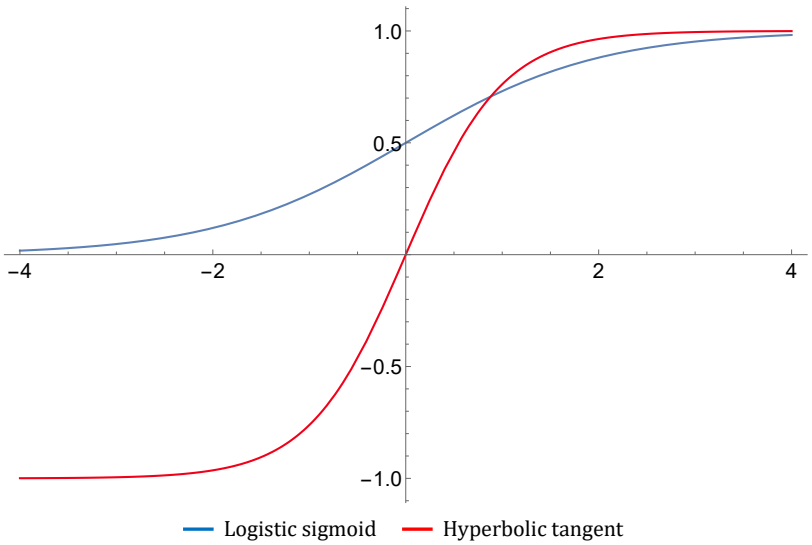


Figure 5.2: Comparison between Logistic sigmoid and Hyperbolic tangent.

because it involves no exponential calculation [9]:

$$f(x) = \frac{1}{x^2 + 1} \tag{5.5}$$

This function also displays a bell shape and a derivative that behaves similarly to RBF. Compare the variants to RBF in Figure 5.3.

Radial basis functions, however, have been shown to work well in approximators, but do not generalize as well from reduced training data as sigmoid activations do [6].

A rectifier activation is a special case of activation functions because it is not differentiable at 0. It is defined as:

$$f(x) = \max(0, x) \tag{5.6}$$

For training purposes, the derivative at 0 is usually taken from the left only, thus:

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

Neurons with a rectifier activation function are commonly referred to as *Rectified Linear Units* or *ReLU*.

It has been experimentally shown that ReLU in hidden layers, and especially in convolutional layers, can dramatically improve the convergence speed of a neural network, but they suffer from the "dead" ReLU problem: once the value of x reaches 0, the gradient passing through the neuron vanishes and the neuron stops learning and firing, effectively "dying."

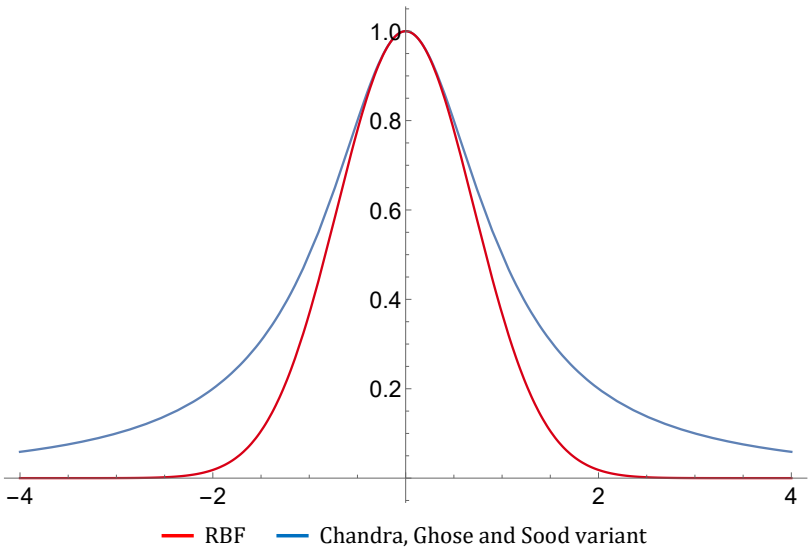


Figure 5.3: Comparison between Radial Basis Function variants.

One approach to reduce the dead ReLU problem is the utilization of small learning rates. Another approach is to modify the neuron into whats called a leaky ReLU [46]:

$$f(x) = \max(x \cdot leak, x) \tag{5.7}$$

where *leak* is a small, positive value, usually below 0.01 and sometimes it is turned into a parameter that is learned.

The softplus function is another alternative to reduce the dead ReLU impact [17]. It is defined as:

$$f(x) = \ln(1 + e^x) \tag{5.8}$$

Recent studies have brought Exponential Linear Units (ELU) as yet another alternative to ReLU [12]. These units have shown to provide even better recognition accuracy than ReLU. The activation function is defined as:

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ a(e^x - 1), & \text{otherwise} \end{cases} \tag{5.9}$$

where $a \geq 0$ is a constant hyperparameter to be tuned before training.

The first derivative of the ELU activation is:

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ ae^x, & \text{otherwise} \end{cases}$$

Figure 5.4 shows a comparison among the ReLU variants. Each iteration is aimed to reduce the dead ReLU problem and increase the differentiation at the point $x = 0$. Notice that for ELU

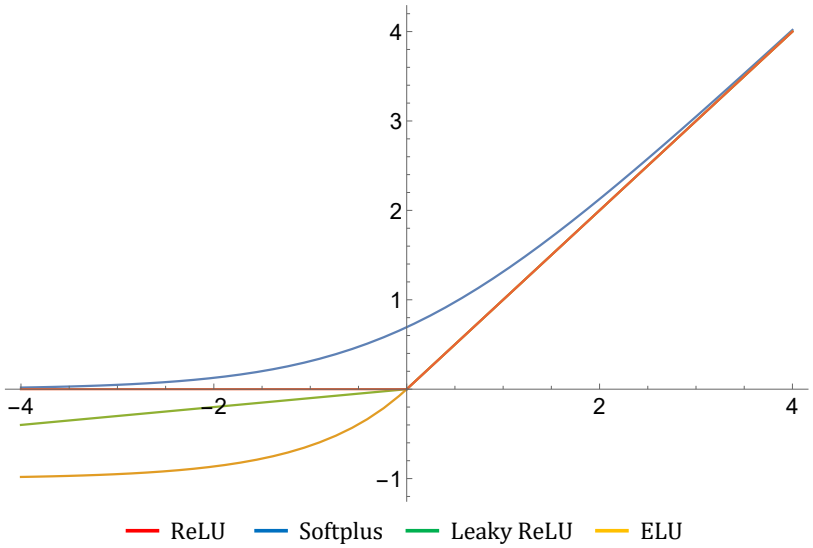


Figure 5.4: Comparison among Rectified Linear Unit variants.

activation, if $a = 1$ then, the first derivative exists at $x = 0$ and $f'(0) = 1$.

The softmax activation is usually applied to output layers in classification problems because of its nature of converting the output into a probability function. The softmax activation is defined as:

$$f(S_i) = \text{softmax}(S_i, \mathbf{S}) = \frac{e^{S_i}}{\sum_{j=1}^{N} e^{S_j}} \tag{5.10}$$

where S_i is the value of the weighted input for neuron i, $\mathbf{S}$ is the vector of all weighted inputs for all the neurons in the same layer as neuron i and $N = ||\mathbf{S}||$ (cardinality of $\mathbf{S}$).

Clearly:

$$\sum_{i=1}^{N} f(S_i) = \sum_{i=1}^{N} \frac{e^{S_i}}{\sum_{j=1}^{N} e^{S_j}} = 1.$$

This is why this function is mostly used as activation for the last layer for classification problems where the target output is a vector that expresses the probability of the input to belong to each class. For back propagation purposes later:

$$f'(S_i) = \frac{e^{S_i}\left(\sum_{j=1}^{N} e^{S_j} - e^{S_i}\right)}{\left(\sum_{j=1}^{N} e^{S_j}\right)^2}. \tag{5.11}$$

The softmax should only be applied for single class classification problems. This is, if there are two or more possible classes in the input at the same time then, a different activation should be used in the output layer. In this case, a sigmoid is preferred for all output neurons.

5.1.2 Architecture and Forward Propagation

The power of neural networks is in the massive capacity for parallelization. Each neuron is able to perform its computation independently from all other neurons (except, maybe, from those upon which its input depends). In layered architectures, all neurons in a single layer can process their inputs in parallel.

Fully connected FNNs are organized in layers of neurons. Each neuron in a layer is fully connected to all the neurons in the previous layer. This means that the outputs of all the neurons in layer l are inputs to every neuron in layer $l + 1$.

Figure 5.5 depicts a generic FNN architecture. This network contains L layers. Layer 0 is often called the *retina* or *input layer*. Normally, computational units are not represented for the input layer because they do not process any information, but the neurons in the input layer have only one input and their output is always the identity. Layer $L-1$ is called the *output layer* while the rest are called hidden layers. The definition of hidden layer varies per literature, but in our context, we call

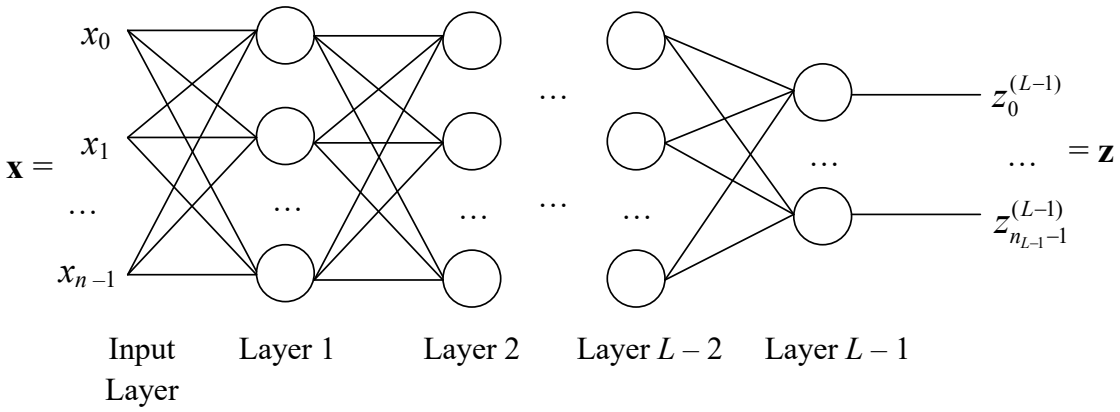


Figure 5.5: Generic Fully connected Feedforward Neural Network also known as Multilayer Perceptron.

hidden layers to those layers whose neuron inputs or outputs are not visible or accessible externally; this is, only other neurons in the network have access to the inputs and outputs of these layers.

The number of neurons for layer $l \in \{0, 1, ..., L-1\}$ is denoted n_l. The output for neuron $i \in \{0, 1, ..., n_l - 1\}$ in layer l is the neuron activation and it is denoted $z_i^{(l)} = f_i^{(l)}(S_i^{(l)})$, where $f_i^{(l)}(x)$ is the activation for the neuron and $S_i^{(l)}$ is the value of the neuron's weighted input function $S_i^{(l)}(\mathbf{x})$. The layer superscript can be omitted to signify that the value of the function evaluation for the neuron or element can belong to any layer in the network. Cardinality of the input vector is $||\mathbf{x}|| = n$. Entry $j \in \{0, 1, ..., n-1\}$ in the input vector is denoted as x_j. If the output of neuron i is input to neuron j, it is said that neuron j is connected to neuron i and it is denoted $j \leftarrow i$ and the weight of the connection is w_{ji}. The reason for the backwards indices is to help with matrix representation during computations later.

In the diagram in Figure 5.5, information propagates from left to right (from the retina towards the output layer). *Forward propagation* is the term used to describe the computation of a neural network output given an input. From the input layer, each neuron computes its activation as described earlier. The result is propagated to the first hidden layer as input. The first hidden layer performs the same propagation step and so on until the results are propagated to the output layer. Therefore, the input to every layer as well as its output are vectors.

5.1.3 Training and Back Propagation

Machine learning is a branch of artificial intelligence that covers a range of methods that allow the training of a machine to be able to approximate some *target function*. The process of *training* involves exposing the learning model to a set of samples and then tweaking the model parameters based on

the output in order to identify patterns in the input, thus learning the appropriate parameters that generalize to the complete known or unknown target function. Neural networks are a classic example of a machine learning model.

Machine learning can be split into two categories: supervised learning and unsupervised learning. *Supervised learning* is the process of "teaching" the machine by supplying the proper known output to each specific input and adapting the learning parameters to approximate the known target. *Unsupervised learning* is the process of running the machine over a provided set of inputs with no known or given output, having the data progressively cluster into different classes [59]. We focus our discussion on supervised learning in which back propagation with gradient descent excels.

Back Propagation with Gradient Descent

The modern algorithm of choice for training a neural network is *back propagation* with gradient descent. Most of today's state-of-the-art neural networks use this algorithm or some variant or extension with modifications to suit the network specifics. While there are other algorithms based on Newton's method, second order derivatives, and so on, back propagation with gradient descent is often the method of choice because of its relative simplicity, ease of computational speed and space optimization and proven results.

Given a training set of exemplary data (pair of input vector and corresponding output vector for supervised learning) containing n elements, an *epoch* is defined as a complete pass over the training set; this is, each training sample has been presented to the network for learning.

Back propagation is aimed at optimizing the parameters of an objective function to minimize the error utilizing the mathematical concept of *gradient descent* or more precisely *stochastic gradient descent*. The difference between gradient descent and stochastic gradient descent is that the former performs the update of the parameters (i.e. weights) of the network after a full epoch has completed, while the later is able to update the parameters after each sample, or after a few samples (called minibatch) [59][33]. Even though initial parameter changes may seem large and erratic, stochastic gradient descent is the method of choice when implementing back propagation because it usually leads to faster convergence after more parameter changes occur along the steepest descent per epoch than classic gradient descent.

Given the objective function $E : \mathbf{W}, \mathbf{x}, \mathbf{t} \rightarrow \mathbb{R}$, where $\mathbf{W}$ is the matrix of parameters we want to train (the weights for the network), $\mathbf{x}$ is the input vector of a sample of exemplary data and $\mathbf{t}$ is the corresponding target output vector of exemplary data, the change for matrix $\mathbf{W}$ is defined by

the Jacobian:

$$\Delta \mathbf{W} = \sum_{i}^{m} \nabla_{\mathbf{W}} E(\mathbf{W}, \mathbf{x}_i, \mathbf{t}_i) \tag{5.12}$$

This is the gradient of E with respect to the matrix $\mathbf{W}$ given the vectors $\mathbf{x}$ and $\mathbf{t}$ for a minibatch. The number of samples in the minibatch used in the step of stochastic gradient descent is m. The sum is over the m gradients obtained from each minibatch sample i (the i-th sample in the current minibatch is a pair composed of the input vector $\mathbf{x}_i$ and output vector $\mathbf{t}_i$).

Then, the stochastic gradient descent is defined as:

$$\mathbf{W} \leftarrow \mathbf{W} - \frac{\eta}{m} \Delta \mathbf{W} \tag{5.13}$$

Where η is the learning rate. This is a constant that controls how fast the change of weights affects the actual matrix of weights [54] (more on the learning rate later).

Note that this is vector notation. To compute the actual change for a single parameter weight k, the scalar formula becomes:

$$\Delta w_k = \sum_{i}^{m} \frac{\partial}{\partial w_k} E(\mathbf{W}, \mathbf{x}_i, \mathbf{t}_i) \tag{5.14}$$

where w_k is an element of matrix $\mathbf{W}$.

The scalar version of the stochastic gradient descent is then:

$$w_k \leftarrow w_k - \frac{\eta}{m} \Delta w_k \tag{5.15}$$

During computations, equation 5.14 is expanded using the derivative chain rule.

When using minibatches, the changes are accumulated for each sample in the minibatch. The actual parameter is updated by the accumulated weight divided by the number of samples in the minibatch as per equation 5.14.

When $m = n$, where n is the total number of samples of exemplary data, the stochastic gradient descent devolves into the gradient descent. If $m = 1$, then it is called online training because the network weights are modified for every sample it sees.

Gradient descent can be visualized as a simulation of a ball rolling down a hill where the objective function is seen as a distance between the network output and the target function. The objective function creates a hypersurface with an absolute minimum where the network output is the same as the target function for all inputs in the domain of the target function. The objective function evaluated in the current network output is the position of the ball on the hypersurface. Gradient descent computes the steepest slope and its direction towards the minimum, also known as gradient. Updating the network parameters to a factor of the gradient steers the network output closer to the

desired function, making the "ball" move further down into the valley on the hypersurface containing the minimum.

Online learning can see a chaotic convergence of the network while full gradient descent sees a steady, but slow approximation [59]. Stochastic gradient descent offers the best of both worlds with a less chaotic convergence, but with network modification that occurs more often than once per epoch [54]. While stochastic gradient descent and online learning have shown little improvement over the other experimentally, stochastic gradient descent is preferred. The reason is that despite the changes to the network occurring less frequently than with online learning (however, frequent enough to accelerate the network learning in contrast to gradient descent) the reduced frequency in changes helps with implementation optimization: the minibatch technique can be implemented as massive matrix operations that can be incredibly optimized with vector libraries and offloaded to co-processors and devices specialized in mass floating-point operations such as GPUs. The speed gained from fast matrix operations surpasses the online learning computational speed.

Practical Implementation: the Delta Rule

The number of weights in a FNN can grow very rapidly. For example, if layer l has n_l neurons and layer $l+1$ has n_{l+1} neurons then, the number of connections from layer l to layer $l+1$ is $n_l \cdot n_{l+1}$. To compute the change for each parameter in those connections we need just as many derivatives. Computing each derivative is an expensive process because each is composed of chains of multiplications and the deeper the network the higher the number of derivatives in the chain.

The solution for a practical implementation is the *delta rule*. It turns out that the chain of multiplications for the derivatives repeats itself on every layer. The delta rule is a dynamic programming algorithm that realizes the back propagation in a neural network.

To define the delta rule, we have to realize that the objective function E is a composite function of the form $E(\mathbf{W}, \mathbf{x}, \mathbf{t}) = (J \circ F)(\mathbf{W}, \mathbf{x}, \mathbf{t}) = J\left(F(\mathbf{W}, \mathbf{x}), \mathbf{t}\right)$, where F is the function represented by the neural network (which is the composition of the activation function of all the neurons) and J is the error or *cost function*.

The general delta rule is then defined for neuron i as:

$$\delta_i = \begin{cases} \left.\dfrac{\partial E}{\partial f_i^{(L-1)}}\right|_{z_i} \left.\dfrac{\partial f_i^{(L-1)}}{\partial S_i^{(L-1)}}\right|_{S_i}, & \text{if } i \text{ is an output neuron} \\ \left.\dfrac{\partial f_i^{(l)}}{\partial S_i^{(l)}}\right|_{S_i} \sum_j \left.\dfrac{\partial S_j^{(l+1)}}{\partial f_i^{(l)}}\right|_{S_j} \delta_j, & \text{otherwise} \end{cases} . \tag{5.16}$$

If we assume that the weighted input function for all neurons in layer $l+1$ is the dot product,

then, the delta rule can be simplified to:

$$\delta_i = \begin{cases} \left.\frac{\partial E}{\partial f}\right|_{z_i} \left.\frac{\partial f}{\partial S}\right|_{S_i}, & \text{if } i \text{ is an output neuron} \\ \left.\frac{\partial f}{\partial S}\right|_{S_i} \sum_j w_{ji}\delta_j, & \text{otherwise} \end{cases} \tag{5.17}$$

where w_{ji} is the weight of the connection from neuron i to neuron j, and δ_j is the corresponding delta value computed for neuron j. This means that for all output neurons, the delta can be computed directly using the network final output vector $\mathbf{z}$. For each hidden neuron i, it is the partial derivative of its activation function with respect to the weighted input function (evaluated in the weighted input value), multiplied by the linear combination of the weight of each outgoing connection and the corresponding connected neuron's delta [59].

The delta of a neuron represents the error of all the computations depending on that neuron's activation value.

After computing all the deltas for the network we have a lookup table of all the needed previous partial derivatives and we do not need to re-compute them each time we need to calculate the change of each weight.

Finally, if the weighted input function is the dot product as defined in equation 5.1 for all neurons then, equation 5.14 for change to the weight of connection from neuron j to this neuron i can be reduced to:

$$\Delta w_{ij} = \delta_i \frac{\partial S}{\partial w_{ij}} = \delta_i z_j \tag{5.18}$$

and

$$\Delta bias_i = \delta_i \frac{\partial S}{\partial bias_i} = \delta_i \tag{5.19}$$

Recall that the output for neuron j is denoted as z_j.

The delta rule basically makes the computation of back propagation an $O(||\mathbf{W}||)$ process, i.e. a linear computation on the number of weights.

The Vanishing Gradient Problem

If we monitor the delta δ of the neurons per layer in a FNN, we will see a pattern that the closer a layer is to the input, the smaller the average delta is. The delta of a neuron is directly proportional to the gradient or change in weight Δw as per equation 5.18, so, the smaller the delta gets, the smaller the change to the weights become, leading to slower learning. This phenomenon is known as the *vanishing gradient problem* [35][56].

The source of the problem is the actual computation of the gradient using the derivative chain rule. The farther a layer is from the output, the longer the chain of multiplications become.

For example, in equation 5.40 later in the chapter, one term of the chain is:

$$t = \frac{\partial E}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial S_1^{(3)}} \frac{\partial S_1^{(3)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial S_1^{(2)}} \frac{\partial S_1^{(2)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial S_1^{(1)}} \frac{\partial S_1^{(1)}}{\partial w_2^{(1)}}$$

where $z_i^{(l)}$ is the activation of neuron i in layer l. If the activation function for all neurons is the sigmoid, their derivative is never larger than 0.25. See Figure 5.6 for a plot of the sigmoid and hyperbolic tangent derivative showing that the derivative of the sigmoid is in the range $(0, 0.25]$ and the derivative of the hyperbolic tangent is in $(0, 1]$. So,

$$\begin{aligned} t &\leq \frac{\partial E}{\partial z_1^{(3)}} \left(\frac{1}{4}\right) \frac{\partial S_1^{(3)}}{\partial z_1^{(2)}} \left(\frac{1}{4}\right) \frac{\partial S_1^{(2)}}{\partial z_1^{(1)}} \left(\frac{1}{4}\right) \frac{\partial S_1^{(1)}}{\partial w_2^{(1)}} \\ &= \left(\frac{1}{64}\right) \frac{\partial E}{\partial z_1^{(3)}} \frac{\partial S_1^{(3)}}{\partial z_1^{(2)}} \frac{\partial S_1^{(2)}}{\partial z_1^{(1)}} \frac{\partial S_1^{(1)}}{\partial w_2^{(1)}} \end{aligned}$$

As we get closer to the retina, the factor shrinks exponentially in the order of 0.25 per layer. This is an informal argument and not a rigorous proof that the vanishing gradient will occur. The message to take is that the gradient in an early layer is the product of all the gradients that follow and the vanishing gradient is a possibility given that most modern techniques to improve accuracy and convergence rate tend to keep parameters within a range close to zero to facilitate training and avoid neuron saturation.

The other side of the coin is the *exploding gradient*. This occurs when one of the repeating terms in the chain grows large making the deltas grow exponentially the farther the layers are from the output for the same reasons as to why vanishing gradient occurs. While this phenomenon is less

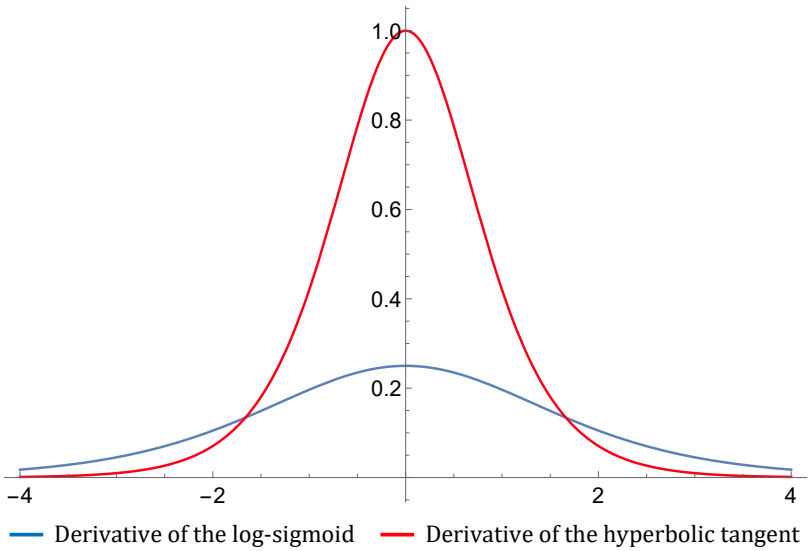


Figure 5.6: Plot of derivatives of the log-sigmoid and hyperbolic tangent functions.

common than vanishing gradient, both are the result of the same structure, therefore, the term *unstable gradient* is used to refer to the general problem.

Deep networks such as convolutional and recurrent neural networks are more susceptible to the vanishing gradient problem mainly because of their large number of layers.

5.1.4 Improving Convergence Rate

Convergence rate is defined as the number of samples required to be presented to the network before it reaches an approximation value within a desired ϵ of the target function or it stops learning. This is often measured in *epochs* since the whole collection of training samples must be presented to the network several times to reinforce learning before achieving good approximation values.

Note that *convergence speed* is a different term. It has to do with the time (in units of time such as seconds) that it takes for a specific network to converge while training on the same hardware. In computer science equivalents, convergence rate can be viewed as algorithm complexity, while convergence speed is related to execution time. However, while high complexity algorithms become impossibly large to compute despite optimization techniques, there are cases in neural network training in which some convergence algorithm may offer significantly better convergence rates, but the implementation and convergence speed are prohibitively expensive.

Most of the research in the area of neural networks today is focused on changes to the core concept, replacing formulas, activation functions, parameter selection, etc. to obtain better convergence rates and tackle the unstable gradient problem, reducing the amount of training necessary while attempting to improve on the already excellent results in accuracy. Research on convergence rate also focuses on striking a balance between techniques to improve convergence rate and feasible implementation and optimization to reduce overall convergence speed on a given hardware. These techniques are spread out throughout several researches. Some of the most commonly used and with better results are compiled here and applied to our practical implementation.

A major factor that affects convergence rate are *hyperparameters*. Hyperparameters are parameters used to control the learning of a network, but do not form part of the network itself. Once the network has been trained, hyperparameters are discarded.

Learning Rate

One problem of gradient descent is that the approach to the minimum error is asymptotic. This is, the closer the network output gets to the target function, the smaller the changes become and the approximation slows down due to the nature of the objective function.

During the initial stages, changes are often big and happen fast because the network output is far from the target function. The learning rate η is the hyperparameter that controls how fast the changes occur (see equation 5.13). Learning rate is a real number typically in the range $(0, 1)$, but, in some cases, it can be above 1 or it must be below 0.1. Deeper architectures favor small learning rates. Normally, constant factors that appear during the derivation of the equations for back propagation are omitted and condensed within the learning rate value.

A relatively large learning rate will make the network start converging fast; however, once the error gets closer to the minimum, a large learning rate applies large changes to the parameters when the asymptotic nature of the method suggests that changes have to be smaller. This may cause the network output to skip over the actual minimum and sometimes produce larger errors after the skipping. In such case, oscillation of the approximation is the best case scenario; gradient explosion, leading to divergence, is a worse case possibility.

A common technique to avoid this problem is to dynamically reduce the learning rate of the network. While choosing the initial learning rate for the network or for each individual neuron can be difficult and it is often a result of trial and error, there are a couple of standard methods to reduce the rate dynamically that work well in practice.

One method increases the learning rate by a very small amount every epoch until the convergence slows down between epochs. When such is the case, the learning rate is drastically reduced, usually by half its value. This method gives good convergence to local minima, but needs some adaptation when working with minibatches [56].

Another method is called annealing in which the learning rate at each iteration t is set to $\frac{a}{c(b+t)}$ where a is the initial learning rate, b is the learning rate when the annealing starts (i.e. the current learning rate) and c is some pre-selected constant [54].

During our practical implementation we did not implement dynamic learning rate directly into the libraries, but let the clients decide on their own implementation. For our experiments, we utilized a hybrid of the two methods: we performed a variation of annealing based on the number of reductions instead of number of iterations as shown by algorithm listed in algorithm 2. The reduction was performed whenever the convergence slowed down and the result of the error function began to increase. The reduction of the learning rate often resulted on continued convergence of the network until the eventual convergence or stabilization.

While reducing the learning rate gives good results regarding convergence speed, the problem of vanishing gradient is ever present in deep networks (more than one hidden layer). The closer

Algorithm 2 Schedule Algorithm for Dynamic Learning Rate Reduction by Annealing on Number of Reductions

Let $a \leftarrow$ initial learning rate.
Let $\eta \leftarrow a$; current learning rate.
Let $d_0 \leftarrow +\infty$; the last error achieved.
Let $D \leftarrow$ desired accuracy.
Let $\epsilon \leftarrow$ small positive value indicating little progress.
Let $threshold \leftarrow$ maximum number of epochs allowed to train with an error difference bellow ϵ.
Let $bad\_epochs \leftarrow 0$; number of epochs that have trained with an error difference bellow ϵ.
Let $t_{\max} \leftarrow$ maximum number of learning rate reductions before stopping training (early stopping).
Let $t \leftarrow 0$; number of learning rate reductions.
repeat
    Train for 1 epoch using current learning rate.
    Let $d \leftarrow$ error achieved between the network output and target function after last training.
    if $(d_0 - d \leq \epsilon)$ **then**
        $bad\_epochs \leftarrow bad\_epochs + 1$
    else
        $d_0 \leftarrow d$
        $bad\_epochs \leftarrow 0$
        $t \leftarrow 0$
    end if
    if $(d_0 - d < 0)$ **or** $(bad\_epochs \geq threshold)$ **then**
        $\eta \leftarrow \dfrac{a}{2(\eta + t)}$
        $t \leftarrow t + 1$
        $bad\_epochs \leftarrow 0$
    end if
until $(d_0 \leq D)$ **or** $(t > t_{\max})$

we get to the input layer, the smaller the deltas for the neurons become. Changes to the error function can mitigate the impact of the vanishing gradient problem, however, we found during our implementation and testing that having individual learning rates for the layers (and even individual neurons) and making the reduction of the learning rate less aggressive, proportional to the distance from the neuron to the output layer would also result in a small improvement on accuracy.

The factor of the reduction was decreased the further away a neuron was from the output when a learning rate reduction was required. However, the selection of the factors should be studied more closely. The factors used in our experiments were selected as the result of trial and error. This implementation consideration made changes to in the initial layers more relevant.

The learning rate can be constant over an entire epoch or there can be algorithmic variations in which the learning rate changes during the course of an epoch based on some condition.

Adding Momentum

If the objective function around the local minimum to which our network function is converging has the form of a shallow ravine with steep walls around it, large learning rates will cause oscillations in the best case; small learning rates can bring the approximation to a crawl or even to a stop the closer we get to the minimum.

We encountered this issue during our implementation as well. Researching the topic, we found that a method used to push the convergence towards the minimum faster is the use of "momentum."

Momentum is an analogy from physics in which a charged particle continues moving in the direction of its velocity even when the forces applied to it change magnitude and direction in an electric field, for example.

Adding the momentum to the change of weight in equation 5.18 results in what is known as the "generalized delta rule" [61]:

$$\Delta w_{ij} \leftarrow \delta_i z_j + \mu \Delta w_{ij} \tag{5.20}$$

where $\mu \in (0, 1]$ determines how many iterations of the previous gradients are incorporated into the current update. Generally $\mu \leq 0.2$ during the initial learning and it is increased to $\mu = 0.9$ or higher when the convergence slows down.

Because $\mathbf{\Delta W}$ is a factor of the learning rate as defined in equation 5.20, momentum has been found to help reduce opposing oscillations by counteracting weight changes in opposite directions while enhancing the effect of the learning rate when there are several weight changes following the same direction [59]. In the literature, the learning rate is a factor of the second term in equation 5.20 and the momentum is defined as an independent constant, or as a value depending on the learning rate. The momentum is, in the end, constant over an entire epoch.

Experimenting with our implementation, the addition of momentum dramatically reduced the convergence rate by shaving off a fifth of epochs needed to train the network to reach the desired error. We increased the momentum after we had applied more than two learning rate reductions at which point the convergence of the network had slowed down to a crawl and no oscillation was occurring.

Cost Functions

The cost function is the distance measure between a single network output and the target output of the known target function being approximated. The cost function composed with the function represented by the network is the objective function. The training process of a neural network entails the minimization of the objective function by adjusting the parameters or weights of the

neural network.

The choice of cost function is related to the activation function selected for the neurons in the output layer as well as to try to mitigate the effect of the vanishing gradient by limiting the reduction of the error value with less multiplications or canceling terms in the chain rule or by producing large error values.

Most common cost functions found in literature and state-of-the-art networks today are the Mean Squares Error (MSE), the Cross-Entropy and closely related log-likelihood function.

The MSE cost function is classically defined as:

$$J(\mathbf{z}, \mathbf{t}) = MSE = \frac{1}{n} \sum_{i=1}^{n} (z_i - t_i)^2 \tag{5.21}$$

where, given the next sample of exemplary data $(\mathbf{x}, \mathbf{t})$ –input vector $\mathbf{x}$ and known target output $\mathbf{t}$–, $\mathbf{z} = F(\mathbf{W}, \mathbf{x})$ is the output vector obtained from the neural network with the current weight matrix $\mathbf{W}$. The number of neurons in the output layer is the cardinality of $||\mathbf{z}|| = ||\mathbf{t}|| = n$, so, z_i and t_i are the i-th elements of vectors $\mathbf{z}$ and $\mathbf{t}$ respectively, $i \in \{0, 1, ..., n-1\}$.

Also,

$$\frac{\partial}{\partial z_i} J(\mathbf{z}, \mathbf{t}) = 2(z_i - t_i) \tag{5.22}$$

Note that in derivatives, the coefficients are usually omitted because they are factored into the learning rate. So, equation 5.22 is actually used as:

$$\frac{\partial}{\partial z_i} J(\mathbf{z}, \mathbf{t}) = z_i - t_i \tag{5.23}$$

The MSE is an umbrella cost function that can be used in most cases in contrast with other cost functions that are specially defined to be used only for certain problems and specific activation functions.

The Cross-Entropy is defined as:

$$J(\mathbf{z}, \mathbf{t}) = \ln \left(\prod_{i=1}^{n} z_i^{t_i} (1 - z_i)^{(1-t_i)} \right)^{-\frac{1}{n}} \tag{5.24}$$

But this definition is computationally expensive. It is most commonly expanded and used as:

$$J(\mathbf{z}, \mathbf{t}) = -\frac{1}{m} \sum_{\mathbf{x}} \sum_{i=1}^{n} (t_i \ln z_i + (1 - t_i) ln(1 - z_i)) \tag{5.25}$$

where m is the number of exemplary inputs $\mathbf{x}$ in a minibatch.

Also,

$$\frac{\partial}{\partial z_i} J(\mathbf{z}, \mathbf{t}) = \frac{z_i - t_i}{z_i(1 - z_i)} \tag{5.26}$$

This cost function is mostly used with activation functions that output results in the range $(0, 1)$ such as the sigmoid or softmax. This function mitigates the vanishing gradient problem because the partial derivatives depend linearly on the output z instead of a quadratic dependence as is the case with MSE [56].

Note that if the activation for output neuron i is the sigmoid as defined in equation 5.2 then:

$$\begin{aligned} z_i &= f(S) = \frac{1}{1+e^{-S}} \\ z_i' &= \frac{\partial}{\partial S} f(S) \cdot S' \\ &= \frac{e^{-S}}{(1+e^{-S})^2} \cdot S' \\ &= z_i(1-z_i) \cdot S' \end{aligned}$$

and by the chain rule, one of the terms of the sum is

$$\begin{aligned} \left.\frac{\partial E}{\partial f}\right|_{z_i} \left.\frac{\partial f}{\partial S}\right|_{S_i} \frac{\partial S}{\partial w_k} &= \frac{z_i - t_i}{z_i(1-z_i)} z_i(1-z_i) \frac{\partial S}{\partial w_k} \\ &= z_i(1-z_i) \frac{\partial S}{\partial w_k} \end{aligned}$$

if E is the cross-entropy cost function.

As we can see, depending on the activation or weighted input functions for the neurons, the choice of cost function can simplify the computations or simplify terms, helping reduce the impact of the vanishing gradient problem.

The log-likelihood function is closely related to cross-entropy with a similar effect on the gradient because while the objective is to maximize the likelihood between the exemplary output and the actual output, it reduces to minimize the quadratic term of the MSE.

Regularization

On the same note, if we apply all previous little details to training a network, we may find that some weights can grow rather large. In such cases, neurons can become saturated by only a few of these since any time the neuron connected with this weight fires, it overcomes all other neurons connected with smaller connection weights, also contributing to the exploding gradient problem. This behavior leads to large weighted input values and learning slowdown, and, sometimes, overfitting.

To maintain weights inside smaller ranges, regularization techniques have been proposed. The purpose of *regularization* is to keep weights controlled in small ranges around the vicinity of zero. This improves learning because small changes on the weights still have impact on the computation of activation and reduces the chance of occurrence of exploding gradients.

A commonly used regularization technique is called L2 regularization. It consists on adding a term to the cost function. If the current cost function is J_0, then, the new cost function is [56]:

$$J = J_0 + \frac{\lambda}{2n} \sum_i w_i^2 \tag{5.27}$$

where n is the size of the training set and $\lambda > 0$ is the *regularization parameter*. The sum is over all the weights in the network.

The effect of this change is to have the network prefer small weights over large weights. Large weights are only preferred if they considerably improve the original cost function J_0. This preference is controlled by the regularization parameter which becomes another hyperparameter for the training of a network. A relatively large λ inclines the network towards smaller weights while a smaller one makes the network prefer weights that impact the original cost.

With this change, equation 5.15 for the update of the weights becomes:

$$w_i \leftarrow \left(1 - \frac{\eta\lambda}{n}\right) w_i - \eta \Delta w_i \tag{5.28}$$

If the regularization parameter is zero, equation 5.28 devolves into equation 5.15 with no changes. The factor $\left(1 - \frac{\eta\lambda}{n}\right)$ is referred to as weight decay. Equation 5.28 is obtained by the chain rule method applied as usual where

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial J}{\partial w_i} \\
&= \frac{\partial}{\partial w_i}\left(J_0 + \frac{\lambda}{2n}\sum_i w_i^2\right) \\
&= \Delta w_i + \frac{\lambda}{2n} \cdot \frac{\partial}{\partial w_i} \sum_i w_i^2 \\
&= \frac{\lambda}{n} w_i + \Delta w_i \\
w_i \leftarrow & w_i - \eta \frac{\partial E}{\partial w_i} \\
&= w_i - \eta\left(\frac{\lambda}{n} w_i + \Delta w_i\right) \\
&= w_i - \frac{\eta\lambda}{n} w_i - \eta \Delta w_i \\
&= \left(1 - \frac{\eta\lambda}{n}\right) w_i - \eta \Delta w_i
\end{aligned}$$

Since the regularization term added to the cost function does not affect the biases, equations to compute the change in bias are not affected.

Dropout

Dropout is a regularization technique different from L1 and L2 regularization. It consists on modifying the network by temporarily randomly eliminating about half of the the network's hidden neurons, then training the remaining half using minibatch gradient descent. After a minibatch training is completed, the removed neurons are restored and the process repeated from the elimination step.

Dropout has been studied since originally introduced because of its effect of considerably improving accuracy, especially within very deep networks such as convolutional neural networks where overfitting is a real problem [27].

The idea about this method leans on the fact that if we train several networks to recognize the same function, we can find a consensus about the result of a single evaluation, thus increasing the probability of a successful recognition. For example, if three out of five networks trained with different initial parameters, and possibly different architecture, recognize the input to be in class A while the others recognize it as class B, then the input most likely belongs to class A.

The technique reduces dependency among neurons and strengthens the level of accuracy by verification of the results via different paths. The reduced number of neurons is forced to learn more robust and general features that are corroborated by the other sub-networks [36]. The resulting network can be viewed as composed by several sub-networks trained more or less individually which come together to achieve a consensus for the solution.

While this is a concise overview of the concept of dropout, the original paper and application of the method by Geoffrey Hinton *et al.* [27][36] go into specific details that are useful for implementation and utilization of the technique.

Batch Normalization

Batch normalization is a novel technique that aims to reduce the need for regularization in deep architectures [31]. This technique is geared directly towards stochastic gradient descent because it works in a per-batch mode.

Even if the input data to a layer is scaled to be in the range $[-1, 1]$, it may still suffer from a shift from a mean value of 0 and a standard deviation of 1. Also, as the data flows from layer to layer, the activation values of each layer may even deviate outside of this range. Recall that sigmoid-like activation functions tend to saturate outside of this range, so, as the data propagates through the network it is preferred to maintain it within $[-1, 1]$ and with a distribution of mean 0 and standard deviation of 1.

This is exactly what batch normalization accomplishes. It shifts and scales the input data to

each layer so that it maintains a distribution that, most likely, will not saturate neurons or cause exploding gradients. Without batch normalization, the shift of the data outside of the desired range is referred to as "internal covariate shift" and every layer is prompt to produce it.

Batch normalization works by adding a special *Batch Normalizing layer* (BN layer) between existing layers of a network. The BN layer will apply a *Batch Normalizing Transform* to the incoming batch B of m inputs (each input is a vector $\mathbf{x}$ of cardinality n) as follows:

$$\mu_{B_i} \leftarrow \frac{1}{m}\sum_{j=1}^{m} x_{ij} \tag{5.29}$$

where $i \in \{1, 2, ..., n\}$ is the dimension of vector $\mathbf{x}$ for which to compute the average μ_{B_i}. So, each entry in vector $\boldsymbol{\mu_B}$ is the average of the corresponding entry of all the input vectors in batch B.

Similarly, the minibatch variance is computed as the unbiased estimation:

$$\sigma^2_{B_i} \leftarrow \frac{1}{m}\sum_{j=1}^{m}(x_{ij} - \mu_{B_i})^2 \tag{5.30}$$

This produces vector $\boldsymbol{\sigma_B}$ where each of its entries is the unbiased estimation of the variance for each corresponding entry of all the input vectors in batch B.

The input is normalized as:

$$\hat{x}_{ij} \leftarrow \frac{x_{ij} - \mu_{B_i}}{\sqrt{\sigma^2_{B_I} + \epsilon}} \tag{5.31}$$

where ϵ "is a constant added to the minibatch variance for numerical stability" [31].

Finally, the activation of for each neuron in the BN layer is computed as a scaling and shifting linear operation:

$$z_{ij} \leftarrow w_i \hat{x}_{ij} + b_i \tag{5.32}$$

where w_i and b_i are the trainable parameters for this layer and can be seen as the weight for the single input to neuron i and its bias.

The scaling and shifting is required because during normalization, we lose information about the original distribution of the data. The network can re-learn this distribution or a completely different distribution through the trainable parameters w_i and b_i from equation 5.32 if needed.

The BN Transform is a differentiable process, thus ensuring that the equations for gradient descent can be applied to it without any problem and training can proceed with this type of layer as with any other layer.

Batch Normalization has been shown to enable the application of larger learning rates in deep networks, speeding up the convergence rate, and the utilization of sigmoidal functions since the input range is kept within limits that avoid saturation. It has also been proven to reduce the need of regularization techniques and dropout and it is a staple component of most recent neural network architectures.

Weight Initialization

There are several ways to initialize the weights of a neural network before training. The strategy chosen can be seen as another hyperparameter for training.

There is empirical evidence that good weight initialization can lead to improved recognition accuracy and convergence rate. Given the nature of most activation functions, weights should be chosen in the range $[-1, 1]$ and as close to zero as possible, but not zero for most values. While complicated methods utilizing genetic algorithms do a decent job of finding good initialization values, the overhead and extra work is significant when compared to the results obtained from random initialization.

A good initialization pushes the objective function to the vicinity of the minimum that is being found during training as well as watches to avoid neuron saturation as explained in section 2.3.1.

By the same reason, the elements of the input vector to the neural network should be scaled to be in the range $[-1, 1]$ or $[0, 1]$ depending on the architecture and activation functions selected for the network.

Organizing Samples and Testing

Neural networks have been shown to possess the universal approximation property given certain constraints. This is good news since it means that with enough trainable parameters, a neural network can approximate any function or fit any data set. This, however, is a double-edged sword. Neural networks are also used as classifiers due to their approximation capabilities, but mostly because under certain circumstances, neural networks perform excellent generalization.

A classifier is a mathematical function that maps input data to a category based on specific features. A classifier can be directly formulated or trained. The desirable trait of any classifier is that it will correctly classify any possible input data correctly. However, when there is no known classification function for a problem, good trainable classifiers allow for an approximation to such a function and generalize to all unknown, but possible input based on the training set.

Normally, a model with a large number of parameters, relative the number of data points leads to overfitting. *Overfitting* refers to a a model describing a phenomenon "too well." The model starts to describe noise and error in the observations instead of the underlying relationship. In this case, the model fits perfectly to the training set, but fails to perform beyond that.

The large number of trainable parameters in a neural network are key to its great strength and flexibility. The parameters are changed and steered through the course of training to fit the training set and to be able to generalize to elements not included in the training set. However, excessive

training or bad choice of training and testing sets may lead the network to overfit, akin to how a polynomial of order 5 can fit perfectly on 4 data points even if their real relationship is linear. In this case, the network stopped learning the general features of the training set and began memorizing particular noise on the training set or overfitting over the testing set.

A technique to avoid overfitting is called *hold out* [56]. In its basic form, it consists on splitting all the data into three sets: *training set*, *validation set*, and *testing set*.

The training set contains all samples that will be used actively during training of the network. Presenting all the samples in this set to the network constitutes an epoch. The order of presentation of the training set samples to the network has been the subject of research. It has been found that a uniform, random order of presentation, especially if there is a uniform distribution of samples per class in each minibatch, allows a network to learn better by not skewing towards certain classes that may have been clustered at the start of the epoch.

The validation set shall be used with two purposes. First and foremost, it will be used to assess the progress of the network after each epoch. Second, it will be used to select hyperparameters for training.

Several combinations of hyperparameters, such as learning rate, dynamic learning rate algorithm, regularization parameter, momentum, weight initialization strategy, and even architecture, are usually tested before actual training starts in order to select the best candidates to achieve desired recognition rates with reasonable convergence rates. The success of each combination should be validated against the validation set.

The test set is withheld from the network and only used to test whether the network is generalizing or overfitting against the validations and training sets. A network may be showing use what we want to see when testing against validation set, but it may be doing poor generalization. We can use the testing set to ensure that a proper training was completed and the resulting network generalizes to other samples outside of its training and validation sets.

Last, but not least, a common source of overfitting is training too much. If the network reaches a stable state, where its recognition accuracy does not improve it could be because it reached a point on the error surface with little slope or it has approached to the minimum asymptotically and it can no longer refine its approximation with its current architecture and parameters.

The first case, although not common, it can still occur. During training, one can chose to save the current state of the network and allow it to train for longer in hopes that it may leave the flat area and continue learning, but with the fallback state in case it won't happen.

The second case is the more common as a network will approach the minimum asymptotically and may bounce around it indefinitely given its current state. Depending on how aggressive the training is, this oscillating stable state can be detected by the training software and stop training. This is called *early stopping*.

Continuous training when a network has reached a stable state can lead to overfit on the training set and start learning its noise. When this occurs, a drop in recognition of the validation set begins to appear, unless the overfitting is also being observed over the validation set. If such is the case, the testing set can be used to check for overfitting. Early stopping can successfully avoid overfitting by over training, and not only avoid the undesired effects, but it can shave precious time off the training.

Too aggressive early stopping detection may result in underfitting, the opposite of overfitting and it is even less a desirable effect.

A technique for early stopping can be implemented by counting the number of learning rate reductions from the dynamic learning rate algorithm and stop training after a certain number of reductions occurs without improvement of accuracy within some ϵ. This was our preferred approach during our experiments and it is depicted in algorithm 2 where variable t keeps track of the number of continuous reductions without improvement.

5.1.5 Implementation Remarks of Feedforward Neural Networks

Our current implementation of a neural network library is capable of training fully connected FNNs faster than comparable Python implementations that utilize Numpy or Theano libraries or Google's Tensorflow machine learning library and as fast as Intel's deep learning library as of this writing.

Our implementation uses C++ 11 with OpenMP for multicore computing, along with Intel Math Kernel Library (MKL) to optimize computation speed with mass vector and matrix operations and with the capacity to auto offload work to coprocessors. For trully large matrix operations, our library utilizes OpenCL kernels that offload massive matrices into GPUs if available when the time it takes for the matrix operations to complete in the main processor of the machine outweights the time it takes for data transfer from main memory through the bus into the GPU.

While popular machine learning libraries take advantage of similar optimizations, the direct C++ implementation delivers better performance due to its proximity to hardware. C++ also requires no bridge to communicate with other libraries or hardware directly, feature that Python lacks. However, as these popular libraries improve, the defficiencies from Python become less of an impact.

Our implementation was compiled and deployed to the Cheery-Creek super computer from the National Supercomputing Institute, helping to develop experiments that completed in hours in-

stead of days, taking advantage of Intel Xeon processors and Xeon Phi coprocessors for operation offloading.

Optimizations

Our implementation takes advantage on the fact that forward propagation between fully connected layers is achieved by the following vector-matrix operation:

$$\mathbf{S}^{(l)} = \mathbf{W}^{(l)} \cdot \mathbf{z}^{(l-1)} + \mathbf{b}^{(l)} \tag{5.33}$$

If layer l has n_l neurons and layer $l-1$ has n_{l-1} neurons, then, $\mathbf{b}^{(l)}$ is a vector that can be seen as a $n_l \times 1$ matrix of values representing all the corresponding biases in layer l, $\mathbf{W}^{(l)}$ is a $n_l \times n_{l-1}$ matrix where $w_{ji} \in \mathbf{W}^{(l)}$ is the weight of the connection from neuron j in layer $l1$ to neuron i in layer l; $\mathbf{z}^{(l-1)}$ is a vector that can be seen as a $n_{l-1} \times 1$ matrix of values representing the corresponding activations of all the neurons in layer $l-1$. Finally, $\mathbf{S}^{(l)}$ is a $n_l \times 1$ matrix of values representing all the corresponding weighted input values for the neurons in layer l.

Furthermore, if we train using minibatches of size m, we can condense all minibatch samples into matrix form:

$$\mathbf{S}^{(l)} = \mathbf{W}^{(l)} \cdot \mathbf{Z}^{(l-1)} + \mathbf{B}^{(l)} \tag{5.34}$$

where $\mathbf{Z}^{(l-1)}$ is the equivalent to vector $\mathbf{z}^{(l-1)}$, a $n_{l-1} \times m$ matrix where each column represents the activation vector resulting from the corresponding sample in the minibatch; $\mathbf{B}^{(l)}$ is the equivalent to vector $\mathbf{b}^{(l)}$, a $n_l \times m$ matrix where each column would be a replicated vector $\mathbf{b}^{(l)}$. The resulting $\mathbf{S}^{(l)}$ would be then a $n_l \times m$ matrix of all the weighted input values for the minibatch.

This is the reason minibatch training of stochastic gradient descent is the preferred method. Matrix operations are much faster than vector-matrix operations (up to a point) because the processor can cache large amounts of information limiting memory access and bus traffic. Furthermore, for very large matrices, operation offloading offers even more computation speed because dedicated devices such as GPUs can perform massively parallel operations and since all the data is transferred once every batch instead of every sample, the operation speed gain surpasses the bus load and transfer times penalty. The thin grain parallelization allowed by neural network computational units is a great fit for the greatly parallelizable matrix operations.

The delta rule for backpropagation can also be optimized following a similar philosophy. If $\boldsymbol{\delta}^{(l)}$ is the vector the corresponding neuron deltas in layer l, then:

$$\boldsymbol{\delta}^{(l-1)} = \left.\frac{\partial f}{\partial S}\right|_{\mathbf{S}^{(l-1)}} \cdot \left(\mathbf{W}^{(l)}\right)^{\intercal} \cdot \boldsymbol{\delta}^{(l)} \tag{5.35}$$

where $\left(\mathbf{W}^{(l)}\right)^{\mathsf{T}}$ is the transpose of matrix $\mathbf{W}^{(l)}$, and $\left.\frac{\partial f}{\partial S}\right|_{\mathbf{S}^{(l-1)}}$ is the vector of the derivative of the activation function evaluated in layer $l-1$'s weighted input values.

A similar matrix extension for minibatch processing is easily implemented for this formula.

Bundling operations together in vector and matrix form makes vector libraries such as MKL work quite efficiently, reducing the training time dramatically. Using these methods, the C++ implementation reduced training times by a factor of 20 in the supercomputer and it was still reasonable to train using a commercial workstation, especially with an equiped GPU.

With these improvements in speed, we were able to reproduce and surpass convergence speed of the examples given in Nielsen's book [56], even stress-training for many epochs and adding several more neurons and hidden layers.

5.2 Convolutional Neural Networks

Filters are major tools used for digital signal and image processing. A *Convolutional Neural Network* (CNN) is a feedforward neural network with an architecture that allows it to incorporate the concept of filters to fully connected designs. CNNs were biologically inspired by the animal visual cortex where individual nerve cells are stimulated by specific events or patterns from a constrained portion of the whole receptive field. Each of these constrained neurons is basically a filter that fires if the pattern or event being filtered is present.

Through training, filter neurons learn the parameters for a pattern. During forward propagation, the filters extract the relevant pattern information and features, passing it to the fully connected portion of the network which is in charge of analyzing and recognizing the patterns extracted. This type of network is relatively new and has proven very powerful in its intended area since its practical inception in seminal works such as in LeNet-5 [42].

The concept of a filter involves a kernel that operates on a signal by applying it to every element of the input through a process called *convolution*. Convolution is an integral that represents the amount of overlap of one function g as it is shifted over another function f [74]. The resulting signal will have the kernel features enhanced while others would be attenuated.

The concept of a filter is realized in a CNN as a neuron with sparse connections to the previous layer. Ideally, a single filter would be convolved over a whole input signal to produce an output, and in the case of a filter neuron, the neuron would fire if the feature detected by its filter is found over the whole input. However, a single neuron acting as a filter is not optimal in the context of a neural network, therefore, for a single filter, there is a neuron per input element in a layer where each neuron of pertaining to some filter shares weights with the other neurons of the same filter. A

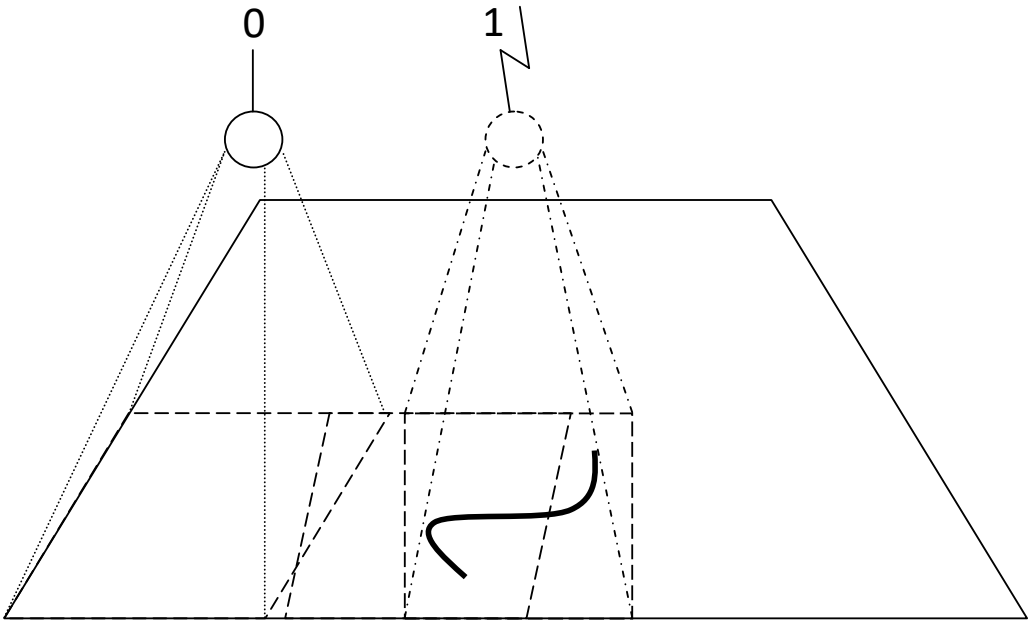


Figure 5.7: Filter neurons in a convolutional layer.

Picture the filter neuron looking at a portion of the input image, then moving to the next portion until the complete image has been looked at. Whenever the neuron "sees" the pattern that it is trained for, it will fire. A convolutional layer actually has a filter neuron per portion sharing the filter instead of moving a single neuron over the whole input.

layer composed of filter neurons is called a *convolutional layer*. The output of a filter convolution is called the feature map containing the activation values of all the neurons sharing the same filter. The filter along with the step at which it is applied to the input is called a feature. Figure 5.7 graphically depicts the operation of a feature in a convolutional layer.

Because of the nature of the convolution operation, features are said to be translation invariant. This is, the localization of the feature in the input is not consequential, but its existence is. Features are not, however, scale invariant. In their paper "Learning Hierarchical Features for Scene Labeling" [18], Clement Farabet *et al.* apply CNN to label objects in an image. They trained their network with samples of a single scale. To account for the lack of scale invariance during testing, they created what is known as a image pyramid by manually scaling the original image to different sizes before applying a CNN to each size in order to identify features when they appeared at the trained scale.

Often, convolutional layers are followed by so called pooling layers. Pooling layers contain neurons that take the output of the preceding convolutional layer and subsample it by finding the maximum output value or the average value of regions from the convolutional layer output. The objective of pooling layers is to abstract from locality where a feature may have been found. The loss of locality is a necessary drawback of CNNs.

Pairs of convolutional-pooling layers form the initial layers of a CNN. Fully connected layers after the convolutional-pooling layers perform the actual pattern recognition based on the features extracted. The number of fully connected layers is usually 2 counting the output layer, but rarely more than 3. This is based on the fact that any fully connected FNN with non-linear units possesses the universal approximation property when it has a hidden layer with a sufficient number of neurons [29][45].

Because a convolutional-pooling layer pair outputs a feature map, this can be used as input to another convolutional layer to extract other features from the resulting map. This chain of feature maps allows for the addition to several connected convolutional layers. CNNs can have from a few layers of depth like LeNet-5 with 5 layers to 20 or more layers like Google's Inception. Recent news have revealed extravagant amounts of combination of layer types and techniques from Microsoft research that totals to over 150 layers of depth to win the 2015 ImageNet Large Scale Visual Recognition Challenge, overcoming the vanishing gradient and difficulty of training very deep networks with a new method dubbed "Deep Residual Learning" [25]. These new architectures are the reason of the appearance of buzz words to describe the learning mechanisms of these large networks such as "Deep Learning" since their counterpart vanilla multilayer perceptrons rarely boasted more than 2 hidden layers.

On initialization, before training, the actual values for the filters in the convolutional layers are not known and are initialized using similar techniques as for weights in fully connected layers. The goal is to utilize learning techniques like gradient descent to train the whole network to automatically converge to the appropriate filters that will recognize the features specific to the problem.

The great strength of CNNs with respect to classic signal processing methods is that convolutional filter kernels need not (and should not) be known before training, but are learned by the network using the exact same back propagation algorithm as with fully connected networks resulting on accurate filters that extract the correct features needed for recognition, while classical methods utilize pre-defined filters that may not be as appropriate for every situation.

5.2.1 Convolutional Architecture

In classical statistical pattern recognition, several (orthogonal) features are often used to enhance the accuracy of classification. Clearly, the more identifying features an input has, the better it can be classified as being part of a specific class of inputs sharing the same features. The same principle applies to convolutional layers. A single convolutional layer should scan its input for more than one feature, so, there can be s sets of neurons, each set sharing its own filter among its neurons. Refer to Figure 5.7 showing a diagram representing a convolutional layer and picture, instead of a

single filter neuron, s filter neurons looking at the same portion (however, due to the nature of the features, portions in which an input is divided need not be the same for each feature).

Despite the underlying implementation of a convolutional layer, a convolutional layer is interpreted as a 3-dimensional layer: width, height and depth where depth is the number of features while width and height directly correspond to the dimensions of the input signal.

Assume we have a training set of 50000 samples of input images with dimensions 28×28 and they are full RGB (3 color channels) and the target classification into 10 classes for each input. The input dimensions are $28 \times 28 \times 3$. Let's create a simple CNN architecture to classify the images. The architecture will have a single 15-features-convolutional layer connected to the input, followed by a max-pooling layer, ending with a 10 output neurons fully connected layer. Note that this is just an illustrative example of a CNN and may not be an optimal CNN recognizer for this classification problem.

We will define all the features in the convolutional layer to be $5 \times 5 \times 3$ filters operating on all the color channels with a step of 1 pixel vertical and 1 step horizontal (note that we can have filters of any size with any step that operate on a specific input channel or on several input channels). This will make the output for a feature to be of size $24 \times 24 \times 1$. Note that the output of a feature is always of depth 1, so, the depth of an n-features convolutional layer is $output\_width \times output\_height \times n$. Therefore, the output for the convolutional layer will be of size $24 \times 24 \times 15$.

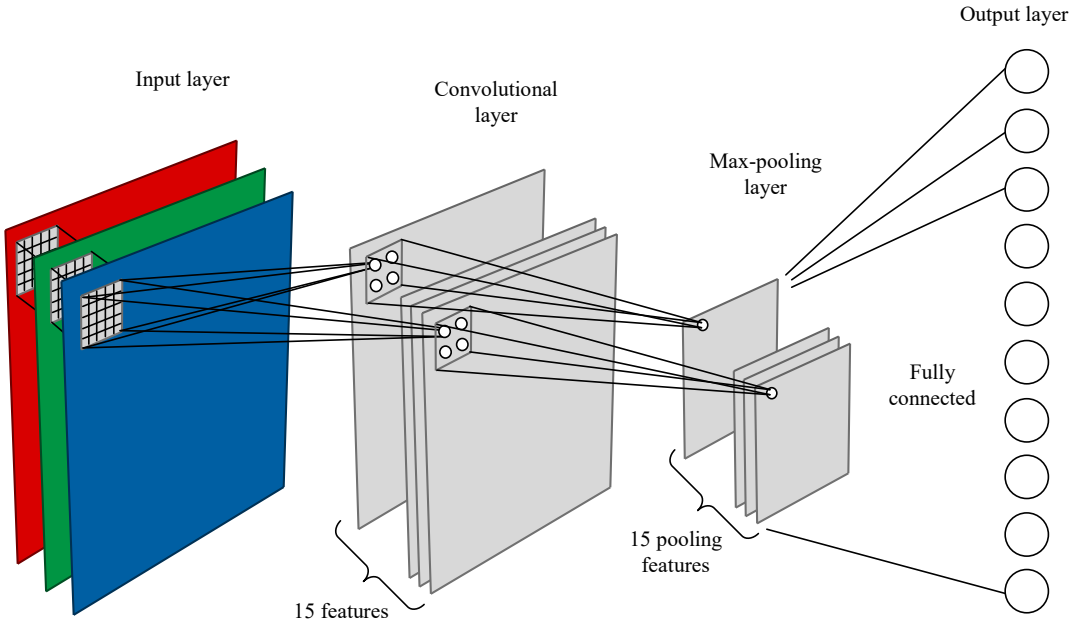


Figure 5.8: Example of convolutional neural network.

Max-pooling layers are special cases of convolutional layers where the weighted input function is not the dot product, but the maximum function. The pooling layer will perform max-pooling in an area of 2×2 with a step of 2 for each feature. The output of the max-pooling layer will be of size $12 \times 12 \times 15$. While the step in a convolutional layer feature is designed to have the filtered regions overlap, it is desired in pooling layers that pooled regions do not overlap, but they are contiguous in the layer input.

See Figure 5.8 for a graphical depiction of the described network. Note that this depiction is shows the functional architecture. The actual architecture realizes the function by connecting the sparse connections from convolutional neurons to the appropriate outputs from the previous layer.

All the neurons in a feature of the convolutional layer share the same weights. These weights are the convolution filter. Each feature has its own filter. Because the filters for each feature have the same size by the design in our example, the i-th neuron in feature f will apply feature f's filter to the same input pixels as the i-th neuron in feature g, but with feature g's filter. Note that this need not be the case since features can have filters of different sizes and steps as long as the outputs of each feature have the same dimensions.

Weighted Input Functions for Pooling Layers

As explained before, pooling layers are special cases of convolutional layers where the weighted input function is not the dot product. The purpose of a pooling layer is to reduce locality and reinforce feature recognition.

Common pooling layers use the maximum function, hence their name *max-pool layers*, or the average function.

The maximum function applied by a neuron in a max-pool layer to its input has the form:

$$S(\mathbf{x}) = \max \{x : \forall x \in \mathbf{x}\} \tag{5.36}$$

where $\mathbf{x}$ is the input vector to the neuron.

When training with back propagation, we encounter the term $\dfrac{\partial S}{\partial x}$ where $x \in \mathbf{x}$, so,

$$\frac{\partial S}{\partial x} = \begin{cases} 1, & \text{if } x = S(\mathbf{x}) \\ 0, & \text{otherwise} \end{cases} \tag{5.37}$$

The weighted input function for an average pool layer is:

$$S(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{5.38}$$

where the cardinality $||\mathbf{x}|| = n$ and $x_i \in \mathbf{x}$ with $i \in \{0, 1, ..., n-1\}$.

And

$$\frac{\partial S}{\partial x} = \frac{1}{n} \tag{5.39}$$

Biases and weights can be added to the computations and while these may add extra training parameters, often the optimization trade off is not worth it.

Activation Functions for Convolutional Layers

The convolution operation (either with dot product, maximum, etc.) is considered the weighted input function of a convolutional layer's neuron. The activation of a neuron can be any arbitrary non-linear activation function, or no activation (i.e. identity activation function) in which case, the output of a neuron is the same as its weighted input value.

It is common to see identity functions as activations of convolutional layers while pooling layers have a proper activation such as sigmoid or rectifiers. The reasoning is that if the pooling layer is max-pool, computation cycles can be saved if there is no activation in the convolutional output because the max-pool layer will discard results from smaller activations (note that most activation functions f are increasing, therefore, $\max(f(x_0), f(x_1), ..., f(x_n)) = f(\max(x_0, x_1, ..., x_n))$).

As stated in section 5.1.1, ReLU and their variants, especially in pooling layers, have shown to greatly improve convergence speed. During weight initialization of convolutional layers, setting biases to 1 has shown to reduce the dead ReLU problem.

Batch Normalizing Layers

There have been attempts to introduce normalization layers between two convolutional-pooling pairs and between fully connected layers to help with the vanishing gradient and regularization, but standard normalization of activations where $z_i \leftarrow \frac{z_i}{\sum_j z_j}$, discards information that the network may need [31]. The novel batch normalization technique is used to apply normalization while preserving all the information [31] and it is part of almost every state-of-the-art deep learning network today.

The practical moment to apply batch normalization in CNNs, however is to the result of the weighted input function of the pooling layer, and use the normalized result as input to the pooling layer's activation function.

Batch normalization is said to reduce or eliminate the need for dropout and other regularization techniques.

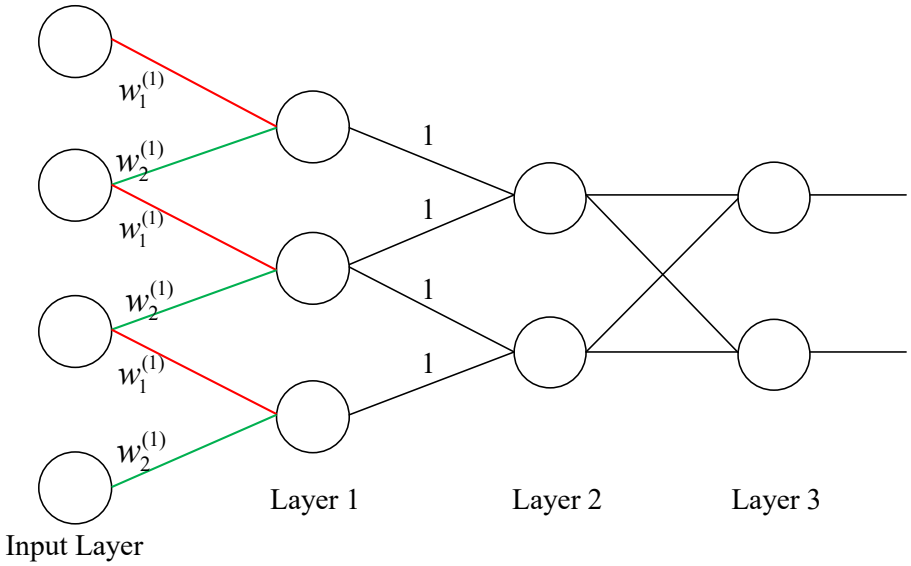


Figure 5.9: Simple convolutional neural network.

5.2.2 Training Convolutional Layers

We found from our survey [33][54][56] that the delta rule works for convolutional neural networks with little change, but these sources either gloss over the details or "leave it to the reader" without

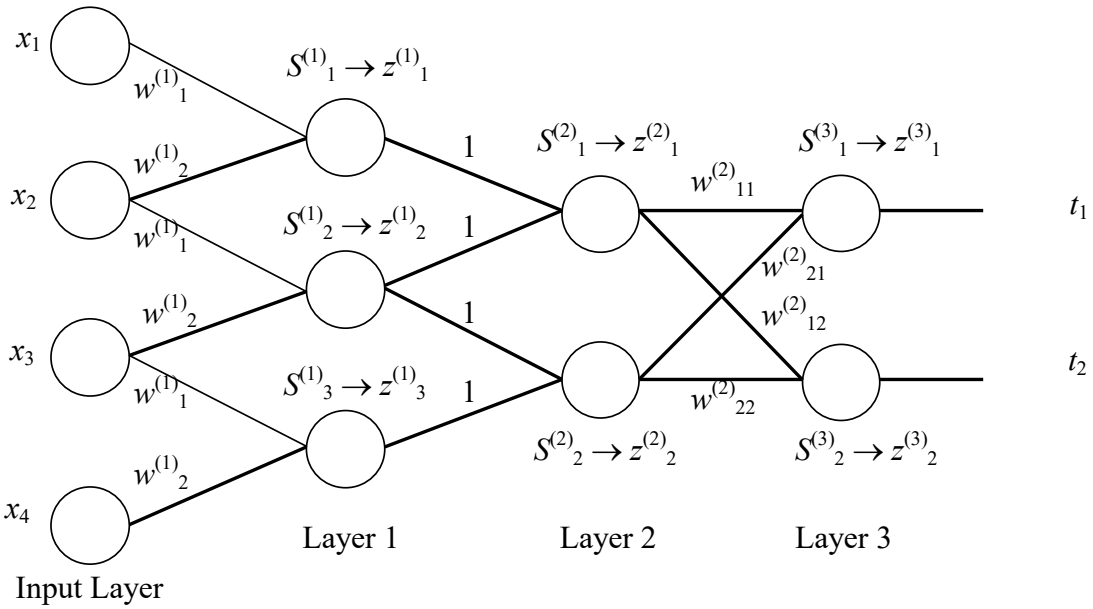


Figure 5.10: Branches affecting shared weights in a convolutional neural network.
Highlighted in bold are the connections from network in Figure 5.9 that affect shared weight $w_2^{(1)}$.

any convincing explanation.

Next we explain how the delta rule should works for convolutional neural networks with a small addition.

A typical convolutional layer with pooling layer architecture looks like in Figure 5.9.

Layer 1 is a convolutional layer and layer 2 is a pooling layer. Layer 3 is the output layer. Notice how neurons in layer 1 share their weights $w_1^{(1)}$ and $w_2^{(1)}$.

Figure 5.10 shows what branches of the network affect one of the shared weights. In this case $w_2^{(1)}$.

Practically every branch affects the specified shared weight. Computing $\Delta w_2^{(1)}$ using the back propagation method from equation 5.14 we obtain the result in equation 5.40.

For notation brevity $\frac{F}{\partial z_i^{(l)}} = F'\left(z_i^{(l)}\right)$ means partial derivative of a function F with respect to the activation function evaluated in the activation value of neuron i in layer l. So, for example, if $E = \sum_{i=1}^{2}\left(z_i^{(3)} - t_i\right)^2$ (the MSE objective function) and $z_i^{(3)} = f\left(S_i^{(3)}\right)$, then $\left.\frac{\partial E}{\partial f_1^{(3)}}\right|_{z_1^{(3)}} = \frac{E}{\partial z_1^{(3)}} = 2\left(z_1^{(3)} - t_1\right)$. Similarly, $S_i^{(l)}$ is the result of the weighted input function of neuron i in layer l and so on.

$$
\begin{aligned}
\Delta w_2^{(1)} =& \frac{\partial E}{\partial w_2^{(1)}} \\
=& \frac{\partial E}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial S_1^{(3)}} \left[\frac{\partial S_1^{(3)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial S_1^{(2)}} \left(\frac{\partial S_1^{(2)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial S_1^{(1)}} \frac{\partial S_1^{(1)}}{\partial w_2^{(1)}} + \frac{\partial S_1^{(2)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial S_2^{(1)}} \frac{\partial S_2^{(1)}}{\partial w_2^{(1)}} \right) + \right. \\
& \left. \frac{\partial S_1^{(3)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial S_2^{(2)}} \left(\frac{\partial S_2^{(2)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial S_2^{(1)}} \frac{\partial S_2^{(1)}}{\partial w_2^{(1)}} + \frac{\partial S_1^{(2)}}{\partial z_3^{(1)}} \frac{\partial z_3^{(1)}}{\partial S_3^{(1)}} \frac{\partial S_3^{(1)}}{\partial w_2^{(1)}} \right) \right] \\
& + \frac{\partial E}{\partial z_2^{(3)}} \frac{\partial z_2^{(3)}}{\partial S_2^{(3)}} \left[\frac{\partial S_2^{(3)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial S_1^{(2)}} \left(\frac{\partial S_1^{(2)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial S_1^{(1)}} \frac{\partial S_1^{(1)}}{\partial w_2^{(1)}} + \frac{\partial S_1^{(2)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial S_2^{(1)}} \frac{\partial S_2^{(1)}}{\partial w_2^{(1)}} \right) + \right. \\
& \left. \frac{\partial S_2^{(3)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial S_2^{(2)}} \left(\frac{\partial S_2^{(2)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial S_2^{(1)}} \frac{\partial S_2^{(1)}}{\partial w_2^{(1)}} + \frac{\partial S_1^{(2)}}{\partial z_3^{(1)}} \frac{\partial z_3^{(1)}}{\partial S_3^{(1)}} \frac{\partial S_3^{(1)}}{\partial w_2^{(1)}} \right) \right]
\end{aligned}
\tag{5.40}
$$

If we apply the delta rule to build the look up table as in equation 5.16:

$$
\begin{aligned}
\delta_1^{(3)} &= \frac{\partial E}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial {S^{(3)}}_1} \\
\delta_2^{(3)} &= \frac{\partial E}{\partial z_2^{(3)}} \frac{\partial z_2^{(3)}}{\partial {S^{(3)}}_2} \\
\delta_1^{(2)} &= \left(\delta_1^{(3)} \frac{\partial {S^{(3)}}_1}{\partial z_1^{(2)}} + \delta_2^{(3)} \frac{\partial {S^{(3)}}_2}{\partial z_1^{(2)}} \right) \frac{\partial z_1^{(2)}}{\partial S_1^{(2)}} \\
\delta_2^{(2)} &= \left(\delta_1^{(3)} \frac{\partial {S^{(3)}}_1}{\partial z_2^{(2)}} + \delta_2^{(3)} \frac{\partial {S^{(3)}}_2}{\partial z_2^{(2)}} \right) \frac{\partial z_2^{(2)}}{\partial S_2^{(2)}} \\
\delta_1^{(1)} &= \delta_1^{(2)} \frac{\partial S_1^{(2)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial S_1^{(1)}} \\
\delta_2^{(1)} &= \left(\delta_1^{(2)} \frac{\partial S_1^{(2)}}{\partial z_2^{(1)}} + \delta_2^{(2)} \frac{\partial S_2^{(2)}}{\partial z_2^{(1)}} \right) \frac{\partial z_2^{(1)}}{\partial S_2^{(1)}} \\
\delta_3^{(1)} &= \delta_2^{(2)} \frac{\partial S_2^{(2)}}{\partial z_3^{(1)}} \frac{\partial z_3^{(1)}}{\partial S_3^{(1)}}
\end{aligned}
$$

Then we use the computed deltas to find $\Delta w_2^{(1)}$ as follows:

$$
\Delta w_2^{(1)} = \delta_1^{(1)} \frac{\partial S_1^{(1)}}{\partial w_2^{(1)}} + \delta_2^{(1)} \frac{\partial S_2^{(1)}}{\partial w_2^{(1)}} + \delta_3^{(1)} \frac{\partial S_3^{(1)}}{\partial w_2^{(1)}} \tag{5.41}
$$

If we substitute the deltas and expand the result of equation 5.41, then we expand the right hand side of equation 5.40, the equivalency becomes evident.

This means that the delta rule still works for shared weights. The only implementation change is that instead of setting values for the change in weight, the values should be accumulated:

$$
\Delta w_k \leftarrow \Delta w_k + \delta_i \frac{\partial S}{\partial w_k} \tag{5.42}
$$

for every connection from a neuron j to a neuron i for which w_k is the shared weight parameter.

Gradient Propagation Through Convolutional Layers

Because pooling layers (such as max-pool or average-pool) are special cases of pooling layers, the delta rule applies exactly the same as long as we respect the weighted input function and its derivative.

The weighted input function for a convolutional layer is the dot product as applied in a filter convolution:

$$
S_j^{(l)} = \sum_i w_i^{(l)} z_k^{(l-1)} + b
$$

This is, the weighted input value of neuron j in layer l is the shared (filter) bias plus the sum of the products of each shared weight $w_i^{(l)}$ (filter element) and the activation of the neuron k in layer $l-1$ that is connected to neuron j with weight $w_i^{(l)}$.

Therefore $\dfrac{\partial S_j^{(l)}}{\partial z_k^{(l-1)}} = w_i^{(l)}$ as we expected and it is the same as with a fully connected layer. Of course, if the neuron k is not connected to j, then, this is the equivalent as having a connection with weight $w_{jk}^{(l)} = 0$ and the derivative is zero.

This last observation is used to reduce the implementation of convolutional layers to the existing fully connected layers where non-existing connections are conceptually established in order to fully connect the layers and a weight of zero is given to the conceptual connections. The resulting conceptual fully connected layer would have a large number of connections given the nature of convolutional layers and the large number of neurons. The matrix of weights ($\mathbf{W}$) is, however, a sparse matrix with a large number of zero entries compared to the non-zero. The nature of shared weights and sparse connections turns an impractically huge fully connected layer into a manageable and resource smart convolutional layer.

For the case of an optimized max-pooling layer (where weights of connections are ignored and assumed to be 1) equation 5.36 translates into:

$$S_j^{(l)} = \max\left\{ z_k^{(l-1)} : \forall \text{ neurons } k \text{ in layer } l-1 \text{ connected to neuron } j \text{ in layer } l \right\} \tag{5.43}$$

which means that the weighted input value of neuron j in layer l is the maximum of all the activations of the neurons k in layer $l-1$ connected to neuron j.

Therefore:

$$\frac{\partial S}{\partial z_k^{(l-1)}} = \begin{cases} 1, & \text{if } S_j^{(l)} = z_k^{(l-1)} \\ 0, & \text{otherwise.} \end{cases} \tag{5.44}$$

Of course, if the neuron k is not connected to j, then the derivative is zero, since the first portion of the conditional will never be true.

5.2.3 Implementation Remarks

Adding convolutional layers to our previous implementation was simple once the change to the delta rule was devised. The multidimensionality of the convolutional layers was realized logically while maintaining the internal one-dimensional structure of the neurons. This allowed us to keep the matrix format and reuse the MKL and OpenCL code for optimization. Pooling layers were devised as convolutional layers as well. The weighted input and activation functions were changed to reflect their behavior, but the underlying functionality was that of a convolutional layer with immutable weights.

The other only major change came in the form of protecting all the weights from data races. A critical section had to be created specific to each weight. Due to the massive parallelization possible by the inherent architecture of neural networks, changes to shared weights have to be protected because the delta rule now works by accumulating instead of setting values. This change did not impact the efficiency of the implementation as much because only convolutional layers have shared weights and the great amount of connections makes the neurons work on different weights at a time. Actual races to enter a shared weight critical section are uncommon.

To test our implementation we decided to create a convolutional neural network to recognize the MNIST image set of handwritten digits. The simple network featured one 20 feature convolutional layer with 5×5 filters of step 1 in both dimensions, one max-pooling layer with 20 2×2 pooling filters, one hidden fully connected layer with 40 neurons and the output layer with 10 neurons, and no dropout or batch normalization.

The results were impressive. Only after 4 epochs our network had reached the values from previous tests using fully connected networks. After 10 epochs it had surpassed the previous tests and was close to results obtained by Tensorflow implementations 99% recognition that apply dropout.

Optimizations

A matrix is considered sparse when most of its elements are zero. If we have the CNN portrayed in Figure 5.9 and we extend layer 1 to be a fully connected layer using conceptual connections of weight zero, we obtain a weight matrix of the form:

$$\mathbf{W} = \begin{bmatrix} w_1 & w_2 & 0 & 0 \\ 0 & w_1 & w_2 & 0 \\ 0 & 0 & w_1 & w_2 \end{bmatrix} \tag{5.45}$$

While this matrix can be barely considered sparse, if the number of neurons in the input layer increases, the number of rows and number of columns per new neuron increases by 1, but there will be still only 2 non-zero entries per row because the feature filter has only 2 parameters. So, we see that the larger input and the convolutional layer get, the sparser the matrix becomes. In practical applications, a small convolutional layer has over 1400 neurons (for an input of 700 elements and 20 convolutional features). The resulting matrix has a large level of sparsity. If this were a fully connected layer, the number of parameters would be huge and the speed of computation would suffer. Multiplication of sparse matrices saves time because the large number of zero entries is ignored.

Sparse matrix multiplication implementations take advantage of the packed matrix concept. A well packed matrix not only saves space, but it can help with proximity caching, making use

Table 5.1: Example of sparse matrix in zero-based indexing CSR format.

Index	0	1	2	3	4	5
val	w_1	w_2	w_1	w_2	w_1	w_2
col_ind	0	1	1	2	2	3
row_ptr	0	2	4	6	-	-

during operations of nearby memory locations, mitigating the penalty of memory access. For our implementation, the sparse matrix packing method of choice was the *Compressed Sparse Row* (CSR) format [30].

The CSR format for a sparse matrix is composed of three arrays: (val, col_ind, row_ptr).

The *val* array stores the non-zero elements of the matrix as they are traversed in row-major order (top-to-bottom, left-to-right). The size of this array is the number of non-zero elements in the matrix.

The *col_ind* array stores the column indices for each element inside its row. The size of *col_ind* is the same as *val*'s.

The *row_ptr* array stores the index in *val* at which a new row starts. The size of *row_ptr* is the number of rows plus 1. Usually, the last element of this array is the number of non-zero elements in the matrix.

For example, the zero-based indexing CSR format for matrix $\mathbf{W}$ in equation 5.45 is shown in Table 5.1.

With the concept of sparse matrix, the matrix multiplication variants for forward propagation 5.34 and back propagation 5.35 can be reused.

5.3 Results

With the knowledge gained with our survey and observations on neural network architectures, training algorithms and deep learning, we implemented our own Feedforward Neural Network library. The library was implemented in C++ 11 utilizing modern multicore computing techniques and CPU architectures along with optimization technologies such as OpenMP for multithreaded programming, Intel MKL for highly optimized vector and matrix operations, OpenCL for co-processor and GPU offloading to achieve even higher processing speeds and take advantage of the huge parallelization possible by the Many-Integrated Cores designs of Intel Xeon Phi co-processors and AMD GPUs.

The motivation for creating our own library was to have the ability to inspect, tweak and monitor internal details of a neural network such as internal deltas, change in parameters, learning rate schedule, and to have fine grain control of the training and operation of our networks with research

purposes.

A condensed version of our library can be found in appendix B. The original code has support for 32 bits floating point and 64 bits floating point values. Only the 32 bits portion is shown here.

Chapter 6

Quantitative Evaluation Method for Neural Network Weight Initialization Strategies

The adaptive parameters in fully connected feed-forward neural networks are usually referred to as weights. These parameters change during training to approximate the desired function using some training algorithm. These weights in a neural network are often the coefficients applied to the linear combination of the inputs to every computing unit, including the bias (the weight of a constant extra input of value 1).

Parameter or weight initialization (including biases) for neural networks has been empirically shown to be a critical step during training. The motivation for good parameter initialization is to chose values in range of the attractor of the final desired solution in order to speed up training, increase speed of convergence and probability of correct classification. A proper weight initialization can reduce the number of training samples needed by the network to achieve an accepted error value, meanwhile, improper initialization could set the initial hyperplane division so far away from the optimum solution that it may lead to divergence and an approximation of the final function that is no better than chance.

Initialization strategies are evaluated by training a neural network and comparing the results with other trained neural networks that applied different initialization techniques. However, there appears to be no practical quantitative measure that can compare different weight initialization strategies regarding the gain in convergence rate and convergence accuracy for the same neural network architecture, and thus, there is a lack of formal evaluation that can determine whether

experimental results are consistent or simply the result of random anomalies given the random nature of existing initialization methods.

We developed a method to quantitatively compare different initialization techniques by using multivariate analysis, statistics of extremes, analysis of variance and estimation theory. The results allow for the assessment of whether some strategy is superior to another or if the differences between them are not significant enough to warrant such superiority. The goal is to provide a quick, objective method to compare initialization techniques and select the best possible beforehand without having to perform multiple training sessions for each candidate strategy and directly compare final results. Selecting the statistically superior strategy for training gives the neural network to be trained the advantage to converge to a better approximation faster with high probability. We show how, if such strategy exists among the techniques being compared, our method successfully determines the best strategy within the first epoch of training.

6.1 Assessing the Effect of Weight Initialization on Learning Speed and Approximation Accuracy

Let P_{eij}, denote an estimate of the probability of correct classification in epoch e, where $e = 1, 2, ..., E$, using the sampling initialization strategy i, where $i = 1, 2, ..., I$, and the number of samples j, with $j = 1, 2, ..., J$. The index j represents the j-th initialization sampling set θ_{eij} chosen for the sampling strategy i at epoch e. The number of times J we reinitialize the weights and repeat the neural network training could be different for every initialization strategy J. Due to the large number of samples used to train the neural network the probability estimate P_{eij} converges to its true value for this initialization set of random weights. Each time we repeat the training of the neural network using the same initialization strategy the new P_{eij+1} is different than the probabilities found in the previous trainings using the same sampling strategy i. Thus P_{eij} is a random variable having a probability density function $f_{ei}(p)$, and a cumulative distribution function $F_{ei}(p)$.

The following theorem establishes the probability functions, the means and the variances of the maximum, and minimum of the random variables P_{eij}. The purpose of this theorem is to be used in order to understand if there is any overlap between the probability estimates of the various weight initialization strategies.

Theorem 6.1.1. *Let $P_{ei1}, P_{ei2}, P_{ei3}, ..., P_{eiJ}$, be random variables with cumulative distribution function F_{ei}, $j = 1, 2, ..., J$, for a given epoch e, and a given weight initialization strategy i. Where $e = 1, 2, ..., E$, and $i = 1, 2, ..., I$. Let $P_{ei\max} = \max(P_{ei1}, P_{ei2}, ..., P_{eiJ})$ be a random variable which is the maximum of the J random variables $P_{ei1}, P_{ei2}, ..., P_{eiJ}$, then the cumulative distribution func-*

tion of this maximum is:

$$F_{\max}(p) = (F_{ei}(p))^J \tag{6.1}$$

therefore the probability density function of the maximum is:

$$f_{\max}(p) = J(F_{ei}(p))^{J-1} f_{ei}(p). \tag{6.2}$$

The expected value for the maximum is:

$$E(P_{ei\,\max}) = \int_{P_{\min}}^{P_{\max}} pJ(F_{ei}(p)^{J-1}) f_{ei}(p)dp \tag{6.3}$$

and the variance is:

$$\sigma^2_{\max} = \int_{P_{\min}}^{P_{\max}} p^2 J(F_{ei}(p))^{J-1} f_{ei}(p)dp - \left(\int_{P_{\min}}^{P_{\max}} pJ(F_{ei}(p))^{J-1} f_{ei}(p)dp \right)^2. \tag{6.4}$$

The 95% right side tolerance interval for the max *is:*

$$P_{\max} < E(P_{ei\,\max}) + a\sigma_{\max} \tag{6.5}$$

where the a is chosen so that $P(P_{ei\,\max} > a) = 0.05$.

Proof. Let $P_{ei\,\max} = \max(P_{ei1}, P_{ei2}, ..., P_{eiJ})$ denote a random variable which is the maximum of the J random variables $P_{ei1}, P_{ei2}, ..., P_{eiJ}$ then $F_{\max}(p) = P(P_{ei\,\max} \le p) = P(\max(P_{ei1}, P_{ei2}, ..., P_{eiJ}) \le p) = P(P_{ei1} \le p, P_{ei2} \le p, ..., P_{eiJ} \le p) = P(P_{ei1} \le p) \cdot P(P_{ei2} \le p) \cdot ... \cdot P(P_{eiJ} \le p)$. The product of probabilities in the last term is due to independence of the random weights selected in each separate neural network training. Thus from the above we have:

$$F_{\max}(p) = F_{ei}(p)F_{ei}(p)...F_{ei}(p) = (F_{ei}(p))^J$$

From this result we obtain the density function:

$$\begin{aligned} f_{\max}(p) &= \frac{d}{dp}(F_{ei}(p))^J \\ &= J(F_{ei}(p))^{J-1} f_{ei}(p). \end{aligned}$$

The mean of the maximum is:

$$\begin{aligned} \mu_{\max} = E(P_{ei\,\max}) &= \int_{P_{\min}}^{P_{\max}} pJ(F_{ei}(p))^{J-1} f_{ei}(p)dp \\ &= P_{\max} - \int_{P_{\min}}^{P_{\max}} (F_{ei}(p))^J dp \end{aligned}$$

and second moment:

$$\begin{aligned} E(P^2_{ei\,\max}) &= \int_{P_{\min}}^{P_{\max}} p^2 J(F_{ei}(p))^{J-1} f_{ei}(p)dp \\ &= P^2_{\max} - 2\int_{P_{\min}}^{P_{\max}} p(F_{ei}(p))^J dp. \end{aligned}$$

Therefore, the variance is:

$$\begin{aligned}\sigma^2_{\max} &= E(P^2_{ei\,\max}) - E^2(P_{ei\,\max}) \\ &= P^2_{\max} - 2\int_{P_{\min}}^{P_{\max}} p(F_{ei}(p))^J dp - \left(P_{\max} - \int_{P_{\min}}^{P_{\max}} (F_{ei}(p))^J dp\right)^2.\end{aligned}$$

The right sided 95% tolerance interval for the $P_{ei\,\max}$ is:

$$P_{ei\,\max} \leq \mu_{\max} + a\sigma_{\max}$$

where a is chosen so that $P(P_{ei\,\max} > a) = 0.05$. □

Theorem 6.1.2. *Let $P_{ei1}, P_{ei2}, P_{ei3}, ..., P_{eiJ}$, be random variables with cumulative distribution function F_{ei}, $j = 1, 2, ..., J$, for a given epoch e, and a given weight initialization strategy i. Where $e = 1, 2, ..., E$, and $i = 1, 2, ..., I$. Let $P_{ei\,\min} = \min(P_{ei1}, P_{ei2}, ..., P_{eiJ})$ be a random variable which is the minimum of the J random variables $P_{ei1}, P_{ei2}, ..., P_{eiJ}$, then the cumulative distribution function of this minimum is:*

$$F_{min}(p) = 1 - (1 - F_{ei}(p))^J \tag{6.6}$$

therefore the probability density function of the minimum is:

$$f_{min}(p) = J(1 - F_{ei}(p))^{J-1} f_{ei}(p). \tag{6.7}$$

The expected value for the minimum is:

$$E(P_{ei\,\min}) = \int_{P_{\min}}^{P_{\max}} pJ(1 - (F_{ei}(p))^{J-1}) f_{ei}(p) dp \tag{6.8}$$

and the variance is:

$$\sigma^2_{\min} = \int_{P_{\min}}^{P_{\max}} p^2 J(1 - (F_{ei}(p))^{J-1}) f_{ei}(p) dp - \left(\int_{P_{\min}}^{P_{\max}} pJ(1 - F_{ei}(p))^{J-1} f_{ei}(p) dp\right)^2. \tag{6.9}$$

The 95% left sided tolerance interval for the min *is:*

$$E(P_{ei\,\min}) - a\sigma_{\min} < P_{ei\,\min} \tag{6.10}$$

where the a is chosen so that $P(P_{ei\,\min} < a) = 0.05$.

Proof. Let $P_{ei\,\min} = \min(P_{ei1}, P_{ei2}, ..., P_{eiJ})$ denote a random variable which is the minimum of the J random variables $P_{ei1}, P_{ei2}, ..., P_{eiJ}$, then the probability distribution function of this minimum

is:

$$\begin{aligned} F_{\min}(p) &= P(\min(P_{ei1}, P_{ei2}, ..., P_{eiJ}) \leq p) \\ &= 1 - P(\min(P_{ei1}, P_{ei2}, ..., P_{eiJ}) > p) \end{aligned}$$

which implies that

$$\begin{aligned} F_{\min}(p) &= 1 - P(P_{ei1} > p, P_{ei2} > p, ..., P_{eiJ} > p) \\ &= 1 - (1 - P(P_{ei1} \leq p)) \cdot (1 - P(P_{ei1} \leq p)) \cdot ... \cdot (1 - P(P_{ei1} \leq p)) \\ &= 1 - (1 - F_{ei}(p))^J \end{aligned}$$

From the result above we obtain

$$\begin{aligned} f_{\min}(p) &= \frac{d}{dp}\left(1 - (1 - F_{ei}(p))^J\right) \\ &= J(1 - F_{ei}(p))^{J-1} f_{ei}(p). \end{aligned}$$

The mean of the minimum is:

$$\begin{aligned} \mu_{\min} = E(P_{ei\,\min}) &= \int_{P_{\min}}^{P_{\max}} pJ(1 - F_{ei}(p))^{J-1} f_{ei}(p)dp \\ &= P_{\min} - \int_{P_{\min}}^{P_{\max}} (1 - F_{ei}(p))^J dp \end{aligned}$$

and second moment:

$$\begin{aligned} E(P_{ei\,\min}^2) &= \int_{P_{\min}}^{P_{\max}} p^2 J(1 - F_{ei}(p))^{J-1} f_{ei}(p)dp \\ &= P_{\min}^2 + 2\int_{P_{\min}}^{P_{\max}} p(1 - F_{ei}(p))^J dp. \end{aligned}$$

Therefore, the variance is:

$$\begin{aligned} \sigma_{\min}^2 &= E(P_{ei\,\min}^2) - E^2(P_{ei\,\min}) \\ &= P_{\min}^2 + 2\int_{P_{\min}}^{P_{\max}} p(1 - F_{ei}(p))^J dp - \left(P_{\min} - \int_{P_{\min}}^{P_{\max}} (1 - F_{ei}(p))^J dp\right)^2. \end{aligned}$$

The one sided 95% tolerance interval for the $P_{ei\,\min}$ is:

$$\mu_{\min} - a\sigma_{\min} \leq P_{ei\,\min}$$

where a is chosen so that $P(P_{ei\,\min} < a) = 0.05$. □

If we compute the probability of correct classification for epoch e using two different weight initialization probability distributions, and if for each one of the initialization strategies we repeat the

training J times. Then for the first initialization strategy we will obtain J different probabilities of correct classification for epoch e. Similarly if we repeat the training of the neural network J times using the second initialization strategy, then we will find J different probabilities of correct classification for epoch e. If the probabilities obtained by the first initialization are all greater than the probabilities of the second, and the probability density function of the minimum of the first for epoch e, does not overlap with the probability density function of the maximum of the second for epoch e, then the first initialization strategy is always superior than the second. This theory can be extended to more than two weight initialization strategies.

One can also compute the 95% tolerance intervals of the minimum of the first and if they do not overlap with the tolerance intervals of the maximum of the second then with probability 95% the first is superior to the second. In the case the computed probabilities for epoch e overlap for various strategies then we can use analysis of variance to test the hypothesis that the means are equal. If the hypothesis is rejected that implies that there is a statistical significance between the weight initialization strategies. In this case the strategy with the highest average has an advantage over the strategy with the lowest average. Finally if there is an overlap in the probability distribution functions of correct recognition for a given epoch of the various weight strategies then we identify the one weight strategy with the mean shifted to the right compared to all other strategies and this one would give us an advantage and it will produce the fastest learning neural network of its kind.

6.2 Minimum Number of Epochs Needed for the Neural Network to Converge Using a Large Sample

The data homogeneity expressed by a variety of measures, parametric and non-parametric, such as: variance, variance-covariance matrices, range, cross correlation, or various similarity measures, is an important component relating to the number of epochs needed to teach the neural network to classify the inputs in the correct class with high probability. The number of training samples, the number of validation samples, and the number of test samples are a function of the neural network architecture, which includes the input vector and its characteristics, the number of hidden layers, the number of neurons per hidden layer, and finally the output layer.

The characteristics of the input layer include the number of elements of the input vector, and dependencies between the input vector components. As we pointed out in previous sections weight initialization and learning rate strategies can affect the convergence of the neural network. The law of large numbers that applies to the parameter estimation theory in statistics, applies to properly constructed neural networks as well. The ideas of sampling theory and experimental design apply

to neural networks also. The concept of "epochs," however, is unique to neural networks.

Logically the probability of correct classification should increase as the number epoch increases, to a certain extent where the network is actually learning instead of diverging. This is not always the case. In all the experiments we performed so far, we observed that the probability of correct classification could decrease from one epoch to the next, especially in the initial epochs. As the epoch number increases the probability of correct classification reaches its maximum value for the first time at a certain epoch number k, then for the following few epochs the probability of correct classification often fluctuates with values less than the maximum value, and finally stabilizes at a the maximum or a value close to the maximum, remaining there within a small ϵ for the following epochs in a steady state. At this point, it is feasable to admit that the network stalled and will not learn further. Extra training may result in overfitting or learning decay. The concept of early stopping when the stable state is detected has been studied, involving halting training once the network stops learning at a significant rate to avoid these drawbacks [56].

The number of epochs needed for a network to attain its maximum probability of correct classification, epoch of maximum classification, is a random variable that depends on many factors, including the neural network architecture, weight initialization strategy, number of training samples, learning rate and learning rate adjustment strategy. The number of epochs needed for the neural network to come to a steady state is also a random variable depending on the same factors.

Let C_{eij} be the classification accuracy for network j, using initialization strategy i at epoch e. We define here the *epoch-speed-of-learning* $\hat{e}$ for a neural network with a given architecture, set of training samples, weight initialization strategy, learning rate and learning rate adjustment strategy as $\forall k, l \geq \hat{e}, |C_{kij} - C_{lij}| < \epsilon$ for a small ϵ. This is, the epoch-speed-of-learning is the minimum number of epochs needed for a neural network to come to a steady state.

Clearly, it is preferred to attain a small epoch of maximum classification and a small epoch speed of learning while maximizing the classification accuracy obtained in each of these epochs.

The selection of initialization strategy impacts the epoch where a network attains its maximum classification as well as the epoch-speed-of-learning of the network. We claim that initialization strategies deemed best among the set of tested strategies by our quantitative method will also give best epoch of maximum classification and epoch-speed-of-learning with greater classification accuracy than the other strategies.

6.3 Application of the Theory

We applied all these ideas in our experiment using four different, widely recommended weight initialization strategies selected from our survey to determine if different weight initialization strategies

actually offered different convergence rate results.

The methods we applied as weight initialization strategies for our experiment were:

1. $w_i = \text{uniform}(-1, 1) \cdot 0.5$
2. $w_i = \text{uniform}(-1, 1) \cdot 4\sqrt{6\left(fan\_in_i + fan\_out_i\right)^{-1}}$
3. $w_i = \text{gaussian\_random}(0, 1)$
4. $w_i = \text{gaussian\_random}(0, 1/\sqrt{fan\_in_i})$

and throughout the rest of this paper we will refer to them mostly by number,

One of the most important and most widely used algorithms for neural network training is backpropagation with gradient descent. Most of today's state-of-the-art neural networks use this algorithm or some variant or extension with modifications to suit the network specifics. Therefore, to show how our theory works we chose one of the simplest variants of backpropagation using stochastic gradient descent.

We trained 400 fully connected feedforward neural networks to recognize the MNIST dataset of handwritten digits for our experiment using our custom neural network library. We trained 100 networks per weight initialization strategy.

6.3.1 Network Architecture

Each network we trained figured the same architecture, adapted to recognize the MNIST samples.

Each sample in the MNIST dataset is composed of the pixel intensity for each pixel in the image and the label indicating what digit d that the image represents.

Each sample was used as-is without any kind of preprocessing or filtering before feeding it to the neural network. The image is represented in gray scale with dimensions of 28×28 pixels, where each pixel value is defined as the gray scale intensity, a value between 0 to 255, which we normalized in the range $[0, 1]$ using double precision floating point. The sample was arranged as a one-dimensional array representing the two-dimensional image in row-major format, feeding the 784 input values to the network.

The target output was mapped from the digit d to a vector of zeros and ones where all elements are zero except for the element at position d which is one. For example the digit 4 is mapped to $D = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$.

Every network was composed of three layers: input layer with 784 input units, hidden layer with 45 fully connected sigmoid units and output layer with 10 fully connected sigmoid units.

While we recognize that there are better architectures to improve recognition and convergence rate, our design aimed to create simple networks that produce mutually comparable results regarding weight initialization instead of producing state of the art recognition which is outside the scope of this research.

6.3.2 Training Phase

The MNIST dataset is composed of 60000 training samples and 10000 training samples. We divided the training samples into 10000 randomly selected validation samples and the remaining 50000 samples to use for actual training. The validation samples were selected once at the start and were kept the same over the life of the experiment in order to avoid validation samples dependent results. The testing during training was made over the validation samples until all epochs were completed. Then, a final test was conducted over the testing sample. This ensures that our networks generalize over any image, avoiding over fitting on the actual testing data.

Before training a network, its weights were initialized using the corresponding technique. At the start of every epoch, all the training samples were randomized taking care that the order of the samples in epoch e was always the same for every network. This guarantees that the only parameter affecting the convergence rate is the weight initialization technique used.

Stochastic gradient descent was used to update the weights with a mini-batch size of $m = 10$. Mini-batch size controls how many samples are shown to the network before updating the weights by the accumulated error in each sample. So for the 50000 samples, 5000 mini-batches were shown to the neural network in each epoch.

6.3.3 Testing Phase

Testing was performed after an epoch was completed using the validation samples. The test reports the percentage of correct classification. Thus, the estimate of the probability of correct classification at the end of epoch e is $\frac{total\_correct}{total\_samples}$.

After all epochs are completed, a final test is conducted using the testing sample set to ensure that the neural network is able to generalize and classify correctly samples outside the training and validation sets with high probability. In all trials of the experiment the generalization accuracy, defined as the difference between validation test and final test, was below 0.05%.

In this research, we focus on the classification probability at the end of epoch 1, because this is an indication of how fast the neural network is learning.

6.3.4 Experimental Outcomes

Let μ_1, μ_2, μ_3 and μ_4 be the mean probabilities for weight initialization strategies #1, #2, #3 and #4 respectively.

Table 6.1, 6.2, 6.3 and 6.4 show the classification accuracy obtained after the first epoch of training for each initialization strategy applied to 100 neural networks. In order to see if the fluctuations reported in these results are significant or random, first we did a coarse grain analysis by testing the null hypothesis $H_0 : \mu_1 = \mu_2 = \mu_3 = \mu_4$. Table 6.5 shows the results of analysis of variance to test the null hypothesis [71]. With $F = 435.32$, the results show that the fluctuations are indeed significant with 95% confidence; therefore, the null hypothesis was rejected.

Further statistical investigation with results in Table 6.6 showed that μ_1 and μ_4 are not significantly different. Results from Table 6.7 showed that the rejection of the null hypothesis is due to the fact that μ_3 was statistically different than the other means, μ_2 was also statistically different than μ_1 and μ_4 as seen in Table 6.8, but not as much, while Table 6.9 shows that μ_2 and μ_3 are significantly different.

Looking at the 95% confidence intervals of the means for the first epoch using the t distribution with 99 degrees of freedom we obtain: $94.81 < \mu_1 < 94.90$, $94.45 < \mu_2 < 94.64$, $92.89 < \mu_3 < 93.07$, $94.82 < \mu_4 < 94.99$.

It seems that the analysis of variance and means has led to results where the strategies, ordered from worst to best for the specified architecture, are #3, #2, #1, #4, with strategies #1 and #4 offering similar convergence rates. Next we applied our theory to see if, in the extreme cases, there is a chance of obtaining similar probabilities of correct classification in epoch 1 using weight initialization strategies #1 and #3.

To apply our method, we need to find a probability distribution that fits the measurements of each strategy we are going to compare.

The variables for epoch 1 probability distribution function of correct classification for each of the strategies are random and based on the histogram of the sample data and using the Chi-square goodness of fit, we concluded that the best probability distribution that fits the data is of the form

$$f_{ei}(p) = a_{ei}p^4 + b_{ei}p^3 + c_{ei}p^2 + d_{ei}p + e_{ei}. \tag{6.11}$$

where i is the weight initialization strategy and e is the epoch number.

We estimate the $m = 5$ parameters a_{ei}, b_{ei}, c_{ei}, d_{ei} and e_{ei} of the distribution 6.11 using the data collected and test the goodness of the fit with algorithm 3.

The histogram obtained from the experimental results is sensitive to the interval selection, and

Table 6.1: 100 Independent Neural Network Trainings After 1 Epoch Using Weight Initialization Strategy #1

Network No.	Probability (%) of Correct Classification	Network No.	Probability (%) of Correct Classification
1	95.24	51	95.22
2	94.94	52	95.28
3	95.22	53	94.39
4	94.73	54	94.72
5	95.5	55	93.94
6	94.99	56	95.22
7	94.6	57	95.06
8	94.71	58	94.72
9	94.92	59	94.39
10	95.03	60	95
11	95.45	61	94.32
12	94.18	62	95.16
13	94.64	63	95.29
14	94.82	64	95.41
15	94.46	65	95.03
16	94.62	66	95.19
17	95.28	67	94.25
18	95.18	68	94.17
19	95.18	69	95.05
20	94.32	70	95.2
21	95.05	71	94.9
22	95.29	72	95.12
23	94.98	73	95.49
24	94.11	74	94.94
25	94.46	75	95
26	94.34	76	95.22
27	94.84	77	94.59
28	95.07	78	94.61
29	94.97	79	94.99
30	94.67	80	94.56
31	94.1	81	95.04
32	95.33	82	94.91
33	94.82	83	94.96
34	95.1	84	94.35
35	95.18	85	94.72
36	94.49	86	95.25
37	94.54	87	95.09
38	94.99	88	95.04
39	95.02	89	95.06
40	94.95	90	94.95
41	95.55	91	95.09
42	95.04	92	94.8
43	94.66	93	95.15
44	93.66	94	94.98
45	95.13	95	95.11
46	95.26	96	94.97
47	95.45	97	94.41
48	94.35	98	94.87
49	95.01	99	95.01
50	94.78	100	95.42
Mean	94.89	Variance	0.14

Table 6.2: 100 Independent Neural Network Trainings After 1 Epoch Using Weight Initialization Strategy #2

Network No.	Probability (%) of Correct Classification	Network No.	Probability (%) of Correct Classification
1	94.6	51	94.23
2	94.11	52	94.32
3	94.88	53	94.38
4	94.57	54	92.78
5	95.2	55	94.06
6	94.96	56	94.37
7	95.04	57	94.95
8	94.68	58	94.16
9	93.81	59	95
10	94.96	60	94.39
11	94.4	61	94.7
12	94.48	62	93.92
13	93.71	63	95.44
14	95.05	64	95.02
15	94.48	65	94.97
16	94.91	66	93.87
17	94.87	67	94.68
18	94.6	68	93.28
19	95.14	69	94.46
20	95.1	70	95.03
21	94.88	71	94.83
22	94.28	72	94.28
23	94.86	73	94.65
24	94.86	74	94.66
25	95.01	75	94.68
26	93.86	76	94.65
27	94.5	77	94.9
28	94.35	78	94.46
29	94.34	79	95.01
30	94.1	80	92.92
31	94.94	81	93.64
32	94.47	82	94.52
33	95.1	83	94.71
34	95.04	84	94.79
35	95.32	85	94.64
36	93.43	86	94.66
37	94.16	87	94.44
38	94.45	88	94.78
39	94.71	89	94.03
40	94.64	90	94.69
41	94.69	91	94.41
42	93.94	92	94.47
43	94.79	93	94.97
44	94.16	94	94.66
45	94.67	95	94.68
46	95.21	96	94.96
47	94.46	97	95.01
48	94.38	98	94.42
49	94.54	99	94.52
50	94.09	100	94.92
Mean	94.55	Variance	0.23

Table 6.3: 100 Independent Neural Network Trainings After 1 Epoch Using Weight Initialization Strategy #3

Network No.	Probability (%) of Correct Classification	Network No.	Probability (%) of Correct Classification
1	93.08	51	93.17
2	93.11	52	93.06
3	92.33	53	92.6
4	92.87	54	93.1
5	92.86	55	92.71
6	92.75	56	93.06
7	92.64	57	93.4
8	93.52	58	93.68
9	93.09	59	93.35
10	93.47	60	92.56
11	93.04	61	92.48
12	92.93	62	92.84
13	92.52	63	92.99
14	93.12	64	92.92
15	93.04	65	92.93
16	93.16	66	93.29
17	92.97	67	92.37
18	93.46	68	93.37
19	93.04	69	93.05
20	93.34	70	92.7
21	93.94	71	93.58
22	92.89	72	93.44
23	92.9	73	93.15
24	92.98	74	93.12
25	93.88	75	92.31
26	93.15	76	92.19
27	92.77	77	92.8
28	92.71	78	92.34
29	92.69	79	93.33
30	93.56	80	93.36
31	93.17	81	93.94
32	92.88	82	92.89
33	92.02	83	92.9
34	93.13	84	92.98
35	93.04	85	93.88
36	93.63	86	93.15
37	93.05	87	92.77
38	92.94	88	92.71
39	91.64	89	92.69
40	92.62	90	93.56
41	93.73	91	93.17
42	92.71	92	92.88
43	92.68	93	92.02
44	93.24	94	93.13
45	92.28	95	93.04
46	92.64	96	93.63
47	91.98	97	93.05
48	92.99	98	92.94
49	93.59	99	91.64
50	93.39	100	92.62
Mean	92.98	Variance	0.21

Table 6.4: 100 Independent Neural Network Trainings After 1 Epoch Using Weight Initialization Strategy #4

Network No.	Probability (%) of Correct Classification	Network No.	Probability (%) of Correct Classification
1	95.25	51	94.56
2	94.91	52	94.4
3	93.93	53	95.5
4	94.68	54	95.44
5	94.73	55	94.79
6	94.62	56	94.99
7	94.93	57	95.64
8	95.44	58	95.44
9	94.92	59	94.99
10	94.02	60	95.01
11	94.6	61	94.43
12	94.91	62	95.14
13	94.62	63	95.31
14	95.07	64	95.3
15	93.86	65	95.4
16	95	66	95.53
17	94.49	67	94.34
18	94.81	68	94.73
19	95.15	69	95.14
20	95.24	70	94.73
21	94.61	71	94.73
22	95.09	72	95.56
23	95.02	73	94.89
24	95.16	74	94.92
25	94.7	75	94.25
26	94.71	76	94.54
27	95	77	95.36
28	94.64	78	94.63
29	94.4	79	95.37
30	94.99	80	94.87
31	94.07	81	95.08
32	93.58	82	95.58
33	95.33	83	95.55
34	95.37	84	94.98
35	94.65	85	95.2
36	95.03	86	94.73
37	95.02	87	93.98
38	95.46	88	94.2
39	95.09	89	94.79
40	94.79	90	94.1
41	95.21	91	95.03
42	95.16	92	94.89
43	95.72	93	94.67
44	95.41	94	94.39
45	94.07	95	94.97
46	94.95	96	95.2
47	94.98	97	95.03
48	95.44	98	95.03
49	94.95	99	95.49
50	95.26	100	94.77
Mean	94.91	Variance	0.19

the neural network noise due to the weight initialization, learning rate selection, and neural network design. These factors also affect the goodness of fit of the model with the histogram.

Table 6.5: ANOVA results for testing the null hypothesis $H_0 : \mu_1 = \mu_2 = \mu_3 = \mu_4$.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	251.34	3	83.78
Within-treatments	76.21	396	0.1925
Total	327.55	399	$F = 435.32$

Table 6.6: ANOVA results for testing whether μ_1 and μ_4 are significantly different.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	0.0166	1	0.0166
Within-treatments	33.03	198	0.1668
Total	33.04	199	$F = 0.0993$

Table 6.7: ANOVA results for testing whether μ_1, μ_3 and μ_4 are significantly different.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	245.06	2	122.5301
Within-treatments	53.75	297	0.181
Total	298.81	299	$F = 677.00$

Table 6.8: ANOVA results for testing whether μ_1, μ_2 and μ_4 are significantly different.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	8.1692	2	4.0846
Within-treatments	55.49	297	0.1868
Total	63.65	299	$F = 21.864$

Table 6.9: ANOVA results for testing whether μ_2 and μ_3 are significantly different.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	122.85	1	122.85
Within-treatments	43.18	198	0.2181
Total	166.04	199	$F = 563.27$

Table 6.10: Parameter estimation for probability density function $f_{11}(x)$ for strategy #1 in epoch 1.

Parameter	Estimated value
a_{11}	$-5.9939047533296919 \times 10^9$
b_{11}	$22556963627.494751 \times 10^{10}$
c_{11}	$-3.1832342639344265 \times 10^{10}$
d_{11}	$1.9964573972344696 \times 10^{10}$
e_{11}	$-4.6953597466079311 \times 10^9$

From Table 6.1, representing weight initialization strategy #1, namely $w_i = \text{uniform}(-1, 1) \cdot 0.5$, we see that for epoch 1, the probability of correct classification is in the interval $[0.9366, 0.9555]$ and we have $J = 100$ measurements.

As algorithm 3 indicates, we need to estimate the parameters for the distribution $f_{11}(p)$ with the form specified in 6.11, where $0.9366 \leq p \leq 0.9555$. The parameters are shown in Table 6.10.

Let's consider $k = 17$ intervals. Let A_i be the area under the probability distribution $f_{11}(p)$ for interval i. We compute A_i using numerical integration. Note that $\forall i, A_i \leq 1$ and $\sum_i A_i = 1$. The number of expected samples in the i-th interval E_i is $E_i = nA_i$. Table 6.11 shows the actual (O_i) and expected (E_i) number of samples per interval.

Using the existing data we can compute the Chi-square value for this dataset:

$$\chi^2 = \sum_{i=1}^{17} \frac{(O_i - E_i)^2}{E_i} = 15.76$$

with $k - 1 - m = 11$ degrees of freedom. This value is below the Chi-square critical point for 5% significance, therefore, the proposed function is a good fit as the probability density function to represent the sampled data.

Algorithm 3 Procedure to test the goodness of fit of a probability density function to the classification data.

1: Using the sample data, estimate parameters for the distribution to test.
2: Based on the range of the sample data, create k intervals.
3: Count the number of samples O_i, $i \in 1, 2, ..., k$ that fall in interval i.
4: Compute the area under the distribution curve in each interval.
5: Find the number of expected samples E_i for interval i.
6: Compute the Chi-squared value $\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$
7: Compare χ^2 with the Chi-square critical value at the proper degrees of freedom and significance level. If χ^2 is less than the critical value, then, the proposed distribution is a reasonable model for the data.

From the estimated probability density function we obtain the cumulative distribution function:

$$F_{11}(p) = \int_{0.9366}^{p} f_{11}(p)dp. \tag{6.12}$$

The probability density function of the minimum of the 100 random variables for epoch 1, is:

$$f_{\min}(p) = 100(1 - F_{11}(p))^{99} f_{11}(p). \tag{6.13}$$

By theorem 6.1.2:

$$\begin{aligned} \mu_{\min}(p) &= \int_{0.9366}^{0.9555} 100p(1 - F_{11}(p))^{99} f_{11}(p)dp \\ &= -\left[p(1 - F_{11}(p))^{100} \Big|_{0.9366}^{0.9555} - \int_{0.9366}^{0.9555} (1 - F_{11}(p))^{100} dp \right] \\ &= 0.93866. \end{aligned} \tag{6.14}$$

In a similar way, using the formulas obtained in theorem 6.1.2, $\sigma^2_{\min} = 8.67975 \cdot 10^{-5}$ for strategy #1.

Table 6.11: Division of range $[0.9366, 0.9555]$ using $k = 17$ intervals of equal length with actual and expected number of samples per interval with data from strategy #1.

Interval No.	Interval	Actual No. of Samples	Expected No. of Samples
1	$[0.9366, 0.9377)$	1	0.43
2	$[0.9377, 0.9388)$	0	0.37
3	$[0.9388, 0.9399)$	1	0.52
4	$[0.9399, 0.9410)$	1	0.93
5	$[0.9410, 0.9422)$	3	1.61
6	$[0.9422, 0.9433)$	3	2.59
7	$[0.9433, 0.9444)$	6	3.81
8	$[0.9444, 0.9455)$	4	5.21
9	$[0.9455, 0.9466)$	7	6.73
10	$[0.9466, 0.9477)$	6	8.26
11	$[0.9477, 0.988)$	6	9.68
12	$[0.9488, 0.9499)$	15	10.83
13	$[0.9499, 0.9510)$	18	11.52
14	$[0.9510, 0.9522)$	10	11.60
15	$[0.9522, 0.9533)$	11	10.80
16	$[0.9533, 0.9544)$	3	8.88
17	$[0.9544, 0.9555]$	5	5.57

Now, consider weight initialization strategy #3, namely w_i = gaussian_random$(0, 1)$, and the data from Table 6.3, pertaining to the probability of recognition in epoch 1. The probability of correct classification is in the interval $[0.9164, 0.9394]$ and we have $J = 100$ measurements. Using a similar method as with previous strategy, we can find a probability density function that fits the data. Table 6.12 shows the coefficients estimated for the probability density function f_{13} with the form specified by equation 6.11 where $0.9164 \leq p < 0.9408$.

From the estimated probability density function we obtain the cumulative distribution function:

$$F_{13}(p) = \int_{0.9164}^{p} f_{13}(p)dp. \tag{6.15}$$

The probability density function of the maximum of the 100 random variables for epoch 1, is:

$$f_{\max}(p) = 100(F_{13}(p))^{99} f_{13}(p). \tag{6.16}$$

By theorem 6.1.2:

$$\begin{aligned} \mu_{\max}(p) &= \int_{0.9164}^{0.9408} 100p(F_{13}(p))^{99} f_{13}(p)dp \\ &= p(F_{13}(p))^{100}\Big|_{0.9164}^{0.9408} - \int_{0.9164}^{0.9408} (F_{13}(p))^{100}dp \\ &= 0.9396. \end{aligned} \tag{6.17}$$

In a similar way, using the formulas obtained in theorem 6.1.1, $\sigma^2_{\max} = 6.10439 \cdot 10^{-7}$.

At this point, we have to determine whether $\mu_{max} + a_{max}\sigma^2_{max} < \mu_{min} - a_{min}\sigma^2_{min}$ with 95% probability, which means that the maximum of strategy #3 will never reach the minimum of strategy #1, or whether they overlap. In this case, the distributions for the maximum of strategy #3 and minimum of strategy #1 overlap ($\mu_{min} < \mu_{max}$). While it is clear that the probability of expected maximum for strategy #3 to reach the expected overall value of strategy #1 is very small, we must determine what is the probability that the former will outperform the latter.

Using a data driven approach we can see that the maximum recognition achieved by strategy #3 among the 100 networks on epoch 1 was 93.94% while the minimum recognition achieved by strategy

Table 6.12: Parameter estimation for probability density function $f_{13}(x)$ for strategy #3 in epoch 1.

Parameter	Estimated value
a_{13}	$4.617722411661846 \times 10^{9}$
b_{13}	$-1.718670140594357 \times 10^{10}$
c_{13}	$2.398649025864387 \times 10^{10}$
d_{13}	$-1.487772596119842 \times 10^{10}$
e_{13}	$3.460316849703402 \times 10^{9}$

#1 among the 100 networks on epoch 1 was 93.66%. For strategy #1 there is only 1 instance where recognition is below the maximum recognition achieved by strategy #3, while there are 6 instances in which strategy #3 outperforms strategy #1. The accuracy achieved by one strategy is independent from the other strategy, so, the probability that strategy #3 will outperform #1 is the probability that the accuracy achieved by #3 is above that of #1, given that the accuracy of #1 was the instance where it under performs. The result is $\frac{1}{100} \cdot \frac{6}{100} = \frac{6}{10000} = 0.06\%$.

This is a data-driven result; however, since the expected minimum and maximum values and the variances were computed based on the data, an analytic result using those values would yield similar probabilities.

Then, we see that the probability of strategy #3 outperforming strategy #1 is very small (0.06%). Therefore, strategy #1 is superior to strategy #3.

Our original experiment only had 20 networks per strategy and the real nature of the probability distribution could not be described very well, so, we applied our theory using uniform distribution fits. The predictions that strategy #1 was superior to #3 and that there was no significant difference among #1 and #4 were the same. We increased the number of networks to 100 to obtain a better representation of the behavior of the distributions, allowing us to fit more complex functions that represented the probability better, obtaining more accurate numerical results. For practical purposes, it seems that a reduced number of networks per strategy and a much simpler fit using uniform distributions will produce similar predictions. This last statement, however, requires further study to investigate how much can the number of networks be reduced and what other factors would alter the predictions.

6.3.5 Impact of Initialization Strategy on Network Convergence

To assess the validity of the result obtained from our quantitative method, we proceeded to train all of the 400 neural networks for 60 epochs each. The objective was to compare the results of the actual complete training for each initialization strategy and how the maximum recognition, epoch of maximum recognition, epoch-speed-of-learning and stabilization accuracy behave in relation to the initialization strategy used.

Table 6.17 summarizes the dependency of epoch of maximum recognition and the epoch-speed-of-learning with $\epsilon = 0.01$ on weight initialization strategies for four networks (one per strategy) that can be seen in Table 6.13, 6.14, 6.15 and 6.16.

We created a histogram per strategy for the number of epochs needed by each neural network to reach maximum recognition, maximum recognition accuraccy achieved, epoch-speed-of-learning and stabilization accuraccy for each neural network.

Table 6.13: Accuracy per epoch for a single neural network training for initialization strategy #1

Epoch No.	Probability (%) of Correct Classification	Epoch No.	Probability (%) of Correct Classification
1	95.24	31	96.99
2	95.51	32	96.97
3	96.08	33	97.04
4	95.48	34	97
5	95.99	35	96.96
6	96.33	36	96.98
7	96.42	37	96.99
8	96.04	38	97.02
9	96.77	39	96.99
10	96.73	40	96.98
11	96.96	41	96.99
12	96.84	42	97.01
13	96.98	43	96.99
14	96.6	44	97
15	96.97	45	97
16	96.85	46	97
17	96.98	47	96.99
18	96.99	48	96.99
19	96.98	49	96.99
20	97	50	96.99
21	96.97	51	96.99
22	97.05	52	97
23	97.06	53	97
24	97.04	54	97
25	96.98	55	97
26	97.02	56	97
27	97	57	97
28	96.97	58	97
29	97.04	59	97
30	97.01	60	97

Table 6.14: Accuracy per epoch for a single neural network training for initialization strategy #2

Epoch No.	Probability (%) of Correct Classification	Epoch No.	Probability (%) of Correct Classification
1	94.6	31	96.95
2	95.6	32	96.85
3	95.85	33	96.93
4	95.94	34	96.91
5	96.07	35	96.92
6	96.32	36	96.92
7	96.46	37	96.93
8	96.34	38	96.92
9	96.52	39	96.93
10	96.56	40	96.93
11	96.48	41	96.89
12	96.76	42	96.95
13	96.87	43	96.92
14	96.85	44	96.93
15	97.01	45	96.92
16	96.92	46	96.93
17	96.86	47	96.94
18	96.87	48	96.93
19	97.05	49	96.93
20	96.97	50	96.93
21	97.01	51	96.92
22	96.93	52	96.92
23	96.89	53	96.92
24	96.96	54	96.92
25	96.94	55	96.92
26	96.94	56	96.92
27	96.96	57	96.92
28	96.94	58	96.92
29	96.93	59	96.92
30	96.9	60	96.92

Table 6.15: Accuracy per epoch for a single neural network training for initialization strategy #3

Epoch No.	Probability (%) of Correct Classification	Epoch No.	Probability (%) of Correct Classification
1	93.08	31	96.22
2	94.17	32	96.24
3	94.04	33	96.18
4	95.09	34	96.22
5	95.11	35	96.2
6	95.5	36	96.24
7	95.18	37	96.2
8	95.49	38	96.24
9	95.31	39	96.21
10	95.91	40	96.21
11	95.75	41	96.23
12	96.17	42	96.21
13	95.93	43	96.21
14	96.05	44	96.23
15	95.97	45	96.23
16	96.13	46	96.23
17	96.04	47	96.24
18	96.14	48	96.23
19	96.23	49	96.22
20	96.11	50	96.23
21	96.12	51	96.22
22	96.15	52	96.22
23	96.2	53	96.22
24	96.16	54	96.22
25	96.22	55	96.23
26	96.13	56	96.23
27	96.24	57	96.22
28	96.19	58	96.22
29	96.19	59	96.23
30	96.21	60	96.23

Table 6.16: Accuracy per epoch for a single neural network training for initialization strategy #4

Epoch No.	Probability (%) of Correct Classification	Epoch No.	Probability (%) of Correct Classification
1	94.62	31	97.01
2	95.75	32	97.02
3	95.73	33	97.04
4	96.28	34	97.02
5	96.23	35	96.99
6	96.13	36	97.01
7	96.38	37	97.03
8	96.53	38	97.04
9	96.52	39	97.04
10	96.92	40	97.03
11	96.86	41	97.03
12	96.91	42	97.03
13	96.93	43	97.03
14	96.88	44	97.04
15	96.91	45	97.04
16	97.01	46	97.02
17	97.01	47	97.02
18	97.01	48	97.02
19	97.07	49	97.02
20	96.95	50	97.02
21	97.02	51	97.02
22	96.99	52	97.02
23	97.05	53	97.02
24	96.99	54	97.02
25	97.07	55	97.02
26	97.01	56	97.02
27	97.07	57	97.02
28	97.04	58	97.02
29	97.03	59	97.02
30	97.03	60	97.02

Table 6.17: Maximum and stabilization accuracy per strategy based on Table 6.13, 6.14, 6.15 and 6.16.

Strategy No.	1	2	3	4
Maximum %	97.06	97.05	96.24	97.07
Max. Epoch No.	23	19	27	19
Stabilization %	97.00	96.92	96.22	97.02
Epoch-Speed-of-Learning	43	48	48	46

With these histograms we estimated best model probability functions using the Chi-Square test as explained by the procedure in algorithm 3. The Gaussian distribution proved to be the best fit for all except for the number of epochs needed for the neural network to obtain its maximum value for the first time. In that case, the best fit as determined by the Chi-Squared test, was found to be a Gamma distribution of the form:

$$f(x) = \frac{x^{\alpha-1}e^{-x/\beta}}{\Gamma(\alpha)\beta^{\alpha}} \tag{6.18}$$

where $\Gamma(x)$ is the Gamma function, α is the shape parameter and β is the scale parameter of the Gamma distribution.

The histograms and the best fits are shown in Figure 6.1, 6.2, 6.3 and 6.4.

Using the data obtained for the epoch of maximum recognition and the epoch-speed-of-learning, we performed an analysis of variance to determine whether the results are significantly different or not. We formulated four null hypotheses: $H_0^1 : \mu_{1\,\max} = \mu_{2\,\max} = \mu_{3\,\max} = \mu_{4\,\max}$ where $\mu_{i\,\max}$ is the mean of the maximum recognition attained by all networks using the i-th initialization strategy; $H_0^2 : \mu_{1\text{resl}} = \mu_{2\text{resl}} = \mu_{3\text{resl}} = \mu_{4\text{resl}}$ where $\mu_{i\text{resl}}$ is the mean of the recognition for the epoch-speed-of-learning attained by all networks using the i-th initialization strategy; $H_0^3 : \mu_{1\text{max\_epoch}} = \mu_{2\text{max\_epoch}} = \mu_{3\text{max\_epoch}} = \mu_{4\text{max\_epoch}}$ where $\mu_{i\text{max\_epoch}}$ is the mean epoch where the maximum recognition was attained by all networks using the i-th initialization strategy; $H_0^4 : \mu_{1\text{esl}} = \mu_{2\text{esl}} = \mu_{3\text{esl}} = \mu_{4\text{esl}}$ where $\mu_{i\text{esl}}$ is the mean of the epoch-speed-of-learning for all networks using the i-th initialization strategy.

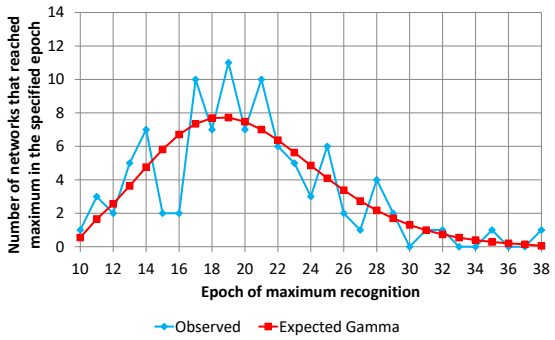
Table 6.18, 6.19, 6.21 and 6.20 show the results of the analysis of variance to test the four null hypotheses.

The results indicate that there is a statistically significant difference with 95% confidence among the maximum recogition accuracy that the strategies can achieve, thus rejecting the null hypothesis H_0^1. A second analysis of variance performed to test the null hypothesis $H_0'^1 : \mu_{1\,\max} = \mu_{2\,\max} = \mu_{4\,\max}$ shows that there is some significant difference among strategies #1, #2 and #4, but not as marked as with strategy #3 (an analysis of variance between strategies #1 and #4 showed that

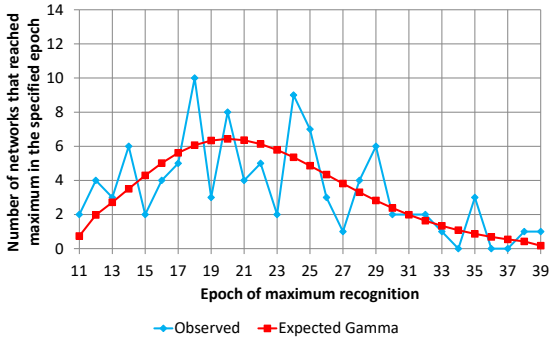
there is no statistically significant difference between them) therefore, strategy #3 is the outlier.

Finding the 95% confidence interval of the means using the t distribution with 99 degrees of freedom we see that $96.94 \leq \mu_{1\,\max} \leq 96.98$, $96.88 \leq \mu_{2\,\max} \leq 96.94$, $96.48 \leq \mu_{3\,\max} \leq 96.54$ and $96.94 \leq \mu_{4\,\max} \leq 96.98$. Because the expected maximum recognition accuracy that strategy #3 can achieve is smaller than all other strategies, we can conclude that strategy #3 is inferior to the others as we had already predicted using our quantitative measure.

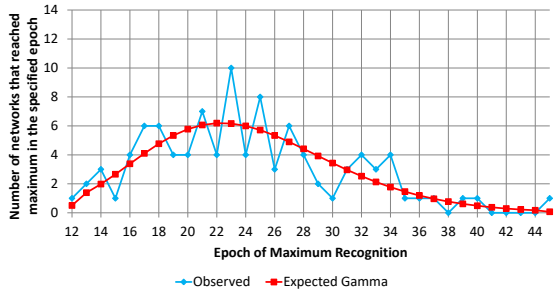
A similar analysis applying analysis of variance rejects the null hypothesis H_0^3. Further study shows again that strategy #3 is the outlier and inferior to the others as predicted with 95% confidence interval of the means using the t distribution with 99 degrees of freedom: $18.96 \leq \mu_{1\,\max\_epoch} \leq 21.06$, $20.64 \leq \mu_{2\,\max\_epoch} \leq 23.20$, $22.82 \leq \mu_{3\,\max\_epoch} \leq 25.46$ and $18.64 \leq \mu_{4\,\max\_epoch} \leq 21.22$. We can see how, not only strategy #1 can reach a better maximum classification accuracy than strategy #3, but it will do so earlier in training than strategy #3.



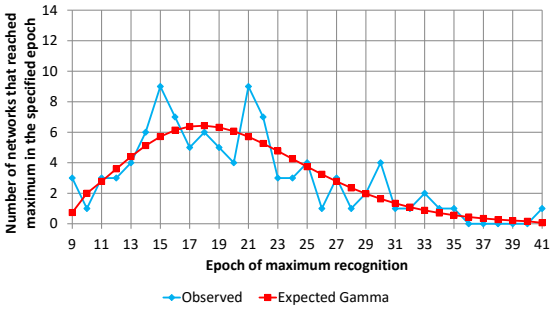
(a) Parameters $\alpha = 14.25$ and $\beta = 1.40$



(b) Parameters $\alpha = 11.63$ and $\beta = 1.89$



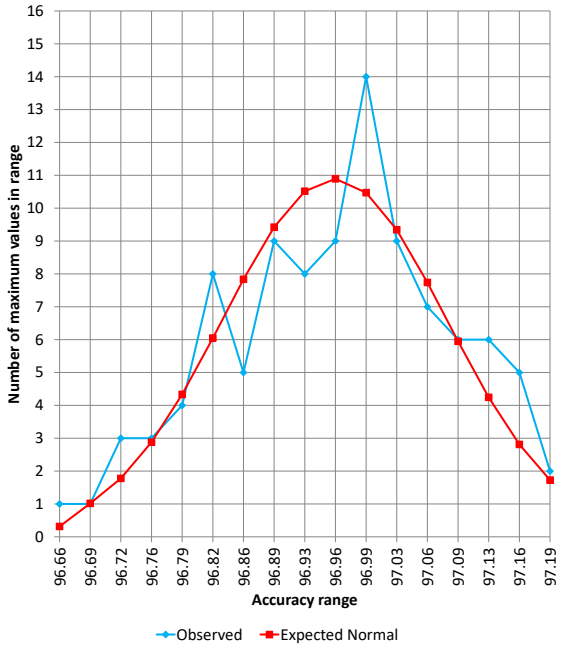
(c) Parameters $\alpha = 13.21$ and $\beta = 1.83$



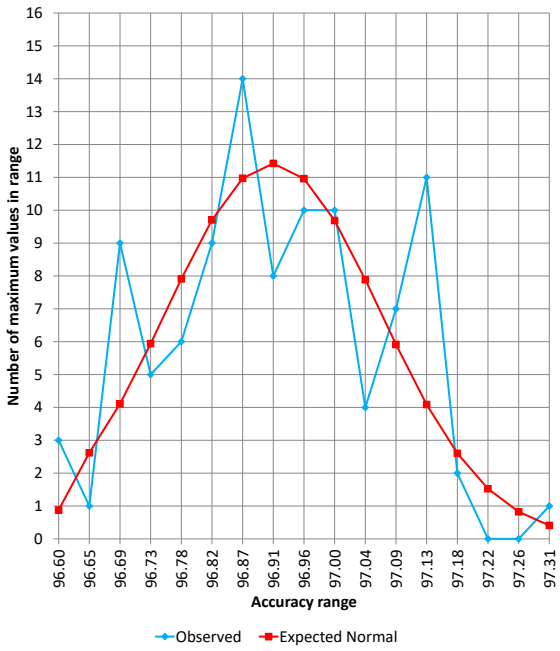
(d) Parameters $\alpha = 9.44$ and $\beta = 2.11$

Figure 6.1: Epoch of maximum recognition histograms and Gamma distribution fit for initialization strategies.

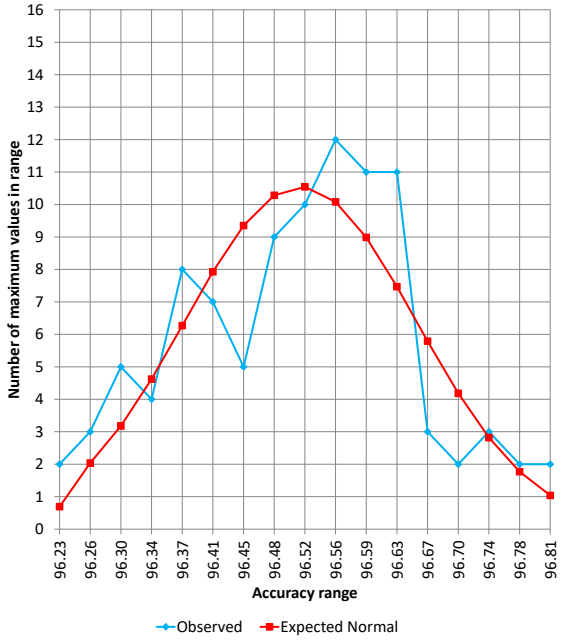
(a) Histogram for strategy #1 with $\mu = 20.01$ and $\sigma^2 = 28.09$; (b) histogram for strategy #2 with $\mu = 21.92$ and $\sigma^2 = 41.33$; (c) histogram for strategy #3 with $\mu = 24.14$ and $\sigma^2 = 44.1$; (d) histogram for strategy #4 with $\mu = 19.93$ and $\sigma^2 = 42.07$.



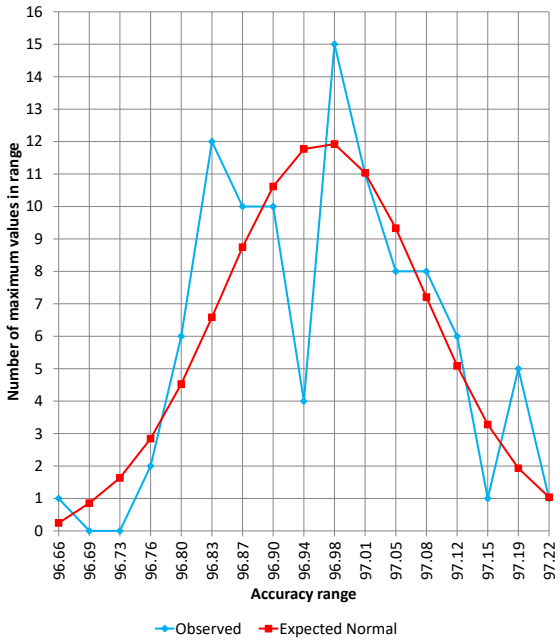
(a) Parameters $\mu = 96.96$ and $\sigma^2 = 0.015$



(b) Parameters $\mu = 96.91$ and $\sigma^2 = 0.0236$



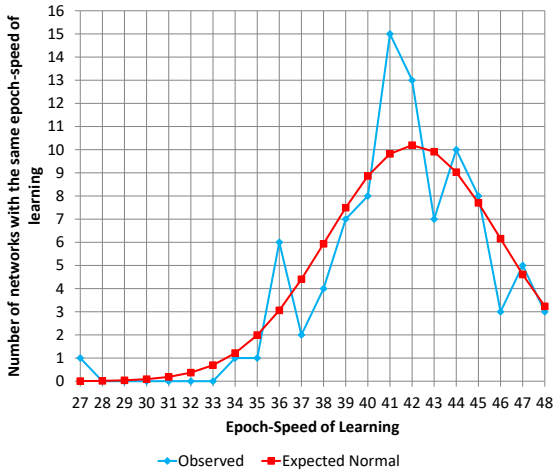
(c) Parameters $\mu = 96.51$ and $\sigma^2 = 0.0189$



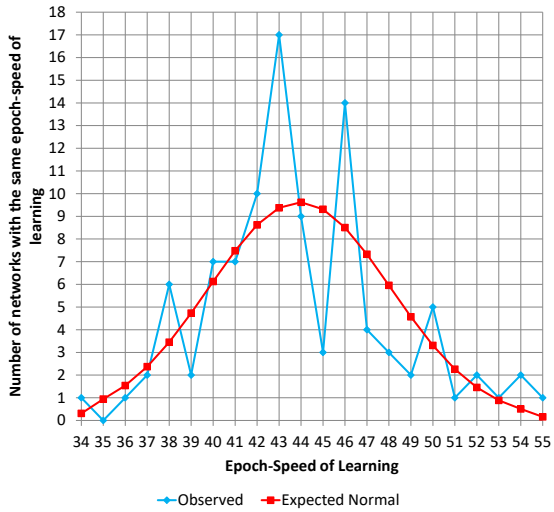
(d) Parameters $\mu = 96.96$ and $\sigma^2 = 0.0137$

Figure 6.2: Maximum recognition accuracy histograms and Gaussian distribution fit for initialization strategies.

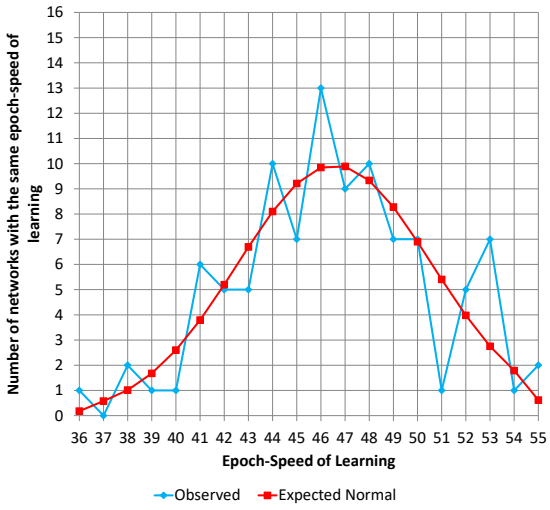
(a) Histogram for strategy #1; (b) histogram for strategy #2; (c) histogram for strategy #3; (d) histogram for strategy #4.



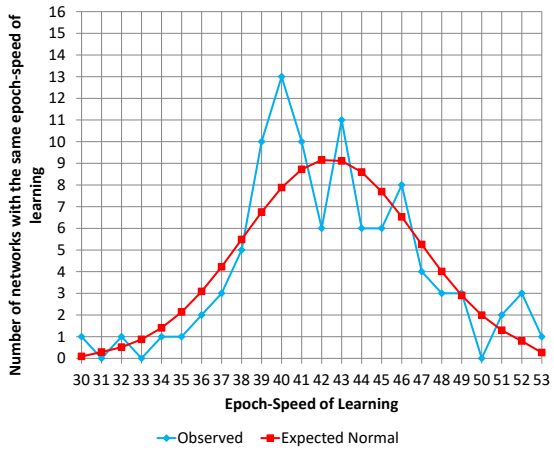
(a) Parameters $\mu = 42.07$ and $\sigma^2 = 15.23$



(b) Parameters $\mu = 43.94$ and $\sigma^2 = 17.12$



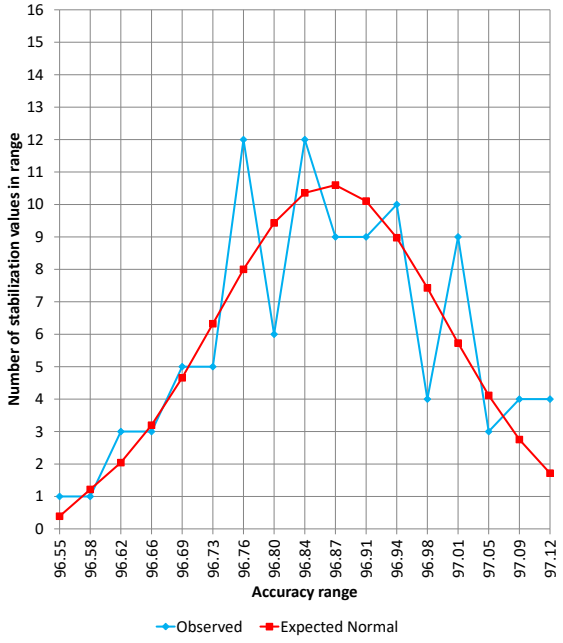
(c) Parameters $\mu = 46.57$ and $\sigma^2 = 16.0$



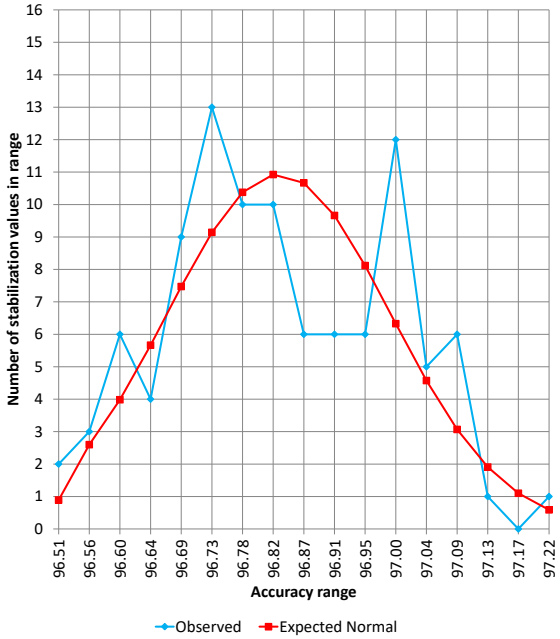
(d) Parameters $\mu = 42.41$ and $\sigma^2 = 18.74$

Figure 6.3: Epoch-Speed-of-Learning histograms and Gaussian distribution fit for initialization strategies.

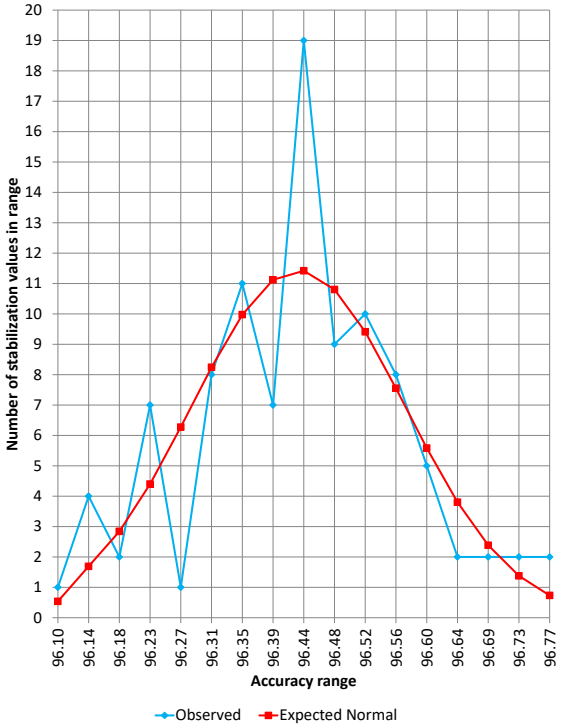
(a) Histogram for strategy #1; (b) histogram for strategy #2; (c) histogram for strategy #3; (d) histogram for strategy #4.



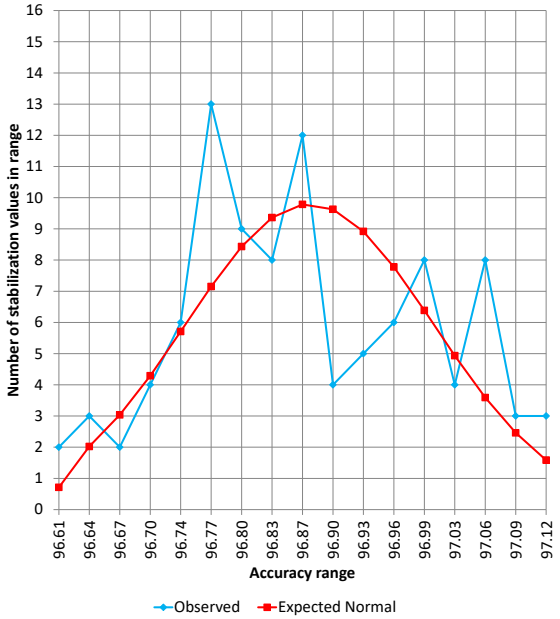
(a) Parameters $\mu = 96.86$ and $\sigma^2 = 0.0181$



(b) Parameters $\mu = 96.83$ and $\sigma^2 = 0.0257$



(c) Parameters $\mu = 96.43$ and $\sigma^2 = 0.021$



(d) Parameters $\mu = 96.87$ and $\sigma^2 = 0.0173$

Figure 6.4: Epoch-Speed-of-Learning recognition accuracy histograms and Gaussian distribution fit for initialization strategies.

(a) Histogram for strategy #1; (b) histogram for strategy #2; (c) histogram for strategy #3; (d) histogram for strategy #4.

Table 6.18: ANOVA results for testing the null hypothesis $H_0^1 : \mu_{1\,\max} = \mu_{2\,\max} = \mu_{3\,\max} = \mu_{4\,\max}$.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	13.957	3	4.652
Within-treatments	7.042	396	0.0178
Total	20.999	399	$F = 261.63$

Table 6.19: ANOVA results for testing the null hypothesis $H_0^3 : \mu_{1\text{max\_epoch}} = \mu_{2\text{max\_epoch}} = \mu_{3\text{max\_epoch}} = \mu_{4\text{max\_epoch}}$.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	1183.1	3	394.37
Within-treatments	15558.9	396	39.29
Total	16742	399	$F = 10.04$

Table 6.20: ANOVA results for testing the null hypothesis $H_0^2 : \mu_{1\text{resl}} = \mu_{2\text{resl}} = \mu_{3\text{resl}} = \mu_{4\text{resl}}$.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	13.831	3	4.61
Within-treatments	8.134	396	0.0205
Total	21.965	399	$F = 224.45$

Table 6.21: ANOVA results for testing the null hypothesis $H_0^4 : \mu_{1\text{esl}} = \mu_{2\text{esl}} = \mu_{3\text{esl}} = \mu_{4\text{esl}}$.

Source	Sum of Squares	Degrees of Freedom	MSE
Between-treatments	1260.64	3	420.22
Within-treatments	6708.85	396	16.942
Total	7969.49	399	$F = 24.8$

Finally, looking at the epoch-speed-of-learning, analysis of variance rejects the null hypothesis H_0^4 indicating that there is significant difference among the epoch-speed-of-learning for each strategy. A second analysis of variance performed to test the null hypothesis $H_0'^4 : \mu_{1\text{esl}} = \mu_{2\text{esl}} = \mu_{4\text{esl}}$ shows that there is some significant difference among strategies #1, #2 and #4, but not as marked as with strategy #3. Another analysis of variance between strategies #1 and #4 showed that there is no statistically significant difference between them. Therefore, once more, strategy #3 is the outlier.

Finding the 95% confidence interval of the means using the t distribution with 99 degrees of freedom we see that $41.30 \leq \mu_{1\text{esl}} \leq 42.84$, $43.12 \leq \mu_{2\text{esl}} \leq 44.76$, $45.78 \leq \mu_{3\text{esl}} \leq 47.36$ and

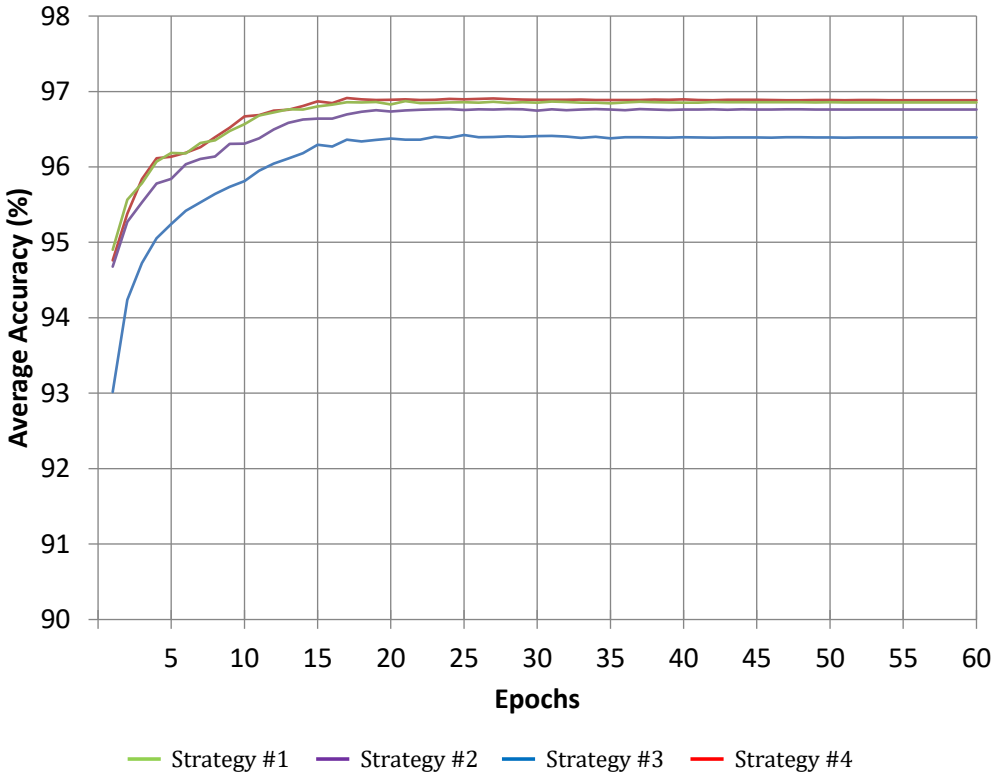


Figure 6.5: Learning curve for all 4 initialization strategies.
Notice how strategies #1 and #4 are close together reaching highest stabilization accuracy around the same epoch-speed-of-learning. They are followed closely by strategy #2 and strategy #3 is a distant fourth place as predicted.

$41.55 \leq \mu_{4\text{esl}} \leq 43.27$. Because the expected epoch-speed-of-learning for strategy #3 is smaller than all other strategies, we can conclude that strategy #3 is inferior to strategy #1. Furthermore, using the same technique we reject the null hypothesis H_0^2 we find the confidence interval for the means to be $96.83 \leq \mu_{1\text{resl}} \leq 96.89$, $96.80 \leq \mu_{2\text{resl}} \leq 96.86$, $96.40 \leq \mu_{3\text{resl}} \leq 96.46$ and $96.84 \leq \mu_{4\text{resl}} \leq 96.90$.

Again, notice the correlation between the stable recognition that a network can achieve and at what epoch the stabilization occurs with the weight initialization strategy applied before training. A superior initialization strategy will also help the network stabilize earlier and achieve better stable recognition.

Finally, Figure 6.5 summarizes the average training accuracy for each strategy. We can see how strategies #1 and #4 each reach similar maxima, peaking around the same epoch and achieving similar stabilization accuracy around the same epoch-speed-of-learning. Strategy #2 follows closely with the second best accuracy and epoch-speed-of-learning where strategy #3 is a distant fourth place as predicted.

6.4 Results

In this research we developed a theory that allows for the quantitative comparison between initialization strategies for neural networks and assess whether the fluctuations of the convergence rate and accuraccy are due to the random nature of the initialization strategies or due to the fact of one strategy being superior to the other. The method compares parameter initialization strategies based on the classification results of the first epoch instead of having to proceed with the time consuming training and compare the results after training completes. We applied our theory to four different initialization strategies, each applied to 100 neural networks with the same architechture, trained to recognize the MNIST dataset of handwritten digits using the backpropagation algorithm. Our theory predicted that one of the strategies was inferior to the others from the quantitative results. We proceeded to train every network over 60 epochs to assess whether the prediction was correct. The results of actual training showed that the prediction was right and not only superior initialization strategies help a neural network to minimize the classification error, but the epoch where a network reaches maximum recognition for the first time and the epoch where the network stabilizes and stops learning (epoch-speed-of-learning) appears faster in training, saving precious training time with smaller epoch numbers for early stopping.

Appendix A

Standard HE and Enhanced IWHE Implementation in C#

This implementation is .NET 4 compliant and was compiled and tested using Microsoft Visual Studio 2015 and Mono 4.8. The following listing makes use of our proprietary C++/CLI .NET 4.0 compliant library, compatible with .NET Bitmap and Image objects, that allows fast and parallel access and modification of image pixel data to the color channel granularity as well as including some tools to create histograms from loaded images. Our custom image processing library is not included in this appendix.

As a disclaimer, the following code, as of this writing, is offered as is for reference purposes, even though it is fully functional. It is constantly tested, maintained, optimized and debugged. The use of this code is at the user's own willingness and the creator of this code shall not be held responsible for any liability resulting from any form of use of this code.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using KImageProcessing; // Custom bitmap library written in C++/CLI
                        // that allows fast access to image pixel data

namespace HistogramEq
{
  public static class HE
  {
    private static void swap<T>(ref T _A, ref T _B)
    {
      T _C = _A;
```

```
    _A = _B;
    _B = _C;
  }

  /// <summary>
  /// Applies median filter to the specified grayscale bitmap.
  /// </summary>
  private static void medFilter(out GrayScaleMap result, GrayScaleMap
      ↪ bmGray)
  {
    GrayScaleMap retval = new GrayScaleMap(bmGray.Width, bmGray.Height
        ↪ );


    for (int y = 0; y < bmGray.Height; y++)
      for (int x = 0; x < bmGray.Width; x++)
      {
        int a, b, c, d;
        int delta;

        delta = (y - 1 < 0 ? 1 : -1);
        a = bmGray.Pixels[x, y + delta];

        delta = (y + 1 >= bmGray.Height ? -1 : 1);
        b = bmGray.Pixels[x, y + delta];

        delta = (x - 1 < 0 ? 1 : -1);
        c = bmGray.Pixels[x + delta, y];

        delta = (x + 1 >= bmGray.Width ? -1 : 1);
        d = bmGray.Pixels[x + delta, y];

        if (a > c)
          swap(ref a, ref c);
        if (b > d)
          swap(ref b, ref d);
        if (a > b)
          swap(ref a, ref b);
        if (c > d)
          swap(ref c, ref d);

        retval.Pixels[x, y] = (b + c) >> 1;
      }

    result = retval;
  }

  /// <summary>
  /// Performs enhanced IWHE with average intensity for bucket spread
  /// after median filter denoising on the specified grayscale bitmap.
  /// </summary>
```

```
public static void ExEqualize_Ave(out GrayScaleMap result, out
    ↪ Histogram resultH,
  GrayScaleMap bmGray, Histogram.ColorComponent hist, int ilimitY)
{
  Dictionary<int, System.Drawing.Point> dictRegions;

  const double inv_9 = 1.0 / 9.0;

  GrayScaleMap bm1;
  Histogram h1;

  medFilter(out bm1, bmGray);
  h1 = new Histogram(bm1);

  GrayScaleMap bmEq;
  Histogram hEq;
  Histogram.ColorComponent histEq;

  IWEqualize(out bmEq, out hEq, bm1, h1.Intensity, ilimitY);
  histEq = hEq.Intensity;

  if (h1 != null)
    h1.Dispose();
  if (bm1 != null)
    bm1.Dispose();

  dictRegions = new Dictionary<int, System.Drawing.Point>();
  int prev = 0;
  int last = -1;
  for (int i = 0; i < histEq.Count; i++)
    if (histEq[i] > 0)
    {
      if (last >= 0)
      {
        int next = ((last + i) >> 1);
        if (prev > 2)
          prev -= 2;
        if (next < 254)
          next += 2;
        dictRegions.Add(last, new System.Drawing.Point(prev, next));
        prev = ((last + i) >> 1) + 1;
      }

      last = i;
    }
  dictRegions.Add(last, new System.Drawing.Point(prev, histEq.Count
    ↪ - 1));

  GrayScaleMap retval = new GrayScaleMap(bmEq.Width, bmEq.Height);

  for (int y = 0; y < bmEq.Height; y++)
```

```
      for (int x = 0; x < bmEq.Width; x++)
      {
        int i_xy;
        int min = 256;
        int max = -1;
        double ave = 0.0;
        for (int dy = -1; dy <= 1; dy++)
          for (int dx = -1; dx <= 1; dx++)
          {
            int _dx = dx;
            int _dy = dy;

            if (x + _dx < 0 || x + _dx >= bmEq.Width)
              _dx = -_dx;
            if (y + _dy < 0 || y + _dy >= bmEq.Height)
              _dy = -_dy;

            i_xy = bmEq.Pixels[x + _dx, y + _dy];
            if (i_xy < min)
              min = i_xy;
            if (i_xy > max)
              max = i_xy;
            ave += (i_xy * inv_9);
          }

        i_xy = bmEq.Pixels[x, y];
        if (max == min)
          retval.Pixels[x, y] = i_xy;
        else
        {
          System.Drawing.Point range = new System.Drawing.Point();
          if (!dictRegions.ContainsKey(i_xy))
          {
            foreach (System.Drawing.Point pt in dictRegions.Values)
              if (pt.Y >= i_xy)
              {
                range = pt;
                break;
              }
          }
          else
            range = dictRegions[i_xy];
          retval.Pixels[x, y] = (int)((range.Y - range.X) * (ave - min
            ↪ ) / (max - min) + range.X);
        }
      }

    if (hEq != null)
      hEq.Dispose();
    if (bmEq != null)
      bmEq.Dispose();
```

```
    result = retval;
    resultH = new Histogram(retval);
  }

  /// <summary>
  /// Performs enhanced IWHE on the specified grayscale
  /// bitmap.
  /// </summary>
  public static void ExEqualize(out GrayScaleMap result, out Histogram
      ↪ resultH,
    GrayScaleMap bmGray, Histogram.ColorComponent hist, int ilimitY)
  {
    ExEqualize_Ave(out result, out resultH, bmGray, hist, ilimitY);
  }

  /// <summary>
  /// Performs IWHE on the specified grayscale bitmap.
  /// </summary>
  public static void IWEqualize(out GrayScaleMap result, out Histogram
      ↪ resultH, GrayScaleMap bmGray,
    Histogram.ColorComponent hist, int iMaxIntensity, int ilimitY)
  {
    double limitY = (ilimitY * bmGray.Length) / 100.0;

    int[] CH;
    double d;

    CH = new int[256];
    d = iMaxIntensity / (double)(bmGray.Length);
    GrayScaleMap retval = new GrayScaleMap(bmGray.Width, bmGray.Height
        ↪ );

    int acc = 0;
    int imax = 0;
    while (imax < 255 && acc < limitY)
      acc += hist[imax++];

    if (imax > 0)
      d /= imax;

    CH[0] = hist[0];
    for (int i = 1; i < hist.Count; i++)
      CH[i] = CH[i - 1] + hist[i];


    for (int i = 0; i < bmGray.Length; i++)
    {
      int clr = (int)(CH[bmGray.Pixels[i]] * d * bmGray.Pixels[i]);
      if (clr > iMaxIntensity)
        clr = iMaxIntensity;
```

```
      else if (clr < 0)
        clr = 0;
      retval.Pixels[i] = clr;
    }

    result = retval;
    resultH = new Histogram(retval);
  }

  /// <summary>
  /// Performs IWHE on the specified grayscale bitmap.
  /// </summary>
  public static void IWEqualize(out GrayScaleMap result, out Histogram
      ↪ resultH, GrayScaleMap bmGray,
    Histogram.ColorComponent hist, int ilimitY)
  {
    IWEqualize(out result, out resultH, bmGray, hist, 255, ilimitY);
  }

  /// <summary>
  /// Performs standard histogram equalization on the specified
  /// grayscale bitmap.
  /// </summary>
  public static void Equalize(out GrayScaleMap result, out Histogram
      ↪ resultH, GrayScaleMap bmGray,
    Histogram.ColorComponent hist, int lowerBound, int upperBound, int
        ↪ minIntensity, int maxIntensity)
  {
    GrayScaleMap retval = new GrayScaleMap(bmGray.Width, bmGray.Height
        ↪ );
    int[] CH = new int[hist.Count];

    double d = maxIntensity / (double)(bmGray.Length);

    CH[lowerBound] = hist[lowerBound];
    for (int i = lowerBound + 1; i <= upperBound; i++)
      CH[i] = CH[i - 1] + hist[i];

    Parallel.For(0, bmGray.Length, (i) =>
    {
      int value = bmGray.Pixels[i];
      if (value < lowerBound)
        retval.Pixels[i] = minIntensity;
      else if (value > upperBound)
        retval.Pixels[i] = maxIntensity;
      else
      {
        int respix = (int)(CH[value] * d);
        if (respix < minIntensity)
          respix = minIntensity;
        else if (respix > maxIntensity)
```

```
                respix = maxIntensity;
            retval.Pixels[i] = respix;
          }
        });

        result = retval;
        resultH = new Histogram(retval);
      }
    }
}
```

Appendix B

Feedforward Neural Network Library Implementation in C++

The following code is C++11 compliant. It requires Intel Math Kernel Library and an OpenCL implementation installed (it was tested using Intel MKL 2016-2017 and AMD OpenCL 1.1 implementation) on Windows 10 and Linux operating systems, compiled with Intel's C++ compiler in Intel Parallel Studio XE. Code using this library was run and tested successfully on machine configurations containing Intel Core i5 CPU with nVidia GTX 760 GPU; Intel Core i7 CPU with AMD Radeon R9 200 Series GPU; and (National Supercomputing Institute, Cherry-Creek cluster) Intel Xeon E5-2697v2 CPU with Intel Xeon Phi 7120P co-processors.

File *kffann.h* is the main entry point of the library. Including this file will enable the whole library to the client. Check this file for specifics, such as what macros to define in order to enable certain features like the trainer.

The library supports multilayer perceptrons and convolutional neural networks with an implementation of back propagation with stochastic gradient descent for training featuring dynamic learning rate, momentum, regularization and a variety of activation and cost functions. It makes use of computational optimization technologies such as OpenMP (version 2.0 for compatibility with Microsoft C++ compilers) for multithreaded operations, Intel MKL for algebra operations and OpenCL for co-processor offloading.

As a disclaimer, the following code, as of this writing, is offered as is for reference purposes, even though it is fully functional. It is constantly tested, maintained, optimized and debugged. The use of this code is at the user's own willingness and the creator of this code shall not be held responsible for any liability resulting from any form of use of this code.

B.1 File: kconvfeature.h

```
#ifndef K_CONVOLUTIONA_FEATURE_H
#define K_CONVOLUTIONA_FEATURE_H

#include <cstddef>

#include "kconvfilter.h"
#include "ksafevector.h"
#include "knnkeys.h"

#include "ktypeutils.h"

namespace NeuralNetwork
{
  namespace Convolutional
  {
    class Feature32
    {
    public:

      static const std::size_t OUTPUT_DEPTH = 1;

      /// <summary>
      /// Depth of the output generated by this feature's convolution.
      /// </summary>
      std::size_t getOutputDepth() const
      {
        return OUTPUT_DEPTH;
      }

      /// <summary>
      /// Width of the output generated by this feature's convolution.
      /// </summary>
      std::size_t getOutputWidth() const
      {
        return m_uiOutputWidth;
      }

      /// <summary>
      /// Height of the output generated by this feature's convolution.
      /// </summary>
      std::size_t getOutputHeight() const
      {
        return m_uiOutputHeight;
      }

      /// <summary>
      /// Step at which the filter will be translated over the input
        ↪ during convolution.
```

```
    /// </summary>
    std::size_t getStep() const
    {
      return m_uiStep;
    }


    const Filter32 *getFilter() const
    {
      return &m_filter;
    }

    Filter32 *getFilter()
    {
      return &m_filter;
    }

    std::size_t getLayerIndex() const
    {
      return m_uiLayerIndex;
    }

    std::size_t getStartWeight() const
    {
      return m_filter.getStartWeight();
    }


    Feature32(std::size_t step, const Filter32 &src,
        KUtilities::atomic_vector<float> &biases, std::size_t
            ↪ uiBiasIndex,
        KUtilities::atomic_vector<float> &weights, std::size_t
            ↪ uiStartWeight,
        std::size_t uiInputWidth, std::size_t uiInputHeight, std::
            ↪ size_t uiLayerIndex,
        const FeatureKey &key) :
      m_filter(src, biases, uiBiasIndex, weights, uiStartWeight),
      m_uiStep(step),
      m_uiLayerIndex(uiLayerIndex)
    {
      m_uiOutputWidth = (uiInputWidth - (m_filter.getWidth() & (~1)) +
          ↪  step
        - (m_filter.getWidth() & 1))
        / step;
      m_uiOutputHeight = (uiInputHeight - (m_filter.getHeight() & (~1)
          ↪ ) + step
        - (m_filter.getHeight() & 1))
        / step;
    }

  private:
```

```
/// <summary>
/// Implements the Filter32 class. Used for the internal
    ↪ representation,
/// sharing parameters, of the feature filter.
/// </summary>
class ReferenceFilter32 : public Filter32
{
public:
  ReferenceFilter32(const Filter32 &src,
      KUtilities::atomic_vector<float> &biases, std::size_t
          ↪ uiBiasIndex,
      KUtilities::atomic_vector<float> &weights, std::size_t
          ↪ uiStartWeight) :
    Filter32(src),
    m_biases(biases),
    m_uiBiasIndex(uiBiasIndex),
    m_weights(weights),
    m_uiStartWeight(uiStartWeight),
    m_uiNumWeights(src.getFilterElementsCount())
  {
    setBias(src.getBias());
    setFilterElements(src.getFilterElements());
  }

  float getBias() const override
  {
    return m_biases[m_uiBiasIndex];
  }

  void setBias(float value) override
  {
    m_biases.store(m_uiBiasIndex, value);
  }

  std::size_t getFilterElementsCount() const override
  {
    return m_uiNumWeights;
  }

  const float *getFilterElements() const override
  {
    return m_weights.data() + m_uiStartWeight;
  }

  void setFilterElements(const float *arr) override
  {
    #pragma omp parallel for
    for (int i = 0; i < (int)getFilterElementsCount(); i++)
      setFilterElement(i, arr[i]);
  }
```

```
        void setFilterElement(std::size_t index, float value) override
        {
          m_weights.store(m_uiStartWeight + index, value);
        }

        std::size_t getStartWeight() const
        {
          return m_uiStartWeight;
        }
      private:
        KUtilities::atomic_vector<float> &m_biases;
        std::size_t m_uiBiasIndex;
        KUtilities::atomic_vector<float> &m_weights;
        std::size_t m_uiStartWeight;
        std::size_t m_uiNumWeights;
      };


      ReferenceFilter32 m_filter;
      std::size_t m_uiOutputWidth;
      std::size_t m_uiOutputHeight;
      std::size_t m_uiStep;

      std::size_t m_uiLayerIndex;
    };


    typedef Feature32 Feature;
  }
}

#endif
```

B.2 File: kconvfilter.h

```
#ifndef K_CONVOLUTIONA_FILTER_H
#define K_CONVOLUTIONA_FILTER_H

#include <cstddef>
#include <cstring>
#include <vector>
#include <stdexcept>
#include <utility>
#include <istream>
#include <ostream>
```

```
#include <memory>

#include "knnkeys.h"

#include "ktypeutils.h"

namespace NeuralNetwork
{
  namespace Convolutional
  {
    /// <summary>
    /// Base class that represents a convolutional filter.
    /// </summary>
    class Filter32
    {
    public:

      typedef std::size_t depth_index_type;

      /// <summary>
      /// Width of the filter.
      /// </summary>
      std::size_t getWidth() const
      {
        return m_uiWidth;
      }

      /// <summary>
      /// Height of the filter.
      /// </summary>
      std::size_t getHeight() const
      {
        return m_uiHeight;
      }

      /// <summary>
      /// Retrieves the bias of the filter.
      /// </summary>
      virtual float getBias() const = 0;

      /// <summary>
      /// Sets the bias of the filter.
      /// </summary>
      virtual void setBias(float value) = 0;

      Filter32()
      { }

      /// <summary>
      /// Number of elements in the array returned by getDepthIndices().
      /// </summary>
```

```
        std::size_t getDepthCount() const
        {
          return m_arrDepthIndices.size();
        }

        /// <summary>
        /// Collection of indexes for each depth (channel) on which this
            ↪ filter operates.
        /// </summary>
        const depth_index_type *getDepthIndices() const
        {
          return m_arrDepthIndices.data();
        }

        /// <summary>
        /// Number of filter elements.
        /// </summary>
        virtual std::size_t getFilterElementsCount() const = 0;
        /// <summary>
        /// Filter32 elements organized in a one-dimensional array.
        /// </summary>
        virtual const float *getFilterElements() const = 0;
        /// <summary>
        /// Sets the values for all filter elements organized in
        /// a one-dimensional array.
        /// </summary>
        virtual void setFilterElements(const float *arr) = 0;

        /// <summary>
        /// Sets the value of filter element at specified index in filter.
        /// </summary>
        virtual void setFilterElement(std::size_t index, float value) = 0;

        /// <summary>
        /// Sets the value of filter element at position (x, y, z) in
            ↪ filter.
        /// </summary>
        void setFilterElement(std::size_t x, std::size_t y, std::size_t z,
            ↪  float value)
        {
          setFilterElement(getFilterElementIndex(x, y, z), value);
        }

        /// <summary>
        /// Gets the value of filter element at position (x, y, z) in
            ↪ filter.
        /// </summary>
        float getFilterElement(std::size_t x, std::size_t y, std::size_t z
            ↪ ) const
        {
          getFilterElements()[getFilterElementIndex(x, y, z)];
```

```
    }

    /// <summary>
    /// Gets the index of filter element at position (x, y, z) in
    /// one-dimensional array returned by getFilterElements().
    /// </summary>
    std::size_t getFilterElementIndex(std::size_t x, std::size_t y,
        ↪ std::size_t z) const
    {
      return (z * getHeight() + y) * getWidth() + x;
    }

    Filter32(std::size_t width, std::size_t height,
      const Filter32::depth_index_type *arrDepthIndices, std::size_t
          ↪ uiDepthCount)
    {
      if (!arrDepthIndices)
        throw std::invalid_argument("arrDepthIndices cannot be null.")
            ↪ ;
      if (uiDepthCount == 0)
        throw std::invalid_argument("arrDepthIndices cannot be empty."
            ↪ );
      if (width == 0)
        throw std::invalid_argument("width must be positive.");
      if (height == 0)
        throw std::invalid_argument("height must be positive.");

      m_uiWidth = width;
      m_uiHeight = height;

      m_arrDepthIndices.assign(arrDepthIndices, arrDepthIndices +
          ↪ uiDepthCount);
    }

    /// <summary>
    /// Copy constructor.
    /// </summary>
    Filter32(const Filter32 &src) :
      Filter32(src.getWidth(), src.getHeight(),
        src.getDepthIndices(), src.getDepthCount())
    { }

    Filter32(Filter32 &&src)
    {
      m_uiWidth = src.getWidth();
      m_uiHeight = src.getHeight();
      m_arrDepthIndices = std::move(src.m_arrDepthIndices);
    }

    Filter32& operator=(const Filter32 &src)
    {
```

```
      if (this != &src)
      {
        m_uiWidth = src.getWidth();
        m_uiHeight = src.getHeight();
        m_arrDepthIndices = src.m_arrDepthIndices;
      }
      return *this;
    }

    Filter32& operator=(Filter32 &&src)
    {
      if (this != &src)
      {
        m_uiWidth = src.getWidth();
        m_uiHeight = src.getHeight();
        m_arrDepthIndices = std::move(src.m_arrDepthIndices);
      }
      return *this;
    }

    void Save(std::ostream &fNum, const LayerKey &key) const;

  private:
    std::size_t m_uiWidth;
    std::size_t m_uiHeight;
    std::vector<depth_index_type> m_arrDepthIndices;
  };


  /// <summary>
  /// Implements the Filter32 class as an input filter for a
     ↪ convolutional layer.
  /// </summary>
  class InputFilter32 : public Filter32
  {
  public:

    float getBias() const override
    {
      return m_fBias;
    }

    void setBias(float value) override
    {
      m_fBias = value;
    }

    std::size_t getFilterElementsCount() const override
    {
      return m_arrFilter.size();
    }
```

```
const float *getFilterElements() const override
{
  return m_arrFilter.data();
}

float *getFilterElements()
{
  return m_arrFilter.data();
}

void setFilterElements(const float *arr) override
{
  memcpy(m_arrFilter.data(), arr, m_arrFilter.size() * sizeof(
    ↪ float));
}

void setFilterElement(std::size_t index, float value) override
{
  m_arrFilter[index] = value;
}


InputFilter32(const std::vector<std::vector<std::vector<float>>> &
  ↪ arrFilter,
    const std::vector<Filter32::depth_index_type> &
      ↪ arrDepthIndices) :
  InputFilter32(arrFilter.at(0).at(0).size(), arrFilter.at(0).size
    ↪ (), arrDepthIndices.data(), arrFilter.size())
{
  std::size_t depth = arrFilter.size();
  std::size_t height = arrFilter[0].size();
  std::size_t width = arrFilter[0][0].size();

  if (depth != arrDepthIndices.size())
    throw std::invalid_argument("Filter32's␣array␣depth␣and␣number
      ↪ ␣of␣depth␣indices␣do␣not␣match");

  for (std::size_t z = 0; z < depth; z++)
  {
    if (height != arrFilter[z].size())
      throw std::invalid_argument("Jagged␣array␣arrFilter␣not␣
        ↪ supported.");
    for (std::size_t y = 0; y < height; y++)
    {
      if (width != arrFilter[z][y].size())
        throw std::invalid_argument("Jagged␣array␣arrFilter␣not␣
          ↪ supported.");
      for (std::size_t x = 0; x < width; x++)
        Filter32::setFilterElement(x, y, z, arrFilter[z][y][x]);
    }
```

```
    }
  }

  InputFilter32(const float ***arrFilter, std::size_t width, std::
      ↪ size_t height,
      const Filter32::depth_index_type *arrDepthIndices, std::size_t
          ↪ uiDepthCount) :
    InputFilter32(width, height, arrDepthIndices, uiDepthCount)
  {
    for (std::size_t z = 0; z < uiDepthCount; z++)
      for (std::size_t y = 0; y < height; y++)
        for (std::size_t x = 0; x < width; x++)
          Filter32::setFilterElement(x, y, z, arrFilter[z][y][x]);
  }

  InputFilter32(const float *arrFilter, std::size_t width, std::
      ↪ size_t height,
      const Filter32::depth_index_type *arrDepthIndices, std::size_t
          ↪ uiDepthCount) :
    InputFilter32(width, height, arrDepthIndices, uiDepthCount)
  {
    m_arrFilter.assign(arrFilter, arrFilter + m_arrFilter.size());
  }

  InputFilter32(std::size_t width, std::size_t height,
      const Filter32::depth_index_type *arrDepthIndices, std::size_t
          ↪ uiDepthCount) :
    Filter32(width, height, arrDepthIndices, uiDepthCount)
  {
    m_arrFilter.resize(width * height * uiDepthCount);
    setBias(0.0);
  }

  InputFilter32(const InputFilter32 &src) :
    Filter32(src)
  {
    m_fBias = src.m_fBias;
    m_arrFilter = src.m_arrFilter;
  }

  InputFilter32(InputFilter32 &&src) :
    Filter32(src)
  {
    m_fBias = src.m_fBias;
    m_arrFilter = std::move(src.m_arrFilter);
  }

  InputFilter32& operator=(const InputFilter32 &src)
  {
    if (this != &src)
    {
```

```
            *(Filter32*)this = src;
            m_fBias = src.m_fBias;
            m_arrFilter = src.m_arrFilter;
          }
          return *this;
        }

        InputFilter32& operator=(InputFilter32 &&src)
        {
          if (this != &src)
          {
            *(Filter32*)this = std::move(src);
            m_fBias = src.m_fBias;
            m_arrFilter = std::move(src.m_arrFilter);
          }
          return *this;
        }

        static std::shared_ptr<InputFilter32> Load(std::istream &fNum);

      private:
        float m_fBias;
        std::vector<float> m_arrFilter;
      };

      typedef Filter32 Filter;
      typedef InputFilter32 InputFilter;
    }
}

#endif
```

B.3 File: kconvfilter.cpp

```
#include <ostream>

#include "kconvfilter.h"

void NeuralNetwork::Convolutional::Filter32::Save(std::ostream & fNum,
    ↪ const LayerKey & key) const
{
  std::uint64_t u_output = KUtilities::UInt64::getLittleEndian(this->
      ↪ getWidth());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::Filter32::
        ↪ Save(ostream &fNum, const LayerKey &key)]: "
      "Error writing filter width.");
```

```
  u_output = KUtilities::UInt64::getLittleEndian(this->getHeight());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::Filter32::
        ↪ Save(ostream &fNum, const LayerKey &key)]: "
      "Error writing filter height.");

  u_output = KUtilities::UInt64::getLittleEndian(this->getDepthCount());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::Filter32::
        ↪ Save(ostream &fNum, const LayerKey &key)]: "
      "Error writing filter depth count.");
  for (std::size_t depth_i = 0; depth_i < this->getDepthCount(); depth_i
      ↪ ++)
  {
    u_output = KUtilities::UInt64::getLittleEndian(this->getDepthIndices
        ↪ ()[depth_i]);
    fNum.write((char*)&u_output, sizeof(u_output));
    if (!fNum.good())
      throw std::runtime_error("[NeuralNetwork::Convolutional::Filter32
          ↪ ::Save(ostream &fNum, const LayerKey &key)]: "
        "Error writing depth index.");
  }

  float bias = this->getBias();
  fNum.write((const char*)&bias, sizeof(float));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::Filter32::
        ↪ Save(ostream &fNum, const LayerKey &key)]: "
      "Error writing filter bias.");

  fNum.write((const char*)this->getFilterElements(), sizeof(float) *
      ↪ this->getFilterElementsCount());
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::Filter32::
        ↪ Save(ostream &fNum, const LayerKey &key)]: "
      "Error writing filter elements.");
}

std::shared_ptr<NeuralNetwork::Convolutional::InputFilter32>
    ↪ NeuralNetwork::Convolutional::InputFilter32::Load(std::istream &
    ↪ fNum)
{
  char buffer[8];

  std::vector<float> arrFilter;
  std::vector<std::size_t> arrDepthIndices;
  std::size_t uiFilterWidth, uiFilterHeight, uiDepthCount;

  fNum.read(buffer, sizeof(std::uint64_t));
```

```
if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
  throw std::runtime_error("[NeuralNetwork::Convolutional::
    ↪ InputFilter32::Load(istream &fNum)]: "
    "Error reading filter width.");
uiFilterWidth = KUtilities::UInt64::ToEndian(buffer, true);

fNum.read(buffer, sizeof(std::uint64_t));
if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
  throw std::runtime_error("[NeuralNetwork::Convolutional::
    ↪ InputFilter32::Load(istream &fNum)]: "
    "Error reading filter height.");
uiFilterHeight = KUtilities::UInt64::ToEndian(buffer, true);

fNum.read(buffer, sizeof(std::uint64_t));
if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
  throw std::runtime_error("[NeuralNetwork::Convolutional::
    ↪ InputFilter32::Load(istream &fNum)]: "
    "Error reading filter depth count.");
uiDepthCount = KUtilities::UInt64::ToEndian(buffer, true);

arrDepthIndices.resize(uiDepthCount);
for (std::size_t depth_i = 0; depth_i < uiDepthCount; depth_i++)
{
  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Convolutional::
      ↪ InputFilter32::Load(istream &fNum)]: "
      "Error reading filter depth count.");
  arrDepthIndices[depth_i] = KUtilities::UInt64::ToEndian(buffer, true
    ↪ );
}

float bias;
fNum.read((char*)&bias, sizeof(float));
if (!fNum.good() || fNum.gcount() != sizeof(float))
  throw std::runtime_error("[NeuralNetwork::Convolutional::
    ↪ InputFilter32::Load(istream &fNum)]: "
    "Error reading filter bias.");


arrFilter.resize(uiFilterWidth * uiFilterHeight * uiDepthCount);
fNum.read((char*)arrFilter.data(), sizeof(float) * arrFilter.size());
if (!fNum.good() || fNum.gcount() != sizeof(float) * arrFilter.size())
  throw std::runtime_error("[NeuralNetwork::Convolutional::
    ↪ InputFilter32::Load(istream &fNum)]: "
    "Error reading filter elements.");

std::shared_ptr<InputFilter32>retval = std::shared_ptr<InputFilter32>(
  ↪ new InputFilter32(
                    arrFilter.data(),
                    uiFilterWidth, uiFilterHeight,
```

```
                                arrDepthIndices.data(), uiDepthCount));
  retval->setBias(bias);

  return retval;
}
```

B.4 File: kconvlayer.h

```
#ifndef K_FFANN_CONVOLUTIONAL_LAYER_H
#define K_FFANN_CONVOLUTIONAL_LAYER_H

#include <vector>
#include <stdexcept>
#include <memory>

#include "knnkeys.h"

#include "klayer.h"
#include "kconvfilter.h"
#include "kconvfeature.h"

#include "ktypeutils.h"


namespace NeuralNetwork
{
  class Network32;

  namespace Convolutional
  {
    class ConvolutionalLayer32 : public TrainableLayer32
    {
    public:

      static const int LAYER_TYPE = 405;

      /// <summary>
      /// Width of the output (equivalent to number of columns).
      /// </summary>
      std::size_t getOutputWidth() const
      {
        return m_uiOutputWidth;
      }

      /// <summary>
      /// Height of the output (equivalent to number of rows).
      /// </summary>
      std::size_t getOutputHeight() const
```

```
    {
      return m_uiOutputHeight;
    }

    /// <summary>
    /// Depth of the output (equivalent to number of channels or
        ↪ features).
    /// </summary>
    std::size_t getOutputDepth() const
    {
      return m_features.size();
    }

    /// <summary>
    /// Width of the input (equivalent to number of columns).
    /// </summary>
    std::size_t getInputWidth() const
    {
      return m_uiInputWidth;
    }

    /// <summary>
    /// Height of the input (equivalent to number of rows).
    /// </summary>
    std::size_t getInputHeight() const
    {
      return m_uiInputHeight;
    }

    /// <summary>
    /// Depth of the input (equivalent to number of channels).
    /// </summary>
    std::size_t getInputDepth() const
    {
      return m_uiInputDepth;
    }


    std::size_t getFeaturesCount() const
    {
      return m_features.size();
    }

    Feature32 &getFeature(std::size_t index) const
    {
      return *m_features.at(index);
    }

    /// <summary>
    /// Initializes a new convolutional layer.
    /// </summary>
```

```
    ConvolutionalLayer32(const Network32 &network,
      const std::shared_ptr<Filter32> *arrFilters, const std::size_t *
          ↪ arrFeatureSteps, std::size_t uiNumFeatures,
        const NeuralNetwork::ActivationFunctions::IActivationFunction<
            ↪ float> &f,
        std::size_t uiInputWidth, std::size_t uiInputHeight, std::
            ↪ size_t uiInputDepth,
        const Layer32 &prevLayer, std::size_t uiMinibatchSize, const
            ↪ LayerKey &key) :
      ConvolutionalLayer32(network, f, prevLayer, uiMinibatchSize, key
          ↪ )
    {
      init(arrFilters, arrFeatureSteps, uiNumFeatures, uiInputWidth,
          ↪ uiInputHeight, uiInputDepth);
    }

    /// <summary>
    /// Initializes a new convolutional layer using another
        ↪ convolutional layer as input layer.
    /// </summary>
    ConvolutionalLayer32(const Network32 &network,
        const std::shared_ptr<Filter32> *arrFilters, const std::size_t
            ↪  *arrFeatureSteps, std::size_t uiNumFeatures,
        const NeuralNetwork::ActivationFunctions::IActivationFunction<
            ↪ float> &f,
        const ConvolutionalLayer32 &prevLayer, std::size_t
            ↪ uiMinibatchSize, const LayerKey &key) :
      ConvolutionalLayer32(network, arrFilters, arrFeatureSteps,
          ↪ uiNumFeatures, f,
        prevLayer.getOutputWidth(), prevLayer.getOutputHeight(),
            ↪ prevLayer.getOutputDepth(),
        prevLayer, uiMinibatchSize, key) { }

    ConvolutionalLayer32(const ConvolutionalLayer32&) = delete;
    ConvolutionalLayer32& operator=(const ConvolutionalLayer32&) =
        ↪ delete;

    virtual ~ConvolutionalLayer32()
    { }

    int getLayerType() const override
    {
      return LAYER_TYPE;
    }

    /// <summary>
    /// Returns the index of the preceeding layer inside
    /// its network.
    /// </summary>
    std::size_t getPreviousLayerIndex() const override
    {
```

```
      return m_prevLayer.getNetworkIndex();
    }

    void Propagate() override;

    void Backpropagate(Layer32 &fromLayer) override;

    void PostBackpropagate(float fInvMinibatchSize, float
        ↪ fInvTotalSamples) override;

    void Save(std::ostream &fNum, const LayerKey &key) const;
    static std::shared_ptr<ConvolutionalLayer32> Load(std::istream &
        ↪ fNum, const Network32 &network,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f,
      const Layer32 &prevLayer, std::size_t uiMinibatchSize, const
          ↪ LayerKey &key);


  protected:
    /// <summary>
    /// Derived class initialization constructor.
    /// </summary>
    ConvolutionalLayer32(const Network32 &network,
        const NeuralNetwork::ActivationFunctions::IActivationFunction<
            ↪ float> &f,
        const Layer32 &prevLayer, std::size_t uiMinibatchSize, const
            ↪ LayerKey &key) :
      TrainableLayer32(network, 1, f, uiMinibatchSize, key),
      m_prevLayer(prevLayer)
    { }

    void init(const std::shared_ptr<Filter32> *arrFilters, const std::
        ↪ size_t *arrFeatureSteps, std::size_t uiNumFeatures,
      std::size_t uiInputWidth, std::size_t uiInputHeight, std::size_t
          ↪  uiInputDepth);

    const Layer32 &m_prevLayer;

    std::vector<float> m_buffer;

  private:
    std::size_t createConvolutionStructure(std::size_t uiFeatureIndex,
                        std::size_t uiConnectionsMade
                       );

    std::vector<std::shared_ptr <Feature32>> m_features;
    std::vector<int32_t> m_arrConnectionsIn;

    std::size_t m_uiOutputWidth;
    std::size_t m_uiOutputHeight;
```

```
      std::size_t m_uiNumNeuronsPerFeature;

      std::size_t m_uiInputWidth;
      std::size_t m_uiInputHeight;
      std::size_t m_uiInputDepth;

      std::vector<float> m_bufferSparseWeights;
      std::vector<int32_t> m_bufferSparseRowsPtr;
      std::vector<float> m_bufferPrevActivationT;
      std::vector<float> m_bufferConvolutionT;
    };

    typedef ConvolutionalLayer32 ConvolutionalLayer;
  }
}

#endif
```

B.5 File: kconvlayer.cpp

```
#include <iostream>

#include "kinputlayer.h"
#include "kconvlayer.h"
#include "kconvneuron.h"

#include "mkl.h"

#include "kopencl.h"


void NeuralNetwork::Convolutional::ConvolutionalLayer32::Propagate()
{




  std::memset(m_bufferConvolutionT.data(), 0, m_bufferConvolutionT.size
    ↪ () * sizeof(float));

  mkl_somatcopy('R', 'T',
    m_uiMinibatchSize, m_prevLayer.getNeuronCount(),
    1.0,
    m_prevLayer.getActivations(),
    m_prevLayer.getNeuronCount(),
    m_bufferPrevActivationT.data(),
    m_uiMinibatchSize);
```

```
const float constant_1 = 1.0;
const char *NoTrans = "N";
int32_t m = (int32_t)(m_uiNumNeuronsPerFeature * m_features.size());
int32_t n = (int32_t)m_uiMinibatchSize;
int32_t k = (int32_t)m_prevLayer.getNeuronCount();
char matdescra[6] = { 'G',
             0, 0,
            'C',
             0, 0 };
const float *b = m_bufferPrevActivationT.data();
int32_t ldb = n;
int32_t ldc = n;

std::vector<float> &val = m_bufferSparseWeights;
const int32_t *rowptr = m_bufferSparseRowsPtr.data();
const int32_t *colind = m_arrConnectionsIn.data();
int32_t nnz = (int32_t)val.size();

float *c = m_bufferConvolutionT.data();

#pragma omp parallel for
for (int feature_i = 0; feature_i < m_features.size(); feature_i++)
{
  Filter32 &filter = *m_features[feature_i]->getFilter();
  std::size_t uiFeatureWeightSize = m_uiNumNeuronsPerFeature * filter.
    ↪ getFilterElementsCount();
  std::memcpy(val.data() + uiFeatureWeightSize * feature_i, filter.
    ↪ getFilterElements(), filter.getFilterElementsCount() * sizeof(
    ↪ float));
  KUtilities::replicate_array(val.data() + uiFeatureWeightSize *
    ↪ feature_i, filter.getFilterElementsCount(),
    ↪ uiFeatureWeightSize);
}

if (NeuralNetwork::bUseGPU)
  KNNUtils::KBLASOpenCL::scsrmm(c, n,
    false,
    val.data(), (uint32_t*)colind, (uint32_t*)rowptr,
    m, k, b);
else
  mkl_scsrmm(NoTrans, &m, &n, &k,
    &constant_1,
    matdescra, val.data(), colind, rowptr, rowptr + 1,
    b, &ldb, &constant_1,
    c, &ldc);

#pragma omp parallel for
for (int feature_i = 0; feature_i < m_features.size(); feature_i++)
```

```
  {
    m_weighted_input[feature_i * m_uiNumNeuronsPerFeature] = m_biases[
        ↪ feature_i];
    KUtilities::replicate_array(m_weighted_input.data() + feature_i *
        ↪ m_uiNumNeuronsPerFeature, 1, m_uiNumNeuronsPerFeature);
  }
  KUtilities::replicate_array(m_weighted_input.data(), this->
      ↪ getNeuronCount(), m_weighted_input.size());



  mkl_somatadd('R', 'T', 'N',
    m_uiMinibatchSize, this->getNeuronCount(),
    1.0,
    m_bufferConvolutionT.data(),
    m_uiMinibatchSize,
    1.0,
    m_weighted_input.data(),
    this->getNeuronCount(),
    m_weighted_input.data(),
    this->getNeuronCount());


  m_pfuncActivation->f(m_activations.data(), m_weighted_input.data(),
      ↪ m_weighted_input.size());
}

void NeuralNetwork::Convolutional::ConvolutionalLayer32::Backpropagate(
    ↪ Layer32 & fromLayer)
{
  if (fromLayer.getLayerType() != NeuralNetwork::InputLayer32::
      ↪ LAYER_TYPE)
  {



    std::vector<float> &delta_from_prime = m_bufferPrevActivationT;
    std::vector<float> &delta_to_Trans = m_bufferConvolutionT;




    std::memset(delta_from_prime.data(), 0, delta_from_prime.size() *
        ↪ sizeof(float));
    mkl_somatcopy('R', 'T',
      m_uiMinibatchSize, this->getNeuronCount(),
      1.0,
      this->getDeltas(),
      this->getNeuronCount(),
```

```
    delta_to_Trans.data(),
    m_uiMinibatchSize);


  const float constant_1 = 1.0;
  const char *Trans = "T";
  int32_t m = (int32_t)(m_uiNumNeuronsPerFeature * m_features.size());
  int32_t n = (int32_t)m_uiMinibatchSize;
  int32_t k = (int32_t)fromLayer.getNeuronCount();
  char matdescra[6] = { 'G',
              0, 0,
               'C',
              0, 0 };
  const float *b = delta_to_Trans.data();
  int32_t ldb = n;
  int32_t ldc = n;

  std::vector<float> &val = m_bufferSparseWeights;
  const int32_t *rowptr = m_bufferSparseRowsPtr.data();
  const int32_t *colind = m_arrConnectionsIn.data();
  int32_t nnz = (int32_t)val.size();

  float *c = delta_from_prime.data();

  #pragma omp parallel for
  for (int feature_i = 0; feature_i < m_features.size(); feature_i++)
  {
    Filter32 &filter = *m_features[feature_i]->getFilter();
    std::size_t uiFeatureWeightSize = m_uiNumNeuronsPerFeature *
        ↪ filter.getFilterElementsCount();
    std::memcpy(val.data() + uiFeatureWeightSize * feature_i, filter.
        ↪ getFilterElements(), filter.getFilterElementsCount() *
        ↪ sizeof(float));
    KUtilities::replicate_array(val.data() + uiFeatureWeightSize *
        ↪ feature_i, filter.getFilterElementsCount(),
        ↪ uiFeatureWeightSize);
  }

  if (NeuralNetwork::bUseGPU)
    KNNUtils::KBLASOpenCL::scsrmm(c, n,
      true,
      val.data(), (uint32_t*)colind, (uint32_t*)rowptr,
      m, k, b);
  else
    mkl_scsrmm(Trans, &m, &n, &k,
      &constant_1,
      matdescra, val.data(), colind, rowptr, rowptr + 1,
      b, &ldb, &constant_1,
      c, &ldc);

  mkl_somatcopy('R', 'T',
```

```
        fromLayer.getNeuronCount(), m_uiMinibatchSize,
        1.0,
        delta_from_prime.data(),
        m_uiMinibatchSize,
        fromLayer.getDeltaVector().unsafe_data(),
        fromLayer.getNeuronCount());

    m_buffer.resize(fromLayer.getWeightedInputVector().size());
    fromLayer.getActivationFunction().df(m_buffer.data(), fromLayer.
        ↪ getWeightedInputValues(), fromLayer.getWeightedInputVector().
        ↪ size());
    vsMul((int)m_buffer.size(), fromLayer.getDeltaVector().data(),
        ↪ m_buffer.data(), fromLayer.getDeltaVector().unsafe_data());
  }


  for (std::size_t sample_i = 0; sample_i < m_uiMinibatchSize; sample_i
      ↪ ++)
  {
    const float * from_activations = fromLayer.getActivations(sample_i);
    #pragma omp parallel for
    for (int neuron_to_i = 0; neuron_to_i < (int)this->getNeuronCount();
        ↪  neuron_to_i++)
    {
      NeuralNetwork::Convolutional::ConvolutionalNeuron32 &neuron_to =
        (NeuralNetwork::Convolutional::ConvolutionalNeuron32&)this->
            ↪ getNeuron(neuron_to_i);
      float neuron_to_delta = neuron_to.getDelta(sample_i);
      const int32_t *neuron_from_indices = neuron_to.getNeuronFrom();

      for (std::size_t connection_index = 0; connection_index <
          ↪ neuron_to.getConnectionsInCount(); connection_index++)
        neuron_to.getDeltaWeights()[connection_index] +=
        (neuron_to_delta * from_activations[neuron_from_indices[
            ↪ connection_index]]);

      neuron_to.DeltaBias += neuron_to_delta;
    }
  }
}

void NeuralNetwork::Convolutional::ConvolutionalLayer32::
    ↪ PostBackpropagate(float fInvMinibatchSize, float fInvTotalSamples)
{
  #pragma omp parallel for
  for (int j = 0; j < (int)this->getNeuronCount(); j++)
  {
    Neuron32 &neuron = this->getNeuron(j);

    neuron.UpdateWeights(fInvMinibatchSize, fInvTotalSamples);
    neuron.UpdateBias(fInvMinibatchSize);
```

```
    neuron.ScaleDeltaWeights();
    neuron.ScaleDeltaBias();
  }
}

void NeuralNetwork::Convolutional::ConvolutionalLayer32::Save(std::
    ↪ ostream & fNum, const LayerKey & key) const
{

  std::uint64_t u_output;


  u_output = KUtilities::UInt64::getLittleEndian(this->getInputWidth());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ ConvolutionalLayer32::Save(std::ostream &fNum, const LayerKey
        ↪ &key)]: "
      "Error writing input width.");

  u_output = KUtilities::UInt64::getLittleEndian(this->getInputHeight())
      ↪ ;
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ ConvolutionalLayer32::Save(std::ostream &fNum, const LayerKey
        ↪ &key)]: "
      "Error writing input height.");

  u_output = KUtilities::UInt64::getLittleEndian(this->getInputDepth());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ ConvolutionalLayer32::Save(std::ostream &fNum, const LayerKey
        ↪ &key)]: "
      "Error writing input depth.");

  u_output = KUtilities::UInt64::getLittleEndian(this->getFeaturesCount
      ↪ ());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ ConvolutionalLayer32::Save(ostream &fNum, const LayerKey &key)
        ↪ ]: "
      "Error writing feature count.");

  for (std::size_t feature_i = 0; feature_i < this->getFeaturesCount();
      ↪ feature_i++)
  {
```

```
    NeuralNetwork::Convolutional::Feature32 &feature = this->getFeature(
        ↪ feature_i);

    u_output = KUtilities::UInt64::getLittleEndian(feature.getStep());
    fNum.write((char*)&u_output, sizeof(u_output));
    if (!fNum.good())
      throw std::runtime_error("[NeuralNetwork::Convolutional::
          ↪ ConvolutionalLayer32::Save(ostream &fNum, const LayerKey &
          ↪ key)]: "
        "Error writing next feature step.");


    feature.getFilter()->Save(fNum, key);
  }
}

std::shared_ptr<NeuralNetwork::Convolutional::ConvolutionalLayer32>
    ↪ NeuralNetwork::Convolutional::ConvolutionalLayer32::Load(std::
    ↪ istream & fNum,
  const Network32 & network, const NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float> & f,
  const Layer32 & prevLayer, std::size_t uiMinibatchSize, const LayerKey
      ↪  & key)
{
  std::size_t uiNumFeatures,
        uiInputWidth, uiInputHeight, uiInputDepth;
  char buffer[8];



  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ MaxPoolLayer::Load(istream &fNum, const Network32 &network, 
        ↪ size_t uiMinibatchSize, const NeuralNetwork::
        ↪ ActivationFunctions::IActivationFunction &f, const Layer32 &
        ↪ prevLayer, const LayerKey &key)]: "
      "Error reading input width.");
  uiInputWidth = KUtilities::UInt64::ToEndian(buffer, true);

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ MaxPoolLayer::Load(istream &fNum, const Network32 &network, 
        ↪ size_t uiMinibatchSize, const NeuralNetwork::
        ↪ ActivationFunctions::IActivationFunction &f, const Layer32 &
        ↪ prevLayer, const LayerKey &key)]: "
      "Error reading input height.");
  uiInputHeight = KUtilities::UInt64::ToEndian(buffer, true);

  fNum.read(buffer, sizeof(std::uint64_t));
```

```
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ MaxPoolLayer::Load(istream &fNum, const Network32 &network,
        ↪ size_t uiMinibatchSize, const NeuralNetwork::
        ↪ ActivationFunctions::IActivationFunction &f, const Layer32 &
        ↪ prevLayer, const LayerKey &key)]: "
      "Error reading input depth.");
  uiInputDepth = KUtilities::UInt64::ToEndian(buffer, true);

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ MaxPoolLayer::Load(istream &fNum, const Network32 &network,
        ↪ size_t uiMinibatchSize, const NeuralNetwork::
        ↪ ActivationFunctions::IActivationFunction &f, const Layer32 &
        ↪ prevLayer, const LayerKey &key)]: "
      "Error reading number of features.");
  uiNumFeatures = KUtilities::UInt64::ToEndian(buffer, true);

  std::vector<std::size_t> arrFeatureSteps(uiNumFeatures);
  std::vector<std::shared_ptr<NeuralNetwork::Convolutional::Filter32>>
      ↪ arrFilters(uiNumFeatures);
  for (std::size_t filter_i = 0; filter_i < uiNumFeatures; filter_i++)
  {
    fNum.read(buffer, sizeof(std::uint64_t));
    if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
      throw std::runtime_error("[NeuralNetwork::Convolutional::
          ↪ MaxPoolLayer::Load(istream &fNum, const Network32 &network,
          ↪ size_t uiMinibatchSize, const NeuralNetwork::
          ↪ ActivationFunctions::IActivationFunction &f, const Layer32 &
          ↪ prevLayer, const LayerKey &key)]: "
        "Error reading feature step.");
    arrFeatureSteps[filter_i] = KUtilities::UInt64::ToEndian(buffer,
        ↪ true);

    arrFilters[filter_i] = InputFilter32::Load(fNum);
  }


  return std::shared_ptr<ConvolutionalLayer32>(
      new Convolutional::ConvolutionalLayer32(
        network, arrFilters.data(), arrFeatureSteps.data(),
            ↪ uiNumFeatures, f,
        uiInputWidth, uiInputHeight, uiInputDepth,
        prevLayer, uiMinibatchSize, key));
}

void NeuralNetwork::Convolutional::ConvolutionalLayer32::init(const std
    ↪ ::shared_ptr<Filter32> *arrFilters, const std::size_t *
    ↪ arrFeatureSteps, std::size_t uiNumFeatures, std::size_t
    ↪ uiInputWidth, std::size_t uiInputHeight, std::size_t uiInputDepth)
```

```
{
  if (!arrFilters)
    throw std::invalid_argument("arrFilters cannot be null.");
  if (!arrFeatureSteps)
    throw std::invalid_argument("arrFeatureSteps cannot be null.");
  if (uiNumFeatures == 0)
    throw std::out_of_range("uiNumFeatures must be positive.");
  if (uiInputWidth == 0)
    throw std::out_of_range("uiInputWidth must be positive.");
  if (uiInputHeight == 0)
    throw std::out_of_range("uiInputHeight must be positive.");
  if (uiInputDepth == 0)
    throw std::out_of_range("uiInputDepth must be positive.");

  m_uiInputWidth = uiInputWidth;
  m_uiInputHeight = uiInputHeight;
  m_uiInputDepth = uiInputDepth;

  std::size_t uiNumWeights = 0;
  for (int i = 0; i < (int)uiNumFeatures; i++)
  {
    uiNumWeights += arrFilters[i]->getFilterElementsCount();
  }
  m_weights.resize(uiNumWeights);

  m_biases.resize(uiNumFeatures);

  m_uiOutputWidth = 0;
  m_uiOutputHeight = 0;
  m_features.resize(uiNumFeatures);
  for (int i = 0; i < (int)uiNumFeatures; i++)
  {
    m_features[i] = std::shared_ptr<Feature32>(
      new Feature32(arrFeatureSteps[i], *arrFilters[i],
        m_biases, i,
        m_weights, i * arrFilters[i]->getFilterElementsCount(),
        uiInputWidth, uiInputHeight, i,
        FeatureKey()));

    if (m_uiOutputWidth == 0)
      m_uiOutputWidth = m_features[i]->getOutputWidth();
    else if (m_uiOutputWidth != m_features[i]->getOutputWidth())
      throw std::runtime_error("New feature does not match width of pre-
        ↪ existing features.");
    if (m_uiOutputHeight == 0)
      m_uiOutputHeight = m_features[i]->getOutputHeight();
    else if (m_uiOutputHeight != m_features[i]->getOutputHeight())
      throw std::runtime_error("New feature does not match height of pre
        ↪ -existing features.");
  }
```

```
  m_uiNumNeuronsPerFeature = m_uiOutputWidth * m_uiOutputHeight;
  std::size_t uiNumNeurons = m_uiNumNeuronsPerFeature * uiNumFeatures;
  m_activations.resize(uiNumNeurons * m_uiMinibatchSize);
  m_weighted_input.resize(uiNumNeurons * m_uiMinibatchSize);
  m_deltas.resize(uiNumNeurons * m_uiMinibatchSize);
  m_neurons.resize(uiNumNeurons);
  std::size_t uiTotalConnectionsIn = 0;
  for (int feature_i = 0; feature_i < (int)uiNumFeatures; feature_i++)
    uiTotalConnectionsIn += (m_uiNumNeuronsPerFeature * m_features[
        ↪ feature_i]->getFilter()->getFilterElementsCount());
  m_arrConnectionsIn.resize(uiTotalConnectionsIn);

  std::size_t uiNumConnectionsMade = 0;
  for (int feature_i = 0; feature_i < (int)uiNumFeatures; feature_i++)
    uiNumConnectionsMade += createConvolutionStructure(feature_i,
        ↪ uiNumConnectionsMade);

  m_bufferPrevActivationT.resize(m_prevLayer.getActivationsVector().size
      ↪ ());
  m_bufferConvolutionT.resize(m_weighted_input.size());


  m_bufferSparseWeights.resize(uiTotalConnectionsIn);
  m_bufferSparseRowsPtr.resize(uiNumNeurons + 1);
  m_bufferSparseRowsPtr[0] = 0;
  for (int neuron_i = 0; neuron_i < (int)uiNumNeurons; neuron_i++)
    m_bufferSparseRowsPtr[neuron_i + 1] = m_bufferSparseRowsPtr[neuron_i
        ↪ ] +
                          (int)(this->getNeuron(neuron_i).
                              ↪ getConnectionsInCount());
}

std::size_t NeuralNetwork::Convolutional::ConvolutionalLayer32::
    ↪ createConvolutionStructure(std::size_t uiFeatureIndex, std::size_t
    ↪  uiConnectionsMade)
{
  std::size_t uiRetvalNumConnections = 0;
  Feature32 &feature = *m_features[uiFeatureIndex];
  std::size_t neuron_i = m_uiNumNeuronsPerFeature * uiFeatureIndex;
  for (std::size_t outputY = 0; outputY < feature.getOutputHeight();
      ↪ outputY++)
  {
    std::size_t inputTop = outputY * feature.getStep();
    for (std::size_t outputX = 0; outputX < feature.getOutputWidth();
        ↪ outputX++)
    {
      std::size_t inputLeft = outputX * feature.getStep();
      std::size_t uiConnectionInIndex = uiConnectionsMade +
          ↪ uiRetvalNumConnections;

      ConvolutionalNeuron32 *neuron;
```

```
        m_neurons[neuron_i] = std::shared_ptr<Neuron32>(
          neuron = new ConvolutionalNeuron32(m_activations,
            m_biases, feature.getLayerIndex(),
            m_weighted_input,
            m_deltas,
            m_weights, feature.getStartWeight(), feature.getFilter()->
                ↪ getFilterElementsCount(),
            m_arrConnectionsIn, uiConnectionInIndex,
            neuron_i,
            m_neurons.size(), m_uiMinibatchSize, m_NeuronKey)
          );

        for (std::size_t filterZ = 0; filterZ < feature.getFilter()->
            ↪ getDepthCount(); filterZ++)
        {
          std::size_t channel = feature.getFilter()->getDepthIndices()[
              ↪ filterZ];
          std::size_t inputY = inputTop;
          for (std::size_t filterY = 0; filterY < feature.getFilter()->
              ↪ getHeight(); filterY++)
          {
            std::size_t inputX = inputLeft;
            for (std::size_t filterX = 0; filterX < feature.getFilter()->
                ↪ getWidth(); filterX++)
            {
              std::size_t neuron_from_index = (channel * m_uiInputHeight +
                  ↪  inputY) * m_uiInputWidth + inputX;

              neuron->UpdateConnectionIn(feature.getFilter()->
                  ↪ getFilterElementIndex(filterX, filterY, filterZ),
                m_prevLayer.getNeuron(neuron_from_index));

              inputX++;
            }

            inputY++;
          }

        }

        uiRetvalNumConnections += feature.getFilter()->
            ↪ getFilterElementsCount();
        neuron_i++;
      }
    }

    return uiRetvalNumConnections;
}
```

B.6 File: kconvneuron.h

```
#ifndef K_FFANN_CONVOLUTIONAL_NEURON_H
#define K_FFANN_CONVOLUTIONAL_NEURON_H

#include <vector>
#include <unordered_set>
#include <stdexcept>

#include "mkl.h"
#include "kneuron.h"

#include "ktypeutils.h"


namespace NeuralNetwork
{
  namespace Convolutional
  {
    class ConvolutionalNeuron32 : public Neuron32
    {
    public:
      ConvolutionalNeuron32(std::vector<float> &activation,
                KUtilities::atomic_vector<float> &biases, std::size_t
                    ↪ uiBiasIndex,
                std::vector<float> &weighted_inputs,
                KUtilities::atomic_vector<float> &deltas,
                KUtilities::atomic_vector<float> &weights, std::size_t
                    ↪ uiStartWeight, std::size_t uiNumWeights,
                std::vector<int32_t> &arrConnectionsIn, std::size_t
                    ↪ uiConnectionInIndex,
                std::size_t uiLayerIndex, std::size_t uiLayerNeuronCount
                    ↪ , std::size_t uiMinibatchSize, const NeuronKey &
                    ↪ key) :
        Neuron32(activation, biases, uiBiasIndex, weighted_inputs,
            ↪ deltas,
          weights, uiStartWeight, uiNumWeights,
          uiLayerIndex, uiLayerNeuronCount, uiMinibatchSize, key),
        m_arrConnectionsIn(arrConnectionsIn),
        m_uiConnectionInIndex(uiConnectionInIndex)
      { }

      /// <summary>
      /// Updates the connection with weight index by specifying
      /// that it is a connection coming from neuron_from.
      /// </summary>
      void UpdateConnectionIn(std::size_t index, Neuron32 &neuron_from)
      {
```

```
        m_arrConnectionsIn[m_uiConnectionInIndex + index] = (int32_t)
            ↪ neuron_from.getLayerIndex();
      }

      int32_t getNeuronFromIndex(std::size_t connection_index)
      {
        return m_arrConnectionsIn[m_uiConnectionInIndex +
            ↪ connection_index];
      }

      /// <summary>
      /// Returns the array of incoming connection indices to this
          ↪ neuron.
      /// </summary>
      const int32_t *getNeuronFrom() const
      {
        return m_arrConnectionsIn.data() + m_uiConnectionInIndex;
      }

    private:
      std::vector<int32_t> &m_arrConnectionsIn;
      std::size_t m_uiConnectionInIndex;
    };

    typedef ConvolutionalNeuron32 ConvolutionalNeuron;
  }
}

#endif
```

B.7 File: kfclayer.h

```
#ifndef K_FFANN_FULLY_CONNECTED_LAYER_H
#define K_FFANN_FULLY_CONNECTED_LAYER_H

#include <vector>

#include "knnkeys.h"
#include "klayer.h"
#include "ktypeutils.h"


namespace NeuralNetwork
{
  class Network32;

  class FullyConnectedLayer32 : public TrainableLayer32
  {
```

```
public:

  static const int LAYER_TYPE = 402;


  FullyConnectedLayer32(const Network32 &network, std::size_t
      ↪ numNeurons,
            const NeuralNetwork::ActivationFunctions::
                ↪ IActivationFunction<float> &f,
            const Layer32 &prevLayer, std::size_t uiMinibatchSize,
                ↪ const LayerKey &key) :
    TrainableLayer32(network, numNeurons, f, uiMinibatchSize, key),
    m_prevLayer(prevLayer)
  {
    m_weights.resize(numNeurons * prevLayer.getNeuronCount());

    m_neurons.resize(numNeurons);
    for (std::size_t i = 0; i < numNeurons; i++)
      m_neurons[i] = std::shared_ptr<Neuron32>(
                    new Neuron32(m_activations, m_biases,
                        ↪ m_weighted_input, m_deltas,
                            m_weights, prevLayer.getNeuronCount(), i,
                                ↪ numNeurons,
                            uiMinibatchSize, m_NeuronKey)
                            );
  }

  FullyConnectedLayer32(const FullyConnectedLayer32&) = delete;
  FullyConnectedLayer32& operator=(const FullyConnectedLayer32&) =
      ↪ delete;

  virtual ~FullyConnectedLayer32()
  { }


  int getLayerType() const override
  {
    return LAYER_TYPE;
  }

  /// <summary>
  /// Returns the index of the preceeding layer inside
  /// its network.
  /// </summary>
  std::size_t getPreviousLayerIndex() const override
  {
    return m_prevLayer.getNetworkIndex();
  }

  void Propagate() override;
```

```
    void Backpropagate(Layer32 &fromLayer) override;

    void PostBackpropagate(float fInvMinibatchSize, float
        ↪ fInvTotalSamples) override;


    void Save(std::ostream &fNum, const LayerKey &key) const;

    static std::shared_ptr<FullyConnectedLayer32> Load(std::istream &
        ↪ fNum, const Network32 &network, std::size_t numNeurons,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f, const Layer32 &prevLayer, std::size_t
          ↪ uiMinibatchSize, const LayerKey &key);

  protected:
    void Load(std::istream &fNum, const LayerKey &key);


  private:
    std::vector<float> m_buffer;
    const Layer32 &m_prevLayer;
  };


  typedef FullyConnectedLayer32 FullyConnectedLayer;
}

#endif
```

B.8 File: kfclayer.cpp

```
#include <cstdint>
#include <cstring>
#include "mkl.h"

#include "kinputlayer.h"
#include "kfclayer.h"


void NeuralNetwork::FullyConnectedLayer32::Propagate()
{

  std::size_t minibatch_size = m_uiMinibatchSize;



  std::memcpy(m_weighted_input.data(), m_biases.data(), this->
      ↪ getNeuronCount() * sizeof(float));
```

```
  KUtilities::replicate_array(m_weighted_input.data(), this->
      ↪ getNeuronCount(), m_weighted_input.size());

  int m = (int)minibatch_size;
  int n = (int)this->getNeuronCount();
  int k = (int)m_prevLayer.getNeuronCount();
  const float *A = m_prevLayer.getActivations();
  const float *B = this->getWeights();
  float *C = m_weighted_input.data();
  cblas_sgemm(CblasRowMajor,
    CblasNoTrans, CblasTrans,
    m, n, k,
    1.0,
    A,
    k,
    B,
    k,
    1.0,
    C,
    n);



  m_pfuncActivation->f(m_activations.data(), m_weighted_input.data(),
      ↪ m_weighted_input.size());
}

void NeuralNetwork::FullyConnectedLayer32::Backpropagate(NeuralNetwork::
    ↪ Layer32 & fromLayer)
{
  if (fromLayer.getLayerType() != NeuralNetwork::InputLayer64::
      ↪ LAYER_TYPE)
  {



    std::memset(fromLayer.getDeltaVector().unsafe_data(), 0, fromLayer.
        ↪ getDeltaVector().size() * sizeof(float));

    int m = (int)m_uiMinibatchSize;
    int n = (int)fromLayer.getNeuronCount();
    int k = (int)this->getNeuronCount();
    const float *a = this->getDeltas();
    int lda = k;
    const float *b = this->getWeights();
    int ldb = n;
    float *c = fromLayer.getDeltaVector().unsafe_data();
    int ldc = n;

    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
      m, n, k,
```

```
        1.0,
        a, lda,
        b, ldb,
        1.0,
        c, ldc);

    m_buffer.resize(fromLayer.getWeightedInputVector().size());
    fromLayer.getActivationFunction().df(m_buffer.data(), fromLayer.
        ↪ getWeightedInputValues(), m_buffer.size());
    vsMul((int)m_buffer.size(), fromLayer.getDeltaVector().data(),
        ↪ m_buffer.data(), fromLayer.getDeltaVector().unsafe_data());
  }


  for (std::size_t sample_i = 0; sample_i < m_uiMinibatchSize; sample_i
      ↪ ++)
    #pragma omp parallel for
    for (int neuron_to_i = 0; neuron_to_i < (int)this->getNeuronCount();
        ↪  neuron_to_i++)
    {
      Neuron32 &neuron_to = this->getNeuron(neuron_to_i);
      cblas_saxpy((int)fromLayer.getNeuronCount(),
        neuron_to.getDelta(sample_i),
        fromLayer.getActivations(sample_i), 1,
        neuron_to.getDeltaWeights(), 1);

      neuron_to.DeltaBias += neuron_to.getDelta(sample_i);
    }
}

void NeuralNetwork::FullyConnectedLayer32::PostBackpropagate(float
    ↪ fInvMinibatchSize, float fInvTotalSamples)
{
  #pragma omp parallel for
  for (int j = 0; j < (int)this->getNeuronCount(); j++)
  {
    Neuron32 &neuron = this->getNeuron(j);

    neuron.UpdateWeights(fInvMinibatchSize, fInvTotalSamples);
    neuron.UpdateBias(fInvMinibatchSize);

    neuron.ScaleDeltaWeights();
    neuron.ScaleDeltaBias();
  }
}

void NeuralNetwork::FullyConnectedLayer32::Save(std::ostream &fNum,
    ↪ const LayerKey &key) const
{
  std::uint64_t u_output;
```

```
  u_output = KUtilities::UInt64::getLittleEndian(getWeightsCount());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::FullyConnectedLayer32::
      ↪ Save(std::ostream &fNum, const LayerKey &key)]: "
      "Error writing number of weights.");

  fNum.write((char*)m_weights.data(), sizeof(float) * m_weights.size());
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::FullyConnectedLayer32::
      ↪ Save(std::ostream &fNum, const LayerKey &key)]: "
      "Error writing all weights.");

  fNum.write((char*)m_biases.data(), sizeof(float) * m_biases.size());
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::FullyConnectedLayer32::
      ↪ Save(std::ostream &fNum, const LayerKey &key)]: "
      "Error writing all biases.");
}

void NeuralNetwork::FullyConnectedLayer32::Load(std::istream & fNum,
    ↪ const LayerKey & key)
{

  char buffer[8];

  fNum.read(buffer, sizeof(std::int64_t));
  if (!fNum.good() || fNum.gcount() != sizeof(std::int64_t))
    throw std::runtime_error("[NeuralNetwork::FullyConnectedLayer32::
      ↪ Load(std::istream &fNum, const LayerKey &key)]: "
      "Error reading number of weights.");
  std::size_t uiNumWeights = KUtilities::UInt64::ToEndian(buffer, true);
  if (uiNumWeights != this->getWeightsCount())
    throw std::runtime_error("[NeuralNetwork::FullyConnectedLayer32::
      ↪ Load(std::istream &fNum, const LayerKey &key)]: "
      "Number of weights in file is different than weights in layer.");

  fNum.read((char*)m_weights.unsafe_data(), sizeof(float) * m_weights.
      ↪ size());
  if (!fNum.good() || fNum.gcount() != sizeof(float) * m_weights.size())
    throw std::runtime_error("[NeuralNetwork::FullyConnectedLayer32::
      ↪ Load(std::istream &fNum, const LayerKey &key)]: "
      "Error reading all weights.");

  fNum.read((char*)m_biases.unsafe_data(), sizeof(float) * m_biases.size
      ↪ ());
  if (!fNum.good() || fNum.gcount() != sizeof(float) * m_biases.size())
    throw std::runtime_error("[NeuralNetwork::FullyConnectedLayer32::
      ↪ Load(std::istream &fNum, const LayerKey &key)]: "
      "Error reading all biases.");
```

```
}

std::shared_ptr<NeuralNetwork::FullyConnectedLayer32> NeuralNetwork::
    ↪ FullyConnectedLayer32::Load(std::istream & fNum,
  const Network32 & network, std::size_t numNeurons,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<float> &
      ↪ f,
  const Layer32 & prevLayer, std::size_t uiMinibatchSize, const LayerKey
      ↪ & key)
{
  std::shared_ptr<FullyConnectedLayer32> retval = std::shared_ptr<
      ↪ FullyConnectedLayer32>(
    new FullyConnectedLayer32(network, numNeurons, f, prevLayer,
        ↪ uiMinibatchSize, key));

  retval->Load(fNum, key);

  return retval;
}
```

B.9 File: kffann.h

```
#ifndef K_FEED_FORWARD_ARTIFICIAL_NEURAL_NETWORK_H
#define K_FEED_FORWARD_ARTIFICIAL_NEURAL_NETWORK_H

#include "ktypeutils.h"
#include "knetwork.h"
#include "klayer.h"
#include "kfclayer.h"
#include "kconvlayer.h"
#include "kconvfeature.h"
#include "kconvfilter.h"
#include "kconvneuron.h"
#include "kmaxpoollayer.h"
#include "kinputlayer.h"
#include "kfuncact.h"
#include "kfuncobj.h"
#include "kneuron.h"


#ifdef KNN_REQUEST_TRAINER
#include "ktrainer.h"
#endif

#endif
```

B.10 File: kfuncact.h

```
#ifndef K_ACTIVATION_AND_CUMULATIVE_FUNCTIONS_H
#define K_ACTIVATION_AND_CUMULATIVE_FUNCTIONS_H

#include <iostream>

#include <memory>
#include <vector>
#include <stdexcept>
#include <sstream>
#include <cmath>
#include <algorithm>
#include <cstring>

#include "klayer.h"
#include "knnkeys.h"

#include "ktypeutils.h"

namespace NeuralNetwork
{
  namespace ActivationFunctions
  {
    enum ActivationFunctionID {
      STEP_ID = 101,
      INVERTED_STEP_ID,
      HYPERBOLIC_TANGENT_ID,
      SIGMOID_ID,
      IDENTITY_ID,
      SOFTMAX_ID,
      RELU_ID
    };

    /// <summary>
    /// Encapsulates the identity activation function.
    /// </summary>
    class Identity32 : public IActivationFunction<float>
    {
    public:

      /// <summary>
      /// ID of the activation function.
      /// </summary>
      int getID() const override
      {
        return ActivationFunctionID::IDENTITY_ID;
      }

      /// <summary>
```

```
    /// Creates new IActivationFunction<float> object which parameters
        ↪ are
    /// a copy of this activation function.
    /// </summary>
    std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> CreateCopy() const override
    {
      return std::shared_ptr<IActivationFunction<float>>(new
          ↪ Identity32());
    }

    /// <summary>
    /// Activation function.
    /// </summary>
    void f(float *activations, const float *weighted_inputs, std::
        ↪ size_t size) override
    {
      std::memcpy(activations, weighted_inputs, size * sizeof(float));
    }

    /// <summary>
    /// Value of the first derivative of the activation function.
    /// </summary>
    void df(float *result, const float *weighted_inputs, std::size_t
        ↪ size) override
    {
      std::fill_n(result, size, 1.0f);
    }

    void Save(std::ostream &fNum, const ActivationFunctionKey &key)
        ↪ const override
    { }

    /// <summary>
    /// Loads an identity function from stream and returns a pointer
        ↪ to it.
    /// </summary>
    static std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> Load(std::istream &fNum, const
        ↪ ActivationFunctionKey &key)
    {
      return std::shared_ptr<IActivationFunction<float>>(new
          ↪ Identity32());
    }
  };

  /// <summary>
  /// Encapsulates the Step function.
  /// </summary>
  class Step32 : public IActivationFunction<float>
  {
```

```
public:

  Step32(float threshold = 0.0) : Threshold(threshold)
  { }

  /// <summary>
  /// ID of the activation function.
  /// </summary>
  int getID() const override
  {
    return ActivationFunctionID::STEP_ID;
  }

  float Threshold;

  /// <summary>
  /// Creates new IActivationFunction<float> object which parameters
     ↪ are
  /// a copy of this activation function.
  /// </summary>
  std::shared_ptr<NeuralNetwork::ActivationFunctions::
     ↪ IActivationFunction<float>> CreateCopy() const override
  {
    return std::shared_ptr<IActivationFunction<float>>(new Step32(
       ↪ Threshold));
  }

  /// <summary>
  /// Activation function.
  /// </summary>
  void f(float *activations, const float *weighted_inputs, std::
     ↪ size_t size) override
  {
    #pragma omp parallel for
    for (int i = 0; i < (int)size; i++)
      activations[i] = (weighted_inputs[i] <= Threshold ? 0.0f : 1.0
         ↪ f);
  }

  /// <summary>
  /// Value of the first derivative of the activation function.
  /// </summary>
  void df(float *result, const float *weighted_inputs, std::size_t
     ↪ size) override
  {
    std::fill_n(result, size, 0.0f);
  }

  void Save(std::ostream &fNum, const ActivationFunctionKey &key)
     ↪ const override
  {
```

```
      fNum.write((char*)&this->Threshold, sizeof(float));
      if (!fNum.good())
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
            ↪ ::Step::Save(std::ostream &fNum, const
            ↪ ActivationFunctionKey &key)]: Error writing step
            ↪ threshold.");
    }

    /// <summary>
    /// Loads a step function from stream and returns a pointer to it.
    /// </summary>
    static std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> Load(std::istream &fNum, const
        ↪ ActivationFunctionKey &key)
    {
      float threshold;
      fNum.read((char*)&threshold, sizeof(float));
      if (!fNum.good() || fNum.gcount() != sizeof(float))
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
            ↪ ::Step::Load(std::istream &fNum, const
            ↪ ActivationFunctionKey &key)]: Error reading step
            ↪ threshold.");

      return std::shared_ptr<IActivationFunction<float>>(new Step32(
          ↪ threshold));
    }
  };


  /// <summary>
  /// Encapsulates the Inverted Step function.
  /// </summary>
  class InvertedStep32 : public Step32
  {
  public:

    InvertedStep32(float threshold = 0.0) : Step32(threshold)
    { }

    /// <summary>
    /// ID of the activation function.
    /// </summary>
    int getID() const override
    {
      return ActivationFunctionID::INVERTED_STEP_ID;
    }

    /// <summary>
    /// Creates new IActivationFunction<float> object which parameters
        ↪ are
    /// a copy of this activation function.
```

```
    /// </summary>
    std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> CreateCopy() const override
    {
      return std::shared_ptr<IActivationFunction<float>>(new
          ↪ InvertedStep32(Threshold));
    }

    /// <summary>
    /// Activation function.
    /// </summary>
    void f(float *activations, const float *weighted_inputs, std::
        ↪ size_t size) override
    {
      #pragma omp parallel for
      for (int i = 0; i < (int)size; i++)
        activations[i] = (weighted_inputs[i] >= Threshold ? 0.0f : 1.0
            ↪ f);
    }

    /// <summary>
    /// Loads a step function from stream and returns a pointer to it.
    /// </summary>
    static std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> Load(std::istream &fNum, const
        ↪ ActivationFunctionKey &key)
    {
      float threshold;
      fNum.read((char*)&threshold, sizeof(float));
      if (!fNum.good() || fNum.gcount() != sizeof(float))
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
            ↪ ::InvertedStep::Load(std::istream &fNum, const
            ↪ ActivationFunctionKey &key)]: Error reading step
            ↪ threshold.");

      return std::shared_ptr<IActivationFunction<float>>(new
          ↪ InvertedStep32(threshold));
    }
  };


  /// <summary>
  /// Encapsulates the hyperbolic tangent activation function.
  /// </summary>
  class HyperbolicTangent32 : public IActivationFunction<float>
  {
  public:

    /// <summary>
    /// Default constructor. Initializes a hyperbolic tangent
    /// activation function of the form f(x) = tanh(x).
```

```
        /// </summary>
        HyperbolicTangent32() : HyperbolicTangent32(1.0, 1.0, 0.0, 0.0)
        { }

        /// <summary>
        /// Initializes a hyperbolic tangent activation function of the
            ↪ form
        /// f(x) = a * tanh(b * x + c) + d
        /// </summary>
        HyperbolicTangent32(float a, float b, float c, float d) :
          m_a(a), m_b(b), m_c(c), m_d(d), m_bDirtyCache(true) { }

        /// <summary>
        /// ID of the activation function.
        /// </summary>
        int getID() const override
        {
          return ActivationFunctionID::HYPERBOLIC_TANGENT_ID;
        }

        /// <summary>
        /// f(x) = A * tanh(B * x + C) + D
        /// </summary>
        float getA() const
        {
          return m_a;
        }
        /// <summary>
        /// f(x) = A * tanh(B * x + C) + D
        /// </summary>
        void setA(float value)
        {
          m_a = value;
          m_bDirtyCache = true;
        }


        /// <summary>
        /// f(x) = A * tanh(B * x + C) + D
        /// </summary>
        float getB() const
        {
          return m_b;
        }
        /// <summary>
        /// f(x) = A * tanh(B * x + C) + D
        /// </summary>
        void setB(float value)
        {
          m_b = value;
          m_bDirtyCache = true;
```

```
        }


        /// <summary>
        /// f(x) = A * tanh(B * x + C) + D
        /// </summary>
        float getC() const
        {
          return m_c;
        }
        /// <summary>
        /// f(x) = A * tanh(B * x + C) + D
        /// </summary>
        void setC(float value)
        {
          m_c = value;
          m_bDirtyCache = true;
        }


        /// <summary>
        /// f(x) = A * tanh(B * x + C) + D
        /// </summary>
        float getD() const
        {
          return m_d;
        }
        /// <summary>
        /// f(x) = A * tanh(B * x + C) + D
        /// </summary>
        void setD(float value)
        {
          m_d = value;
          m_bDirtyCache = true;
        }


        /// <summary>
        /// Creates new IActivationFunction<float> object which parameters
            ↪  are
        /// a copy of this activation function.
        /// </summary>
        std::shared_ptr<NeuralNetwork::ActivationFunctions::
            ↪ IActivationFunction<float>> CreateCopy() const override
        {
          return std::shared_ptr<IActivationFunction<float>>(new
              ↪ HyperbolicTangent32(getA(), getB(), getC(), getD()));
        }

        void df(float *result, const float *weighted_inputs, std::size_t
            ↪ size) override
```

```
    {

      if (m_bDirtyCache ||
        m_cache_weighted_input != weighted_inputs || m_cache_size !=
           ↪ size)
        updateCache(weighted_inputs, size);

      vsSqr((int)size, m_cache_tanh.data(), m_cache_scalar.data());
      float AB = getA() * getB();
      std::fill_n(result, size, AB);
      cblas_saxpy((int)size, -AB, m_cache_scalar.data(), 1, result, 1)
         ↪ ;

    }

    /// <summary>
    /// Activation function.
    /// </summary>
    void f(float *activations, const float *weighted_inputs, std::
       ↪ size_t size) override
    {
      updateCache(weighted_inputs, size);

      std::fill_n(activations, size, getD());
      cblas_saxpy((int)size, getA(), m_cache_tanh.data(), 1,
         ↪ activations, 1);

    }

    void Save(std::ostream &fNum, const ActivationFunctionKey &key)
       ↪ const override
    {
      fNum.write((char*)&this->m_a, sizeof(float));
      if (!fNum.good())
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
           ↪ ::HyperbolicTangent::Save(std::ostream &fNum, const
           ↪ ActivationFunctionKey &key)]: Error writing
           ↪ HyperbolicTangent parameter A.");

      fNum.write((char*)&this->m_b, sizeof(float));
      if (!fNum.good())
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
           ↪ ::HyperbolicTangent::Save(std::ostream &fNum, const
           ↪ ActivationFunctionKey &key)]: Error writing
           ↪ HyperbolicTangent parameter B.");

      fNum.write((char*)&this->m_c, sizeof(float));
      if (!fNum.good())
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
           ↪ ::HyperbolicTangent::Save(std::ostream &fNum, const
           ↪ ActivationFunctionKey &key)]: Error writing
```

```
            ↪ HyperbolicTangent parameter C.");

        fNum.write((char*)&this->m_d, sizeof(float));
        if (!fNum.good())
          throw std::runtime_error("[NeuralNetwork::ActivationFunctions
              ↪ ::HyperbolicTangent::Save(std::ostream &fNum, const
              ↪ ActivationFunctionKey &key)]: Error writing
              ↪ HyperbolicTangent parameter D.");
      }

      /// <summary>
      /// Loads a hyperbolic tangent function from stream and returns a
          ↪ pointer to it.
      /// </summary>
      static std::shared_ptr<NeuralNetwork::ActivationFunctions::
          ↪ IActivationFunction<float>> Load(std::istream &fNum, const
          ↪ ActivationFunctionKey &key)
      {
        float a, b, c, d;

        fNum.read((char*)&a, sizeof(float));
        if (!fNum.good() || fNum.gcount() != sizeof(float))
          throw std::runtime_error("[NeuralNetwork::ActivationFunctions
              ↪ ::HyperbolicTangent::Load(std::istream &fNum, const
              ↪ ActivationFunctionKey &key)]: Error reading
              ↪ HyperbolicTangent parameter A.");

        fNum.read((char*)&b, sizeof(float));
        if (!fNum.good() || fNum.gcount() != sizeof(float))
          throw std::runtime_error("[NeuralNetwork::ActivationFunctions
              ↪ ::HyperbolicTangent::Load(std::istream &fNum, const
              ↪ ActivationFunctionKey &key)]: Error reading
              ↪ HyperbolicTangent parameter B.");

        fNum.read((char*)&c, sizeof(float));
        if (!fNum.good() || fNum.gcount() != sizeof(float))
          throw std::runtime_error("[NeuralNetwork::ActivationFunctions
              ↪ ::HyperbolicTangent::Load(std::istream &fNum, const
              ↪ ActivationFunctionKey &key)]: Error reading
              ↪ HyperbolicTangent parameter C.");

        fNum.read((char*)&d, sizeof(float));
        if (!fNum.good() || fNum.gcount() != sizeof(float))
          throw std::runtime_error("[NeuralNetwork::ActivationFunctions
              ↪ ::HyperbolicTangent::Load(std::istream &fNum, const
              ↪ ActivationFunctionKey &key)]: Error reading
              ↪ HyperbolicTangent parameter D.");

        return std::shared_ptr<IActivationFunction<float>>(new
            ↪ HyperbolicTangent32(a, b, c, d));
      }
```

```
private:

  void updateCache(const float *weighted_inputs, std::size_t size)
  {
    if (m_cache_scalar.size() != size)
      m_cache_scalar.resize(size);

    m_cache_scalar.assign(size, getC());
    m_cache_tanh.resize(size);

    m_cache_weighted_input = weighted_inputs;
    m_cache_size = size;

    cblas_saxpy((int)size, getB(), weighted_inputs, 1,
       ↪ m_cache_scalar.data(), 1);
    vsTanh((int)size, m_cache_scalar.data(), m_cache_tanh.data());


    m_bDirtyCache = false;
  }

  bool m_bDirtyCache;
  const float *m_cache_weighted_input;
  std::size_t m_cache_size;
  std::vector<float> m_cache_scalar;
  std::vector<float> m_cache_tanh;

  float m_a;
  float m_b;
  float m_c;
  float m_d;
};


/// <summary>
/// Encapsulates a sigmoid activation function (see remarks).
/// </summary>
class Sigmoid32 : public IActivationFunction<float>
{
public:

  /// <summary>
  /// Default constructor. Initializes a sigmoid activation
  /// function of the form f(x) = 1 / (1 + e^-x).
  /// </summary>
  Sigmoid32() : Sigmoid32(1.0, -1.0, 0.0, 0.0)
  { }

  /// <summary>
  /// Initializes a sigmoid activation function of the form
```

```
        /// f(x) = a / (1 + e^(b * x + c)) + d
        /// </summary>
        Sigmoid32(float a, float b, float c, float d) :
          m_a(a), m_b(b), m_c(c), m_d(d), m_bDirtyCache(true)
        { }

        /// <summary>
        /// ID of the activation function.
        /// </summary>
        int getID() const override
        {
          return ActivationFunctionID::SIGMOID_ID;
        }

        /// <summary>
        /// f(x) = A / (1 + e^(B * x + C)) + D
        /// </summary>
        float getA() const
        {
          return m_a;
        }
        /// <summary>
        /// f(x) = A / (1 + e^(B * x + C)) + D
        /// </summary>
        void setA(float value)
        {
          m_a = value;
          m_bDirtyCache = true;
        }


        /// <summary>
        /// f(x) = A / (1 + e^(B * x + C)) + D
        /// </summary>
        float getB() const
        {
          return m_b;
        }
        /// <summary>
        /// f(x) = A / (1 + e^(B * x + C)) + D
        /// </summary>
        void setB(float value)
        {
          m_b = value;
          m_bDirtyCache = true;
        }


        /// <summary>
        /// f(x) = A / (1 + e^(B * x + C)) + D
        /// </summary>
```

```
        float getC() const
        {
          return m_c;
        }
        /// <summary>
        /// f(x) = A / (1 + e^(B * x + C)) + D
        /// </summary>
        void setC(float value)
        {
          m_c = value;
          m_bDirtyCache = true;
        }


        /// <summary>
        /// f(x) = A / (1 + e^(B * x + C)) + D
        /// </summary>
        float getD() const
        {
          return m_d;
        }
        /// <summary>
        /// f(x) = A / (1 + e^(B * x + C)) + D
        /// </summary>
        void setD(float value)
        {
          m_d = value;
          m_bDirtyCache = true;
        }


        /// <summary>
        /// Creates new IActivationFunction<float> object which parameters
            ↪  are
        /// a copy of this activation function.
        /// </summary>
        std::shared_ptr<NeuralNetwork::ActivationFunctions::
            ↪ IActivationFunction<float>> CreateCopy() const override
        {
          return std::shared_ptr<IActivationFunction<float>>(new Sigmoid32
              ↪ (getA(), getB(), getC(), getD()));
        }

        void df(float *result, const float *weighted_inputs, std::size_t
            ↪ size) override
        {
          if (m_bDirtyCache ||
            m_cache_weighted_input != weighted_inputs || m_cache_size !=
                ↪ size)
            updateCache(weighted_inputs, size);
```

```
    vsSqr((int)size, m_cache_exp.data(), m_cache_scalar.data());

    std::memset(result, 0, size * sizeof(float));
    cblas_saxpy((int)size, -getA() * getB(), m_cache_exp.data(), 1,
        ↪ result, 1);

    cblas_saxpy((int)size, getA() * getB(), m_cache_scalar.data(),
        ↪ 1, result, 1);

  }

  void f(float *activations, const float *weighted_inputs, std::
      ↪ size_t size) override
  {
    updateCache(weighted_inputs, size);
    std::fill_n(activations, size, getD());
    cblas_saxpy((int)size, getA(), m_cache_exp.data(), 1,
        ↪ activations, 1);

  }

  void Save(std::ostream &fNum, const ActivationFunctionKey &key)
      ↪ const override
  {
    fNum.write((char*)&this->m_a, sizeof(float));
    if (!fNum.good())
      throw std::runtime_error("[NeuralNetwork::ActivationFunctions
          ↪ ::Sigmoid::Save(std::ostream&fNum,␣const␣
          ↪ ActivationFunctionKey␣&key)]:␣Error␣writing␣Sigmoid␣
          ↪ parameter␣A.");

    fNum.write((char*)&this->m_b, sizeof(float));
    if (!fNum.good())
      throw std::runtime_error("[NeuralNetwork::ActivationFunctions
          ↪ ::Sigmoid::Save(std::ostream&fNum,␣const␣
          ↪ ActivationFunctionKey␣&key)]:␣Error␣writing␣Sigmoid␣
          ↪ parameter␣B.");

    fNum.write((char*)&this->m_c, sizeof(float));
    if (!fNum.good())
      throw std::runtime_error("[NeuralNetwork::ActivationFunctions
          ↪ ::Sigmoid::Save(std::ostream&fNum,␣const␣
          ↪ ActivationFunctionKey␣&key)]:␣Error␣writing␣Sigmoid␣
          ↪ parameter␣C.");

    fNum.write((char*)&this->m_d, sizeof(float));
    if (!fNum.good())
      throw std::runtime_error("[NeuralNetwork::ActivationFunctions
          ↪ ::Sigmoid::Save(std::ostream&fNum,␣const␣
          ↪ ActivationFunctionKey␣&key)]:␣Error␣writing␣Sigmoid␣
```

```
            ↪ parameter D.");
    }

    /// <summary>
    /// Loads a sigmoid function from stream and returns a pointer to
        ↪ it.
    /// </summary>
    static std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> Load(std::istream &fNum, const
        ↪ ActivationFunctionKey &key)
    {
      float a, b, c, d;

      fNum.read((char*)&a, sizeof(float));
      if (!fNum.good() || fNum.gcount() != sizeof(float))
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
            ↪ ::Sigmoid::Load(std::istream &fNum, const
            ↪ ActivationFunctionKey &key)]: Error reading
            ↪ HyperbolicTangent parameter A.");

      fNum.read((char*)&b, sizeof(float));
      if (!fNum.good() || fNum.gcount() != sizeof(float))
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
            ↪ ::Sigmoid::Load(std::istream &fNum, const
            ↪ ActivationFunctionKey &key)]: Error reading
            ↪ HyperbolicTangent parameter B.");

      fNum.read((char*)&c, sizeof(float));
      if (!fNum.good() || fNum.gcount() != sizeof(float))
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
            ↪ ::Sigmoid::Load(std::istream &fNum, const
            ↪ ActivationFunctionKey &key)]: Error reading
            ↪ HyperbolicTangent parameter C.");

      fNum.read((char*)&d, sizeof(float));
      if (!fNum.good() || fNum.gcount() != sizeof(float))
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
            ↪ ::Sigmoid::Load(std::istream &fNum, const
            ↪ ActivationFunctionKey &key)]: Error reading
            ↪ HyperbolicTangent parameter D.");

      return std::shared_ptr<IActivationFunction<float>>(new Sigmoid32
          ↪ (a, b, c, d));
    }

  private:

    void updateCache(const float *weighted_inputs, std::size_t size)
    {
      if (m_cache_scalar.size() != size)
        m_cache_scalar.resize(size);
```

```
      m_cache_exp.resize(size);

      m_cache_weighted_input = weighted_inputs;
      m_cache_size = size;

      m_cache_scalar.assign(size, getC());
      cblas_saxpy((int)size, getB(), weighted_inputs, 1,
         ↪ m_cache_scalar.data(), 1);
      vsExp((int)size, m_cache_scalar.data(), m_cache_exp.data());

      m_cache_scalar.assign(size, 1.0);
      cblas_saxpy((int)size, 1.0, m_cache_exp.data(), 1,
         ↪ m_cache_scalar.data(), 1);
      vsInv((int)size, m_cache_scalar.data(), m_cache_exp.data());


      m_bDirtyCache = false;
    }

    bool m_bDirtyCache;
    const float *m_cache_weighted_input;
    std::size_t m_cache_size;
    std::vector<float> m_cache_scalar;
    std::vector<float> m_cache_exp;

    float m_a;
    float m_b;
    float m_c;
    float m_d;
};


/// <summary>
/// Encapsulates the activation function for a rectified linear
/// unit (ReLU) (see remarks).
/// </summary>
class ReLU32 : public IActivationFunction<float>
{
public:

    ReLU32(float leak = 0.0) : Leak(leak)
    { }

    /// <summary>
    /// ID of the activation function.
    /// </summary>
    int getID() const override
    {
      return ActivationFunctionID::RELU_ID;
    }
```

```
        /// <summary>
        /// Creates new IActivationFunction<float> object which parameters
            ↪ are
        /// a copy of this activation function.
        /// </summary>
        std::shared_ptr<NeuralNetwork::ActivationFunctions::
            ↪ IActivationFunction<float>> CreateCopy() const override
        {
          return std::shared_ptr<IActivationFunction<float>>(new ReLU32(
              ↪ Leak));
        }

        /// <summary>
        /// Activation function.
        /// </summary>
        void f(float *activations, const float *weighted_inputs, std::
            ↪ size_t size) override
        {

          std::memcpy(activations, weighted_inputs, size * sizeof(float));
          #pragma omp parallel for
          for (int i = 0; i < (int)size; i++)
            if (activations[i] < 0)
              activations[i] *= Leak;
        }

        /// <summary>
        /// Value of the first derivative of the activation function.
        /// </summary>
        void df(float *result, const float *weighted_inputs, std::size_t
            ↪ size) override
        {

          #pragma omp parallel
          for (int i = 0; i < (int)size; i++)
            result[i] = (weighted_inputs[i] > 0.0f ? 1.0f : Leak);
        }

        void Save(std::ostream &fNum, const ActivationFunctionKey &key)
            ↪ const override
        {
          fNum.write((char*)&this->Leak, sizeof(float));
          if (!fNum.good())
            throw std::runtime_error("[NeuralNetwork::ActivationFunctions
                ↪ ::ReLU::Save(std::ostream &fNum, const
                ↪ ActivationFunctionKey &key)]: Error writing ReLU leak
                ↪ value.");
        }

        /// <summary>
        /// Loads a step function from stream and returns a pointer to it.
```

```
    /// </summary>
    static std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> Load(std::istream &fNum, const
        ↪ ActivationFunctionKey &key)
    {
      float leak;
      fNum.read((char*)&leak, sizeof(float));
      if (!fNum.good() || fNum.gcount() != sizeof(float))
        throw std::runtime_error("[NeuralNetwork::ActivationFunctions
            ↪ ::ReLU::Load(std::istream &fNum, const
            ↪ ActivationFunctionKey &key)]: Error reading step
            ↪ threshold.");

      return std::shared_ptr<IActivationFunction<float>>(new ReLU32(
          ↪ leak));
    }

    float Leak;
  };

  /// <summary>
  /// Encapsulates the softmax activation function (see remarks).
  /// </summary>
  class Softmax32 : public IActivationFunction<float>
  {
  public:

    Softmax32(std::size_t minibatch_size) :
      m_uiMinibatchSize(minibatch_size),
      m_bDirtyCache(true)
    {
      if (minibatch_size < 1)
        throw std::invalid_argument("Minibatch size must be at least
            ↪ 1.");
      cache_sum_exp.resize(minibatch_size, 0.0);
    }

    /// <summary>
    /// ID of the activation function.
    /// </summary>
    int getID() const override
    {
      return ActivationFunctionID::SOFTMAX_ID;
    }

    /// <summary>
    /// Creates new IActivationFunction<float> object which parameters
        ↪ are
    /// a copy of this activation function.
    /// </summary>
```

```
std::shared_ptr<NeuralNetwork::ActivationFunctions::
    ↪ IActivationFunction<float>> CreateCopy() const override
{
  return std::shared_ptr<IActivationFunction<float>>(new Softmax32
      ↪ (m_uiMinibatchSize));
}

/// <summary>
/// Activation function.
/// </summary>
void f(float *activations, const float *weighted_inputs, std::
    ↪ size_t size) override
{

  updateCache(weighted_inputs, size, size / m_uiMinibatchSize);

  vsExp((int)size, weighted_inputs, m_cache_scalar.data());
  std::memset(activations, 0, size * sizeof(float));
  #pragma omp parallel for
  for (int sample_i = 0; sample_i < (int)m_uiMinibatchSize;
      ↪ sample_i++)
    cblas_saxpy((int)m_sample_size, 1.0f / cache_sum_exp[sample_i
        ↪ ],
      m_cache_scalar.data() + sample_i * m_sample_size, 1,
      activations + sample_i * m_sample_size, 1);

}

/// <summary>
/// Value of the first derivative of the activation function.
/// </summary>
void df(float *result, const float *weighted_inputs, std::size_t
    ↪ size) override
{
  if (m_bDirtyCache ||
    weighted_inputs != m_weighted_inputs || size != m_size)
    updateCache(weighted_inputs, size, size / m_uiMinibatchSize);


  #pragma omp parallel for
  for (int sample_i = 0; sample_i < (int)m_uiMinibatchSize;
      ↪ sample_i++)
    for (int i = 0; i < (int)m_sample_size; i++)
    {
      float e_zi = std::exp(weighted_inputs[i + sample_i *
          ↪ m_sample_size]);
      result[i + sample_i * m_sample_size] = e_zi * (cache_sum_exp
          ↪ [sample_i] - e_zi) /
                    (cache_sum_exp[sample_i] * cache_sum_exp[
                        ↪ sample_i]);
    }
```

```
    }

    void Save(std::ostream &fNum, const ActivationFunctionKey &key)
        ↪ const override
    { }

    /// <summary>
    /// Loads a softmax function from stream and returns a pointer to
        ↪ it.
    /// </summary>
    static std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> Load(std::istream &fNum, std::
        ↪ size_t minibatch_size, const ActivationFunctionKey &key)
    {
      return std::shared_ptr<IActivationFunction<float>>(new Softmax32
          ↪ (minibatch_size));
    }

  private:
    void updateCache(const float *weighted_inputs, std::size_t size,
        ↪ std::size_t sample_size)
    {
      m_cache_scalar.resize(size);

      m_weighted_inputs = weighted_inputs;
      m_sample_size = sample_size;
      m_size = size;

      vsExp((int)(sample_size * m_uiMinibatchSize), weighted_inputs,
          ↪ m_cache_scalar.data());

      #pragma omp parallel for
      for (int sample_i = 0; sample_i < (int)m_uiMinibatchSize;
          ↪ sample_i++)
      {
        cache_sum_exp[sample_i] = cblas_sasum((int)sample_size,
            ↪ m_cache_scalar.data() + sample_i * sample_size, 1);
      }

      m_bDirtyCache = false;

    }

    bool m_bDirtyCache;
    const float *m_weighted_inputs;
    std::size_t m_size;
    std::size_t m_sample_size;
    std::size_t m_uiMinibatchSize;
    std::vector<float> m_cache_scalar;
    std::vector<float> cache_sum_exp;
  };
```

```
    typedef Step32 Step;
    typedef InvertedStep32 InvertedStep;
    typedef HyperbolicTangent32 HyperbolicTangent;
    typedef Sigmoid32 Sigmoid;
    typedef Identity32 Identity;
    typedef Softmax32 Softmax;
    typedef ReLU32 ReLU;
  }
}

#endif
```

B.11 File: kfuncobj.h

```
#ifndef K_OBJECTIVE_FUNCTIONS_H
#define K_OBJECTIVE_FUNCTIONS_H

#include <memory>
#include <vector>
#include <stdexcept>
#include <sstream>
#include <cmath>
#include <algorithm>

#include "mkl.h"

#include "ktrainer.h"
#include "ktypeutils.h"


namespace NeuralNetwork
{
  namespace ObjectiveFunctions
  {
    /// <summary>
    /// Encapsulates the Mean Square Error objective function.
    /// </summary>
    class MSE32 : public NeuralNetwork::Trainer32::IObjectiveFunction
    {
    public:
      /// <summary>
      /// Computes the error value of the objective function
      /// based on the input values and the target values.
      /// </summary>
      float error(const float *values, const float *target, std::size_t
          ↪ size) override
      {
```

```
      float retval = 0.0;
      #pragma omp parallel for reduction(+:retval)
      for (int i = 0; i < (int)size; i++)
      {
        float tmp = values[i] - target[i];
        retval += (tmp * tmp);
      }

      return retval;
    }

    /// <summary>
    /// Returns the partial derivative of the objective function
    /// with respect to the objective function's input vector's
    /// element index-th.
    /// </summary>
    void derror(float *result, const float *values, const float *
        ↪ target, std::size_t size) override
    {
      vsSub((int)size, values, target, result);
    }
  };


  /// <summary>
  /// Encapsulates the Cross Entropy objective function.
  /// </summary>
  class CrossEntropy32 : public NeuralNetwork::Trainer32::
      ↪ IObjectiveFunction
  {
  public:
    /// <summary>
    /// Computes the error value of the objective function
    /// based on the input values and the target values.
    /// </summary>
    float error(const float *values, const float *target, std::size_t
        ↪ size) override
    {
      float retval = 0.0;
      #pragma omp parallel for reduction(+:retval)
      for (int i = 0; i < (int)size; i++)
      {
        if (target[i] != 1 || values[i] != 1)
          retval += (target[i] * std::log(values[i]) + (1 - target[i])
              ↪  * std::log(1 - values[i]));
      }

      return -retval;
    }

    /// <summary>
```

```
      /// Returns the partial derivative of the objective function
      /// with respect to the objective function's input vector's
      /// element index-th.
      /// </summary>
      void derror(float *result, const float *values, const float *
          ↪ target, std::size_t size) override
      {
        #pragma omp parallel for
        for (int i = 0; i < (int)size; i++)
          result[i] = (values[i] - target[i]) /
                (values[i] * (1 - values[i]));
      }
    };

    typedef MSE32 MSE;
    typedef CrossEntropy32 CrossEntropy;
  }
}

#endif
```

B.12 File: kinputlayer.h

```
#ifndef K_FFANN_INPUT_LAYER_H
#define K_FFANN_INPUT_LAYER_H

#include <cstring>

#include "knnkeys.h"
#include "klayer.h"
#include "kfuncact.h"
#include "ktypeutils.h"


namespace NeuralNetwork
{
  class Network32;

  class InputLayer32 : public Layer32
  {
  public:

    static const int LAYER_TYPE = 401;

    InputLayer32(const Network32 &network, std::size_t numInputs, std::
        ↪ size_t uiMinibatchSize, const LayerKey &key) :
      Layer32(network, numInputs, NeuralNetwork::ActivationFunctions::
          ↪ Identity32(), uiMinibatchSize, key)
```

```
    {
      m_weights.clear();
      m_neurons.resize(numInputs);
      for (std::size_t i = 0; i < numInputs; i++)
        m_neurons[i] = std::shared_ptr<Neuron32>(
          new Neuron32(m_activations, m_biases, m_weighted_input,
              ↪ m_deltas,
            m_deltas, 0, i, numInputs, uiMinibatchSize, m_NeuronKey)
          );
    }


    int getLayerType() const override
    {
      return LAYER_TYPE;
    }

    /// <summary>
    /// Returns maximum size_t (equivalent to -1) since input layers
       ↪ have no previous layer.
    /// </summary>
    std::size_t getPreviousLayerIndex() const override
    {
      return (std::size_t)(-1);
    }

    /// <summary>
    /// Initializes the inputs of the input layer for
    /// a whole minibatch to the specified vector.
    /// </summary>
    void InitializeInput(const float *arr)
    {
      std::memcpy(m_activations.data(), arr, m_activations.size() *
          ↪ sizeof(float));
    }

    /// <summary>
    /// Initializes the inputs of the input layer for
    /// a specific sample of a minibatch to the specified vector.
    /// </summary>
    void InitializeInput(const float *arr, std::size_t sample)
    {
      std::memcpy(m_activations.data() + sample * getNeuronCount(), arr,
          ↪  getNeuronCount() * sizeof(float));
    }

    /// <summary>
    /// Initializes the inputs of the input layer for
    /// a whole minibatch to the specified vector.
    /// </summary>
    void InitializeInput(const std::vector<float> &input)
```

```
    {
      InitializeInput(input.data());
    }

    void setBiases(const float *arr) override
    {

      Layer32::setBiases(arr);

      InitializeInput(arr, 0);
    }


    void Propagate() override
    {
    }
  };

  typedef InputLayer32 InputLayer;
}

#endif
```

B.13 File: klayer.h

```
#ifndef K_FFANN_LAYER_H
#define K_FFANN_LAYER_H

#include <vector>
#include <memory>

#include "ksafevector.h"
#include "knnkeys.h"

#include "kneuron.h"

#include "ktypeutils.h"


namespace NeuralNetwork
{
  class Network32;

  namespace ActivationFunctions
  {
    /// <summary>
    /// Interface for Activation function for a neuron.
    /// </summary>
```

```
    template <class real_t>
    class IActivationFunction
    {
    public:

      IActivationFunction()
      { }

      virtual ~IActivationFunction()
      { }

      /// <summary>
      /// Activation functions cannot be copied by default methods.
      /// Use <see cref="IActivationFunction::CreateCopy"/> instead.
      /// </summary>
      IActivationFunction<real_t>(const IActivationFunction<real_t> &) =
          ↪  delete;
      IActivationFunction<real_t>& operator=(const IActivationFunction<
          ↪ real_t>&) = delete;

      /// <summary>
      /// ID of the activation function.
      /// </summary>
      virtual int getID() const = 0;

      /// <summary>
      /// Creates new IActivationFunction object which parameters are
      /// a copy of this activation function.
      /// </summary>
      virtual std::shared_ptr<NeuralNetwork::ActivationFunctions::
          ↪ IActivationFunction<real_t>> CreateCopy() const = 0;

      /// <summary>
      /// Activation function.
      /// </summary>
      virtual void f(real_t *activations, const real_t *weighted_inputs,
          ↪  std::size_t size) = 0;
      /// <summary>
      /// Value of the first derivative of the activation function.
      /// </summary>
      virtual void df(real_t *result, const real_t *weighted_inputs, std
          ↪ ::size_t size) = 0;

      virtual void Save(std::ostream &fNum, const ActivationFunctionKey
          ↪ &key) const = 0;
    };
}


class Layer32
{
```

```
public:

  /// <summary>
  /// Initializes the base components of the Layer32 class.
  /// </summary>
  Layer32(const Network32 &network, std::size_t numNeurons,
    const NeuralNetwork::ActivationFunctions::IActivationFunction<
        ↪ float> &f,
    std::size_t uiMinibatchSize, const LayerKey &key) :
    m_activations(numNeurons * uiMinibatchSize),
    m_biases(numNeurons),
    m_weighted_input(numNeurons * uiMinibatchSize),
    m_deltas(numNeurons * uiMinibatchSize),
    m_uiMinibatchSize(uiMinibatchSize)
  {
    setActivationFunction(f);
  }

  Layer32(const Layer32&) = delete;
  Layer32& operator=(const Layer32&) = delete;

  virtual ~Layer32()
  { }


  virtual int getLayerType() const = 0;

  /// <summary>
  /// Returns whether this layer is trainable or not.
  /// </summary>
  virtual bool IsTrainable() const
  {
    return false;
  }

  /// <summary>
  /// Returns a reference to the activation function for this neuron.
  /// </summary>
  ActivationFunctions::IActivationFunction<float> &
     ↪ getActivationFunction() const
  {
    return *m_pfuncActivation;
  }
  /// <summary>
  /// Replaces this neuron's activation function by a copy of
  /// the specified activation function.
  /// </summary>
  void setActivationFunction(const ActivationFunctions::
     ↪ IActivationFunction<float> &f)
  {
    m_pfuncActivation = f.CreateCopy();
```

```
    }

    /// <summary>
    /// Index of this layer inside its network.
    /// </summary>
    std::size_t getNetworkIndex() const
    {
      return m_uiNetworkIndex;
    }

    /// <summary>
    /// ID of this layer inside its network. Alias for <see cref="
        ↪ Layer32.NetworkIndex"/>.
    /// </summary>
    std::size_t getID() const
    {
      return getNetworkIndex();
    }

    /// <summary>
    /// Sets the index of this layer inside its network.
    /// Only objects with the key can call this method.
    /// </summary>
    void setNetworkIndex(std::size_t value, const LayerKey &key)
    {
      setNetworkIndex(value);
    }

    /// <summary>
    /// Returns the index of the preceeding layer inside
    /// its network.
    /// </summary>
    virtual std::size_t getPreviousLayerIndex() const = 0;

    /// <summary>
    /// Returns a constant array pointing to the
    /// activations for all the neurons in this layer
    /// for the specified sample.
    /// </summary>
    const float *getActivations(std::size_t sample = 0) const
    {
      return m_activations.data() + sample * getNeuronCount();
    }

    const std::vector<float>& getActivationsVector() const
    {
      return m_activations;
    }

    /// <summary>
    /// Returns a constant array pointing to the
```

```
    /// biases for all the neurons in this layer.
    /// </summary>
    const float *getBiases() const
    {
      return m_biases.data();
    }

    /// <summary>
    /// Sets all the biases for the neurons in this layer.
    /// </summary>
    virtual void setBiases(const float *arr)
    {
      m_biases.assign(m_biases.size(), arr);
    }

    /// <summary>
    /// Returns a constant array pointing to the weighted input
    /// values for all the neurons in this layer.
    /// </summary>
    const float *getWeightedInputValues(std::size_t sample = 0) const
    {
      return m_weighted_input.data() + sample * getNeuronCount();
    }

    const std::vector<float>& getWeightedInputVector() const
    {
      return m_weighted_input;
    }

    /// <summary>
    /// Returns a constant array pointing to the delta
    /// values for all the neurons in this layer.
    /// </summary>
    const float *getDeltas(std::size_t sample = 0) const
    {
      return m_deltas.data() + sample * getNeuronCount();
    }

    /// <summary>
    /// Allows access to the array of deltas contained in this layer.
    /// </summary>
    KUtilities::atomic_vector<float>& getDeltaVector()
    {
      return m_deltas;
    }

    /// <summary>
    /// Allows access to the array of deltas contained in this layer.
    /// </summary>
    const KUtilities::atomic_vector<float>& getDeltaVector() const
    {
```

```
    return m_deltas;
  }


  /// <summary>
  /// Returns the number of neurons contained in this layer.
  /// </summary>
  virtual std::size_t getNeuronCount() const
  {
    return m_neurons.size();
  }

  /// <summary>
  /// Returns a reference to the specified neuron.
  /// </summary>
  virtual Neuron32 &getNeuron(std::size_t index) const
  {
    return *m_neurons.at(index);
  }

  /// <summary>
  /// Returns the number of connections between this layer and the
    ↪ previous layer.
  /// </summary>
  std::size_t getConnectionsCount() const
  {
    return getWeightsCount();
  }

  /// <summary>
  /// Returns the weights in this layer. This is equivalent to the
    ↪ number of
  /// connections between this layer and the previous layer.
  /// </summary>
  std::size_t getWeightsCount() const
  {
    return m_weights.size();
  }

  /// <summary>
  /// Allows access to the array of weights contained in this layer.
  /// </summary>
  KUtilities::atomic_vector<float>& getWeightsVector()
  {
    return m_weights;
  }

  /// <summary>
  /// Allows access to the array of weights contained in this layer.
  /// </summary>
  const KUtilities::atomic_vector<float>& getWeightsVector() const
```

```
    {
      return m_weights;
    }

    /// <summary>
    /// Sets all the weights for the neurons in this layer.
    /// </summary>
    void setWeights(const float *arr)
    {
      m_weights.assign(m_weights.size(), arr);
    }

    /// <summary>
    /// Allows access to the array of weights contained in this layer.
    /// nullptr if no weights.
    /// </summary>
    const float *getWeights() const
    {
      return (m_weights.size() > 0 ? m_weights.data() : nullptr);
    }

    virtual void Propagate() = 0;


  protected:
    /// <summary>
    /// Sets the index of this layer inside its network.
    /// </summary>
    virtual void setNetworkIndex(std::size_t value)
    {
      m_uiNetworkIndex = value;
    }

    std::vector<float> m_activations;
    std::vector<float> m_weighted_input;

    KUtilities::atomic_vector<float> m_biases;
    KUtilities::atomic_vector<float> m_deltas;

    std::size_t m_uiMinibatchSize;

    KUtilities::atomic_vector<float> m_weights;

    /// <summary>
    /// Array of neurons for this layer.
    /// </summary>
    std::vector<std::shared_ptr<Neuron32>> m_neurons;

    std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>> m_pfuncActivation;
```

```
  NeuronKey m_NeuronKey;

private:

  std::size_t m_uiNetworkIndex;

  bool m_bLocked;
};


/// <summary>
/// Interface for trainable layers.
/// </summary>
class TrainableLayer32 : public Layer32
{
public:
  /// <summary>
  /// Returns whether this layer is trainable or not.
  /// </summary>
  bool IsTrainable() const override
  {
    return true;
  }

  /// <summary>
  /// Performs backpropagation by training weights
  /// of connections between fromLayer and this layer.
  /// </summary>
  virtual void Backpropagate(NeuralNetwork::Layer32 &fromLayer) = 0;

  /// <summary>
  /// Performs post backpropagation operation of updating
  /// the actual weights of the layer neurons by the trained value.
  /// </summary>
  virtual void PostBackpropagate(float fInvMinibatchSize, float
    ↪ fInvTotalSamples) = 0;

  /// <summary>
  /// Initializes the base components of the Layer32 class.
  /// </summary>
  TrainableLayer32(const Network32 &network, std::size_t numNeurons,
    const NeuralNetwork::ActivationFunctions::IActivationFunction<
      ↪ float> &f,
    std::size_t uiMinibatchSize, const LayerKey &key) :
    Layer32(network, numNeurons, f, uiMinibatchSize, key)
  { }

  TrainableLayer32(const TrainableLayer32&) = delete;
  TrainableLayer32& operator=(const TrainableLayer32&) = delete;

  virtual ~TrainableLayer32()
```

```
    { }
  };

  typedef Layer32 Layer;
  typedef TrainableLayer32 TrainableLayer;
}

#endif
```

B.14 File: kloader.h

```
#ifndef K_FFANN_LOADER_H
#define K_FFANN_LOADER_H

#include <istream>
#include <ostream>

#include "knetwork.h"
#include "knnkeys.h"
#include "ktypeutils.h"


namespace NeuralNetwork
{
  class Loader32
  {
  public:

    static const int VERSION = 1;

    /// <summary>
    /// Loads a network previously saved by Save() from the specified
       ↪ file.
    /// </summary>
    static bool Load(const char *sFilename, Network32 &network);
    /// <summary>
    /// Loads a network previously saved by Save() from the specified
       ↪ stream.
    /// </summary>
    static void Load(std::istream &fNum, Network32 &network);

    /// <summary>
    /// Saves a network previously to the specified file.
    /// </summary>
    static void Save(const char *sFilename, const Network32 &network);
    /// <summary>
    /// Saves a network previously to the specified stream.
    /// </summary>
```

```
    static void Save(std::ostream &fNum, const Network32 &network);
  };

  typedef Loader32 Loader;
}

#endif
```

B.15 File: kloader.cpp

```
#include <fstream>
#include <stdexcept>

#include "kloader.h"

bool NeuralNetwork::Loader32::Load(const char * sFilename, Network32 &
    ↪ network)
{
  bool retval = true;
  std::ifstream fNumIn;

  try
  {
    fNumIn.open(sFilename, std::ofstream::in | std::ofstream::binary);
    if (!fNumIn.is_open())
      throw std::runtime_error("Unable to open specified file.");
  }
  catch (...)
  {
    if (fNumIn.is_open())
      fNumIn.close();

    retval = false;
  }

  if (retval)
    Load(fNumIn, network);

  if (fNumIn.is_open())
    fNumIn.close();

  return retval;
}

void NeuralNetwork::Loader32::Load(std::istream & fNum, Network32 &
    ↪ network)
{
  network.Load(fNum, { });
```

```
}

void NeuralNetwork::Loader32::Save(const char * sFilename, const
    ↪ Network32 & network)
{
  std::ofstream fNumOut;

  fNumOut.open(sFilename, std::ofstream::out | std::ofstream::trunc |
      ↪ std::ofstream::binary);
  if (!fNumOut.is_open())
    throw std::runtime_error("Unable to open specified file.");

  try
  {
    Save(fNumOut, network);
  }
  catch (...)
  {
    if (fNumOut.is_open())
      fNumOut.close();

    throw;
  }

  if (fNumOut.is_open())
    fNumOut.close();
}

void NeuralNetwork::Loader32::Save(std::ostream & fNum, const Network32
    ↪ & network)
{
  network.Save(fNum, {});
}
```

B.16 File: kmaxpoollayer.h

```
#ifndef K_FFANN_MAXPOOL_LAYER_H
#define K_FFANN_MAXPOOL_LAYER_H

#include <vector>
#include <stdexcept>

#include "knnkeys.h"
#include "kconvlayer.h"
#include "ktypeutils.h"


namespace NeuralNetwork
```

```
{
  class Network32;

  namespace Convolutional
  {
    class MaxPoolLayer32 : public ConvolutionalLayer32
    {
    public:

      static const int LAYER_TYPE = 407;


      /// <summary>
      /// Initializes a new maxpool layer.
      /// </summary>
      MaxPoolLayer32(const Network32 &network,
        std::size_t uiPoolingWidth, std::size_t uiPoolingHeight, std::
            ↪ size_t uiPoolingStep,
        const NeuralNetwork::ActivationFunctions::IActivationFunction<
            ↪ float> &f,
        std::size_t uiInputWidth, std::size_t uiInputHeight, std::size_t
            ↪  uiInputDepth,
        const Layer32 &prevLayer, std::size_t uiMinibatchSize, const
            ↪ LayerKey &key);

      /// <summary>
      /// Initializes a new maxpool layer using another convolutional
          ↪ layer as input layer.
      /// </summary>
      MaxPoolLayer32(const Network32 &network,
        std::size_t uiPoolingWidth, std::size_t uiPoolingHeight, std::
            ↪ size_t uiPoolingStep,
        const NeuralNetwork::ActivationFunctions::IActivationFunction<
            ↪ float> &f,
        const ConvolutionalLayer32 &prevLayer, std::size_t
            ↪ uiMinibatchSize, const LayerKey &key) :
        MaxPoolLayer32(network,
          uiPoolingWidth, uiPoolingHeight, uiPoolingStep,
          f,
          prevLayer.getOutputWidth(), prevLayer.getOutputHeight(),
              ↪ prevLayer.getOutputDepth(),
          prevLayer, uiMinibatchSize, key) { }

      MaxPoolLayer32(const MaxPoolLayer32&) = delete;
      MaxPoolLayer32& operator=(const MaxPoolLayer32&) = delete;

      virtual ~MaxPoolLayer32()
      { }


      int getLayerType() const override
```

```
    {
      return LAYER_TYPE;
    }

    /// <summary>
    /// Returns the index of the preceeding layer inside
    /// its network.
    /// </summary>
    std::size_t getPreviousLayerIndex() const override
    {
      return m_prevLayer.getNetworkIndex();
    }

    void Propagate() override;

    void Backpropagate(Layer32 &fromLayer) override;

    void PostBackpropagate(float fInvMinibatchSize, float
        ↪ fInvTotalSamples) override
    {
    }

    /// <summary>
    /// Returns the array of indices where the maximum for the
        ↪ weighted input value was found.
    /// </summary>
    const std::size_t *getWeightedInputIndices(std::size_t sample = 0)
        ↪  const
    {
      return m_weighted_input_index.data() + sample * this->
          ↪ getNeuronCount();
    }

    void Save(std::ostream &fNum, const LayerKey &key) const;
    static std::shared_ptr<MaxPoolLayer32> Load(std::istream &fNum,
        ↪ const Network32 &network, std::size_t uiMinibatchSize,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f, const Layer32 &prevLayer, const LayerKey &key);

  protected:
    /// <summary>
    /// Returns the array of indices where the maximum for the
        ↪ weighted input value was found.
    /// </summary>
    std::size_t *getWeightedInputIndices(std::size_t sample = 0)
    {
      return m_weighted_input_index.data() + sample * this->
          ↪ getNeuronCount();
    }
```

```
    private:
      std::vector<std::size_t> m_weighted_input_index;
    };


    typedef MaxPoolLayer32 MaxPoolLayer;
  }
}

#endif
```

B.17 File: kmaxpoollayer.cpp

```
#include "kinputlayer.h"
#include "kmaxpoollayer.h"
#include "kconvneuron.h"

NeuralNetwork::Convolutional::MaxPoolLayer32::MaxPoolLayer32(const
    ↪ Network32 &network,
  std::size_t uiPoolingWidth, std::size_t uiPoolingHeight, std::size_t
      ↪ uiPoolingStep,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<float> &
      ↪ f,
  std::size_t uiInputWidth, std::size_t uiInputHeight, std::size_t
      ↪ uiInputDepth,
  const Layer32 &prevLayer, std::size_t uiMinibatchSize, const LayerKey
      ↪ &key) :
  ConvolutionalLayer32(network, f, prevLayer, uiMinibatchSize, key)
{

  std::vector<std::shared_ptr<Filter32>> filters;
  std::vector<float> elements;
  filters.resize(uiInputDepth);
  for (std::size_t i = 0; i < uiInputDepth; i++)
  {
    Filter32::depth_index_type j = i;
    filters[i] = std::shared_ptr<InputFilter32>(new InputFilter32(
        ↪ uiPoolingWidth, uiPoolingHeight,
      &j,
      1));
    elements.resize(filters[i]->getFilterElementsCount(), 1.0);
    filters[i]->setFilterElements(elements.data());
    filters[i]->setBias(0);
  }

  std::vector<std::size_t> arrFeatureSteps(uiInputDepth, uiPoolingStep);
  init(filters.data(), arrFeatureSteps.data(), arrFeatureSteps.size(),
    uiInputWidth, uiInputHeight, uiInputDepth);
```

```
  m_weighted_input_index.resize(m_weighted_input.size());
}

void NeuralNetwork::Convolutional::MaxPoolLayer32::Propagate()
{
  for (std::size_t sample_i = 0; sample_i < m_uiMinibatchSize; sample_i
      ↪ ++)
  {
    const float *prevActivations = m_prevLayer.getActivations() +
        ↪ sample_i * m_prevLayer.getNeuronCount();
    std::size_t *weighted_input_indices = getWeightedInputIndices(
        ↪ sample_i);
    #pragma omp parallel for
    for (int i = 0; i < getNeuronCount(); i++)
    {
      ConvolutionalNeuron32 &neuron = (ConvolutionalNeuron32&)(getNeuron
          ↪ (i));
      const int32_t * neuron_from_indices = neuron.getNeuronFrom();
      float maxValue = 0.0f;
      if (neuron.getConnectionsInCount() > 0)
      {
        maxValue = prevActivations[neuron_from_indices[0]];
        weighted_input_indices[i] = neuron_from_indices[0];
      }
      for (int connection_in_i = 1; connection_in_i < neuron.
          ↪ getConnectionsInCount(); connection_in_i++)
      {
        std::size_t neuron_from_i = neuron_from_indices[connection_in_i
            ↪ ];
        if (maxValue < prevActivations[neuron_from_i])
        {
          maxValue = prevActivations[neuron_from_i];
          weighted_input_indices[i] = neuron_from_i;
        }
      }

      neuron.setWeightedInput(maxValue, sample_i);
    }
  }

  m_pfuncActivation->f(m_activations.data(), m_weighted_input.data(),
      ↪ m_weighted_input.size());
}

void NeuralNetwork::Convolutional::MaxPoolLayer32::Backpropagate(Layer32
    ↪  & fromLayer)
{
  if (fromLayer.getLayerType() != NeuralNetwork::InputLayer32::
      ↪ LAYER_TYPE)
  {
```

```
    std::memset(fromLayer.getDeltaVector().unsafe_data(), 0, fromLayer.
        ↪ getDeltaVector().size() * sizeof(float));
    for (std::size_t sample_i = 0; sample_i < m_uiMinibatchSize;
        ↪ sample_i++)
    {
      std::size_t *weighted_input_indices = this->
          ↪ getWeightedInputIndices(sample_i);
      #pragma omp parallel for
      for (int neuron_to_i = 0; neuron_to_i < (int)this->getNeuronCount
          ↪ (); neuron_to_i++)
      {
        NeuralNetwork::Convolutional::ConvolutionalNeuron32 &neuron_to =
          (NeuralNetwork::Convolutional::ConvolutionalNeuron32&)this->
              ↪ getNeuron(neuron_to_i);
        const int32_t *neuron_from_indices = neuron_to.getNeuronFrom();

        std::size_t from_sample_start = sample_i * fromLayer.
            ↪ getNeuronCount();
        for (std::size_t connection_index = 0; connection_index <
            ↪ neuron_to.getConnectionsInCount(); connection_index++)
        {
          std::size_t neuron_from_i = neuron_from_indices[
              ↪ connection_index];
          if (weighted_input_indices[neuron_to_i] == neuron_from_i)
            fromLayer.getDeltaVector().accumulate(from_sample_start +
                ↪ neuron_from_i, neuron_to.getDelta(sample_i));

        }
      }
    }
  }

  m_buffer.resize(fromLayer.getWeightedInputVector().size());
  fromLayer.getActivationFunction().df(m_buffer.data(), fromLayer.
      ↪ getWeightedInputValues(), m_buffer.size());
  vsMul((int)m_buffer.size(), fromLayer.getDeltaVector().data(),
      ↪ m_buffer.data(), fromLayer.getDeltaVector().unsafe_data());
}

void NeuralNetwork::Convolutional::MaxPoolLayer32::Save(std::ostream &
    ↪ fNum, const LayerKey & key) const
{
  std::uint64_t u_output;

  u_output = KUtilities::UInt64::getLittleEndian(this->getFeature(0).
      ↪ getFilter()->getWidth());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ MaxPoolLayer32::Save(std::ostream &fNum, const LayerKey &key
        ↪ )]: "
```

```
      "Error writing pooling width.");

  u_output = KUtilities::UInt64::getLittleEndian(this->getFeature(0).
     ↪ getFilter()->getHeight());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
       ↪ MaxPoolLayer32::Save(std::ostream &fNum, const LayerKey &key
       ↪ )]: "
      "Error writing pooling height.");

  u_output = KUtilities::UInt64::getLittleEndian(this->getFeature(0).
     ↪ getStep());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
       ↪ MaxPoolLayer32::Save(std::ostream &fNum, const LayerKey &key
       ↪ )]: "
      "Error writing pooling step.");


  u_output = KUtilities::UInt64::getLittleEndian(this->getInputWidth());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
       ↪ MaxPoolLayer32::Save(std::ostream &fNum, const LayerKey &key
       ↪ )]: "
      "Error writing input width.");

  u_output = KUtilities::UInt64::getLittleEndian(this->getInputHeight())
     ↪ ;
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
       ↪ MaxPoolLayer32::Save(std::ostream &fNum, const LayerKey &key
       ↪ )]: "
      "Error writing input height.");

  u_output = KUtilities::UInt64::getLittleEndian(this->getInputDepth());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Convolutional::
       ↪ MaxPoolLayer32::Save(std::ostream &fNum, const LayerKey &key
       ↪ )]: "
      "Error writing input depth.");
}

std::shared_ptr<NeuralNetwork::Convolutional::MaxPoolLayer32>
NeuralNetwork::Convolutional::MaxPoolLayer32::Load(std::istream & fNum,
   ↪ const Network32 &network, std::size_t uiMinibatchSize,
```

```
    const NeuralNetwork::ActivationFunctions::IActivationFunction<float> &
        ↪ f, const Layer32 &prevLayer, const LayerKey & key)
{
    std::size_t uiPoolingWidth, uiPoolingHeight, uiPoolingStep,
      uiInputWidth, uiInputHeight, uiInputDepth;

    char buffer[8];

    fNum.read(buffer, sizeof(std::uint64_t));
    if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
      throw std::runtime_error("[NeuralNetwork::Convolutional::
          ↪ MaxPoolLayer32::Load(istream &fNum, const Network32 &network,
          ↪ size_t uiMinibatchSize, const NeuralNetwork::
          ↪ ActivationFunctions::IActivationFunction<float> &f, const
          ↪ Layer32 &prevLayer, const LayerKey &key)]: "
        "Error reading pooling width.");
    uiPoolingWidth = KUtilities::UInt64::ToEndian(buffer, true);

    fNum.read(buffer, sizeof(std::uint64_t));
    if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
      throw std::runtime_error("[NeuralNetwork::Convolutional::
          ↪ MaxPoolLayer32::Load(istream &fNum, const Network32 &network,
          ↪ size_t uiMinibatchSize, const NeuralNetwork::
          ↪ ActivationFunctions::IActivationFunction<float> &f, const
          ↪ Layer32 &prevLayer, const LayerKey &key)]: "
        "Error reading pooling height.");
    uiPoolingHeight = KUtilities::UInt64::ToEndian(buffer, true);

    fNum.read(buffer, sizeof(std::uint64_t));
    if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
      throw std::runtime_error("[NeuralNetwork::Convolutional::
          ↪ MaxPoolLayer32::Load(istream &fNum, const Network32 &network,
          ↪ size_t uiMinibatchSize, const NeuralNetwork::
          ↪ ActivationFunctions::IActivationFunction<float> &f, const
          ↪ Layer32 &prevLayer, const LayerKey &key)]: "
        "Error reading pooling step.");
    uiPoolingStep = KUtilities::UInt64::ToEndian(buffer, true);



    fNum.read(buffer, sizeof(std::uint64_t));
    if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
      throw std::runtime_error("[NeuralNetwork::Convolutional::
          ↪ MaxPoolLayer32::Load(istream &fNum, const Network32 &network,
          ↪ size_t uiMinibatchSize, const NeuralNetwork::
          ↪ ActivationFunctions::IActivationFunction<float> &f, const
          ↪ Layer32 &prevLayer, const LayerKey &key)]: "
        "Error reading input width.");
    uiInputWidth = KUtilities::UInt64::ToEndian(buffer, true);

    fNum.read(buffer, sizeof(std::uint64_t));
```

```
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ MaxPoolLayer32::Load(istream &fNum, const Network32 &network,
        ↪ size_t uiMinibatchSize, const NeuralNetwork::
        ↪ ActivationFunctions::IActivationFunction<float> &f, const
        ↪ Layer32 &prevLayer, const LayerKey &key)]: "
      "Error reading input height.");
  uiInputHeight = KUtilities::UInt64::ToEndian(buffer, true);

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Convolutional::
        ↪ MaxPoolLayer32::Load(istream &fNum, const Network32 &network,
        ↪ size_t uiMinibatchSize, const NeuralNetwork::
        ↪ ActivationFunctions::IActivationFunction<float> &f, const
        ↪ Layer32 &prevLayer, const LayerKey &key)]: "
      "Error reading input depth.");
  uiInputDepth = KUtilities::UInt64::ToEndian(buffer, true);

  return std::shared_ptr<MaxPoolLayer32>(
    new Convolutional::MaxPoolLayer32(network,
      uiPoolingWidth, uiPoolingHeight, uiPoolingStep,
      f,
      uiInputWidth, uiInputHeight, uiInputDepth,
      prevLayer, uiMinibatchSize, key));
}
```

B.18 File: kmergelayer.h

```
#ifndef K_FFANN_MERGE_LAYER_H
#define K_FFANN_MERGE_LAYER_H

#include <memory>
#include <cstring>
#include <istream>
#include <ostream>

#include "knnkeys.h"
#include "klayer.h"
#include "kfuncact.h"
#include "ktypeutils.h"


namespace NeuralNetwork
{
  class Network32;

  /// <summary>
```

```
/// Encapsulates the functionality of a merge layer.
/// A merge layer takes inputs from all the layers connected to itself
/// that need to be merged and then replicates the inputs as outputs
/// as if all merged layers were actually one.
/// </summary>
class MergeLayer32 : public TrainableLayer32
{
public:

  static const int LAYER_TYPE = 410;

  /// <summary>
  /// Creates a new merge layer.
  /// </summary>
  MergeLayer32(const Network32 &network,
    std::size_t numInputs, const std::shared_ptr<Layer32> *arrLayers,
      ↪ std::size_t numLayers,
    std::size_t uiMinibatchSize, const LayerKey &key) :
    TrainableLayer32(network, numInputs, NeuralNetwork::
      ↪ ActivationFunctions::Identity32(), uiMinibatchSize, key),
    m_bTrainable(true)
  {
    m_weights.resize(numInputs);
    m_neurons.resize(numInputs);
    for (std::size_t i = 0; i < numInputs; i++)
      m_neurons[i] = std::shared_ptr<Neuron32>(
        new Neuron32(m_activations, m_biases, m_weighted_input,
          ↪ m_deltas,
          m_weights, 1,
          i, numInputs, uiMinibatchSize, m_NeuronKey)
        );

    m_arrLayers.assign(arrLayers, arrLayers + numLayers);
  }


  int getLayerType() const override
  {
    return LAYER_TYPE;
  }

  /// <summary>
  /// Returns index of the first previous layer.
  /// </summary>
  std::size_t getPreviousLayerIndex() const override
  {
    return m_arrLayers[0]->getNetworkIndex();
  }

  /// <summary>
  /// Returns the number of layers merged into this one.
```

```
    /// </summary>
    std::size_t getPreviousLayerCount() const
    {
      return m_arrLayers.size();
    }

    const Layer32& getPreviousLayer(std::size_t index) const
    {
      return *m_arrLayers[index];
    }

    /// <summary>
    /// Returns whether this layer's weights and biases can be changed
       ↪ through training.
    /// </summary>
    bool IsTrainable() const override
    {
      return m_bTrainable;
    }

    /// <summary>
    /// Sets whether this layer's weights and biases can be changed
       ↪ through training.
    /// </summary>
    void setTrainable(bool value)
    {
      m_bTrainable = value;
    }

    void Propagate() override;

    void Backpropagate(NeuralNetwork::Layer32 &fromLayer) override;
    void PostBackpropagate(float fInvMinibatchSize, float
       ↪ fInvTotalSamples) override;


    void Save(std::ostream &fNum, const LayerKey &key) const;
    static std::shared_ptr<MergeLayer32> Load(std::istream &fNum, const
       ↪ Network32 &network,
      std::size_t numInputs, const std::shared_ptr<Layer32> *arrLayers,
         ↪ std::size_t numLayers,
      std::size_t uiMinibatchSize, const LayerKey &key);

  protected:
    void Load(std::istream &fNum, const LayerKey &key);

  private:
    bool m_bTrainable;
    std::vector<std::shared_ptr<Layer32>> m_arrLayers;
  };
```

```
  typedef MergeLayer32 MergeLayer;
}

#endif
```

B.19 File: kmergelayer.cpp

```
#include "kmergelayer.h"
#include "kinputlayer.h"
#include "ktypeutils.h"

#include "mkl.h"

void NeuralNetwork::MergeLayer32::Propagate()
{


  std::size_t uiActivationOffset = 0;
  for (std::size_t prev_layer_i = 0; prev_layer_i <
     ↪ getPreviousLayerCount(); prev_layer_i++)
  {
    const Layer32 &prevLayer = getPreviousLayer(prev_layer_i);
    #pragma omp parallel for
    for (int sample_i = 0; sample_i < (int)m_uiMinibatchSize; sample_i
       ↪ ++)
      vsMul((int)prevLayer.getNeuronCount(), m_weights.data() +
         ↪ uiActivationOffset, prevLayer.getActivations(sample_i),
        m_activations.data() + sample_i * this->getNeuronCount() +
           ↪ uiActivationOffset);

    uiActivationOffset += prevLayer.getNeuronCount();
  }

  #pragma omp parallel for
  for (int sample_i = 0; sample_i < (int)m_uiMinibatchSize; sample_i++)
  {
    cblas_saxpy((int)this->getNeuronCount(),
      1.0, m_biases.data(), 1,
      m_activations.data() + sample_i * this->getNeuronCount(), 1);
  }
}

void NeuralNetwork::MergeLayer32::Backpropagate(NeuralNetwork::Layer32 &
   ↪  fromLayer)
{
```

```
  std::size_t uiActivationOffset = 0;
  for (std::size_t prev_layer_i = 0; prev_layer_i <
     ↪ getPreviousLayerCount(); prev_layer_i++)
  {
    Layer32 &prevLayer = *m_arrLayers[prev_layer_i];
    if (prevLayer.getLayerType() != NeuralNetwork::InputLayer32::
       ↪ LAYER_TYPE)
    {
      #pragma omp parallel for
      for (int sample_i = 0; sample_i < (int)m_uiMinibatchSize; sample_i
         ↪ ++)
        vsMul((int)prevLayer.getNeuronCount(), m_weights.data() +
           ↪ uiActivationOffset, this->getDeltas(sample_i) +
           ↪ uiActivationOffset,
          prevLayer.getDeltaVector().unsafe_data() + sample_i *
             ↪ prevLayer.getNeuronCount());

    }
    uiActivationOffset += prevLayer.getNeuronCount();
  }


  if (this->IsTrainable())
  {
    for (std::size_t sample_i = 0; sample_i < m_uiMinibatchSize;
       ↪ sample_i++)
    {
      std::size_t uiActivationOffset = 0;
      for (std::size_t prev_layer_i = 0; prev_layer_i <
         ↪ getPreviousLayerCount(); prev_layer_i++)
      {
        Layer32 &prevLayer = *m_arrLayers[prev_layer_i];
        #pragma omp parallel for
        for (int neuron_from_i = 0; neuron_from_i < (int)prevLayer.
           ↪ getNeuronCount(); neuron_from_i++)
        {
          Neuron32 &neuron_to = this->getNeuron(neuron_from_i +
             ↪ uiActivationOffset);

          neuron_to.getDeltaWeights()[0] +=
            (neuron_to.getDelta(sample_i) * prevLayer.getActivations(
               ↪ sample_i)[neuron_from_i]);
          neuron_to.DeltaBias += neuron_to.getDelta(sample_i);
        }

        uiActivationOffset += prevLayer.getNeuronCount();
      }
    }
  }
}
```

```
void NeuralNetwork::MergeLayer32::PostBackpropagate(float
    ↪ fInvMinibatchSize, float fInvTotalSamples)
{
  if (this->IsTrainable())
  {
    #pragma omp parallel for
    for (int j = 0; j < (int)this->getNeuronCount(); j++)
    {
      Neuron32 &neuron = this->getNeuron(j);

      neuron.UpdateWeights(fInvMinibatchSize, fInvTotalSamples);
      neuron.UpdateBias(fInvMinibatchSize);

      neuron.ScaleDeltaWeights();
      neuron.ScaleDeltaBias();
    }
  }
}

void NeuralNetwork::MergeLayer32::Save(std::ostream & fNum, const
    ↪ LayerKey & key) const
{
  std::uint64_t u_output;

  u_output = KUtilities::UInt64::getLittleEndian((this->IsTrainable() ?
      ↪ 1 : 0));
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Save(std::
        ↪ ostream &fNum, const LayerKey &key)]: "
      "Error writing trainable flag.");


  u_output = KUtilities::UInt64::getLittleEndian(this->getWeightsCount()
      ↪ );
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Save(std::
        ↪ ostream &fNum, const LayerKey &key)]: "
      "Error writing number of weights.");

  fNum.write((char*)m_weights.data(), sizeof(float) * m_weights.size());
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Save(std::
        ↪ ostream &fNum, const LayerKey &key)]: "
      "Error writing all weights.");

  fNum.write((char*)m_biases.data(), sizeof(float) * m_biases.size());
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Save(std::
        ↪ ostream &fNum, const LayerKey &key)]: "
```

```
      "Error writing all biases.");
}

void NeuralNetwork::MergeLayer32::Load(std::istream &fNum, const
    ↪ LayerKey &key)
{

  char buffer[8];

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Load(std::
        ↪ istream &fNum, const LayerKey &key)]: "
      "Error reading trainable flag.");
  this->setTrainable(KUtilities::UInt64::ToEndian(buffer, true) != 0);

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() != sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Load(std::
        ↪ istream &fNum, const LayerKey &key)]: "
      "Error reading number of weights.");
  std::size_t uiNumWeights = KUtilities::UInt64::ToEndian(buffer, true);
  if (uiNumWeights != this->getWeightsCount())
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Load(std::
        ↪ istream &fNum, const LayerKey &key)]: "
      "Number of weights in file is different than weights in layer.");

  fNum.read((char*)m_weights.unsafe_data(), sizeof(float) * m_weights.
      ↪ size());
  if (!fNum.good() || fNum.gcount() != sizeof(float) * m_weights.size())
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Load(std::
        ↪ istream &fNum, const LayerKey &key)]: "
      "Error reading all weights.");

  fNum.read((char*)m_biases.unsafe_data(), sizeof(float) * m_biases.size
      ↪ ());
  if (!fNum.good() || fNum.gcount() != sizeof(float) * m_biases.size())
    throw std::runtime_error("[NeuralNetwork::MergeLayer32::Load(std::
        ↪ istream &fNum, const LayerKey &key)]: "
      "Error reading all biases.");
}

std::shared_ptr<NeuralNetwork::MergeLayer32> NeuralNetwork::MergeLayer32
    ↪ ::Load(std::istream &fNum,
  const Network32 &network, std::size_t numInputs,
  const std::shared_ptr<Layer32>* arrLayers, std::size_t numLayers,
  std::size_t uiMinibatchSize, const LayerKey & key)
{
  std::shared_ptr<MergeLayer32> retval = std::shared_ptr<MergeLayer32>(
    new MergeLayer32(network, numInputs, arrLayers, numLayers,
        ↪ uiMinibatchSize, key));
```

```
  retval->Load(fNum, key);

  return retval;
}
```

B.20 File: knetwork.h

```
#ifndef K_FFANN_NETWORK_H
#define K_FFANN_NETWORK_H

#include <memory>
#include <mutex>
#include <stdexcept>
#include <vector>

#include "klayer.h"
#include "kinputlayer.h"
#include "kfclayer.h"
#include "kconvlayer.h"
#include "kmaxpoollayer.h"
#include "kmergelayer.h"
#include "ktypeutils.h"


namespace NeuralNetwork
{
  class Network32
  {
  public:

    static const int VERSION = 1;

    Network32(std::size_t uiMinibatchSize = 1) :
      m_uiMinibatchSize(uiMinibatchSize),
      m_bLocked(false),
      m_iNumLocks(0)
    {
      if (uiMinibatchSize <= 0)
        throw std::out_of_range("[NeuralNetwork::Network32::Network32(
            ↪ size_t)] uiMinibatchSize must be greater than zero.");
    }

    Network32(const Network32&) = delete;
    Network32& operator=(const Network32&) = delete;


    std::size_t getLayerCount() const
```

```
    {
      return m_Layers.size();
    }

    const std::vector<std::shared_ptr<Layer32>> getLayers() const
    {
      return m_Layers;
    }

    InputLayer32& getInputLayer() const
    {
      return (InputLayer32&)*m_Layers.front();
    }

    Layer32& getOutputLayer() const
    {
      return *m_Layers.back();
    }

    std::size_t getMinibatchSize() const
    {
      return m_uiMinibatchSize;
    }

    /// <summary>
    /// Initializes a new maxpool layer using another convolutional
        ↪ layer as input layer.
    /// </summary>
    Convolutional::MaxPoolLayer32& CreateMaxPoolLayer(
      std::size_t uiPoolingWidth, std::size_t uiPoolingHeight, std::
          ↪ size_t uiPoolingStep,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f)
    {
      return CreateMaxPoolLayer(uiPoolingWidth, uiPoolingHeight,
          ↪ uiPoolingStep, f, getLayerCount() - 1);
    }

    /// <summary>
    /// Initializes a new maxpool layer using another convolutional
        ↪ layer as input layer.
    /// </summary>
    Convolutional::MaxPoolLayer32& CreateMaxPoolLayer(
      std::size_t uiPoolingWidth, std::size_t uiPoolingHeight, std::
          ↪ size_t uiPoolingStep,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f, std::size_t prevLayerIndex);

    /// <summary>
    /// Initializes a new maxpool layer.
    /// </summary>
```

```
Convolutional::MaxPoolLayer32& CreateMaxPoolLayer(
  std::size_t uiPoolingWidth, std::size_t uiPoolingHeight, std::
      ↪ size_t uiPoolingStep,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<
      ↪ float> &f,
  std::size_t uiInputWidth, std::size_t uiInputHeight, std::size_t
      ↪ uiInputDepth)
{
  return CreateMaxPoolLayer(uiPoolingWidth, uiPoolingHeight,
      ↪ uiPoolingStep,
    f,
    uiInputWidth, uiInputHeight, uiInputDepth, getLayerCount() - 1);
}

/// <summary>
/// Initializes a new maxpool layer.
/// </summary>
Convolutional::MaxPoolLayer32& CreateMaxPoolLayer(
  std::size_t uiPoolingWidth, std::size_t uiPoolingHeight, std::
      ↪ size_t uiPoolingStep,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<
      ↪ float> &f,
  std::size_t uiInputWidth, std::size_t uiInputHeight, std::size_t
      ↪ uiInputDepth, std::size_t prevLayerIndex);

/// <summary>
/// Initializes a new convolutional layer using another
    ↪ convolutional layer as input layer.
/// </summary>
Convolutional::ConvolutionalLayer32& CreateConvolutionalLayer(
  const std::shared_ptr<Convolutional::Filter32> *arrFilters, const
      ↪ std::size_t *arrFeatureSteps, std::size_t uiNumFeatures,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<
      ↪ float> &f)
{
  return CreateConvolutionalLayer(arrFilters, arrFeatureSteps,
      ↪ uiNumFeatures, f, getLayerCount() - 1);
}

/// <summary>
/// Initializes a new convolutional layer using another
    ↪ convolutional layer as input layer.
/// </summary>
Convolutional::ConvolutionalLayer32& CreateConvolutionalLayer(
  const std::shared_ptr<Convolutional::Filter32> *arrFilters, const
      ↪ std::size_t *arrFeatureSteps, std::size_t uiNumFeatures,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<
      ↪ float> &f,
  std::size_t prevLayerIndex);

/// <summary>
```

```
    /// Initializes a new convolutional layer.
    /// </summary>
    Convolutional::ConvolutionalLayer32& CreateConvolutionalLayer(
      const std::shared_ptr<Convolutional::Filter32> *arrFilters, const
          ↪ std::size_t *arrFeatureSteps, std::size_t uiNumFeatures,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f,
      std::size_t uiInputWidth, std::size_t uiInputHeight, std::size_t
          ↪ uiInputDepth)
    {
      return CreateConvolutionalLayer(arrFilters, arrFeatureSteps,
          ↪ uiNumFeatures,
        f,
        uiInputWidth, uiInputHeight, uiInputDepth,
        getLayerCount() - 1);
    }

    /// <summary>
    /// Initializes a new convolutional layer.
    /// </summary>
    Convolutional::ConvolutionalLayer32& CreateConvolutionalLayer(
      const std::shared_ptr<Convolutional::Filter32> *arrFilters, const
          ↪ std::size_t *arrFeatureSteps, std::size_t uiNumFeatures,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f,
      std::size_t uiInputWidth, std::size_t uiInputHeight, std::size_t
          ↪ uiInputDepth,
      std::size_t prevLayerIndex);

    FullyConnectedLayer32& CreateFullyConnectedLayer(std::size_t
        ↪ numNeurons,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f)
    {
      return CreateFullyConnectedLayer(numNeurons, f, getLayerCount() -
          ↪ 1);
    }

    FullyConnectedLayer32& CreateFullyConnectedLayer(std::size_t
        ↪ numNeurons,
      const NeuralNetwork::ActivationFunctions::IActivationFunction<
          ↪ float> &f,
      std::size_t prevLayerIndex);

    InputLayer32& CreateInputLayer(std::size_t numInputs);

    /// <summary>
    /// Creates a new MergeLayer32 and adds it to this network.
    /// </summary>
    MergeLayer32& CreateMergeLayer(std::size_t *arrPrevLayerIndices, std
        ↪ ::size_t size);
```

```
    /// <summary>
    /// Creates a new MergeLayer32 and adds it to this network.
    /// </summary>
    MergeLayer32& CreateMergeLayer(std::vector<std::size_t>
        ↪ arrPrevLayerIndices)
    {
      return CreateMergeLayer(arrPrevLayerIndices.data(),
          ↪ arrPrevLayerIndices.size());
    }


    /// <summary>
    /// Performs a per-layer forward propagation through the network.
    /// </summary>
    void PropagatePerLayer()
    {
      for (int i = 0; i < (int)m_Layers.size(); i++)
        m_Layers[i]->Propagate();
    }

    void Save(std::ostream &fNum, const NetworkKey &key) const;
    void Load(std::istream &fNum, const NetworkKey &key);

  protected:
    /// <summary>
    /// Adds a new layer to this network.
    /// </summary>
    virtual void addLayer(std::shared_ptr<Layer32> layer)
    {
      if (!layer)
        throw std::invalid_argument("layer cannot be null");

      layer->setNetworkIndex(m_Layers.size(), m_LayerKey);
      m_Layers.push_back(layer);
    }

    LayerKey m_LayerKey;

  private:
    void saveLayer(std::ostream &fNum, const InputLayer32 &layer) const;
    void saveLayer(std::ostream &fNum, const FullyConnectedLayer32 &
        ↪ layer) const;
    void saveLayer(std::ostream &fNum, const MergeLayer32 &layer) const;
    void saveConvolutionalLayer(std::ostream &fNum, const NeuralNetwork
        ↪ ::Convolutional::ConvolutionalLayer32 &layer) const;
    void saveMaxpoolLayer(std::ostream &fNum, const NeuralNetwork::
        ↪ Convolutional::MaxPoolLayer32 &layer) const;
    std::shared_ptr<NeuralNetwork::ActivationFunctions::
        ↪ IActivationFunction<float>> loadActivationFunction(std::
        ↪ istream & fNum);
    void loadInputLayer(std::istream & fNum);
```

```
    void loadFullyConnectedLayer(std::istream & fNum);
    void loadConvolutionalLayer(std::istream & fNum);
    void loadMaxpoolLayer(std::istream & fNum);
    void loadMergeLayer(std::istream & fNum);

    std::vector<std::shared_ptr<Layer32>> m_Layers;

    std::size_t m_uiMinibatchSize;

    int m_iNumLocks;
    bool m_bLocked;
  };

  typedef Network32 Network;
}

#endif
```

B.21 File: knetwork.cpp

```
#include <cstdint>

#include "knetwork.h"

NeuralNetwork::Convolutional::MaxPoolLayer32 & NeuralNetwork::Network32
    ↪ ::CreateMaxPoolLayer(
  std::size_t uiPoolingWidth, std::size_t uiPoolingHeight, std::size_t
      ↪ uiPoolingStep,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<float> &
      ↪ f, std::size_t prevLayerIndex)
{
  std::shared_ptr<Layer32> retval;

  if (m_Layers.empty())
    throw std::runtime_error("Only Input layer allowed as first layer in
        ↪ the network.");
  if (prevLayerIndex >= getLayerCount())
    throw std::out_of_range("prevLayerIndex outside of range of layers."
        ↪ );

  Convolutional::ConvolutionalLayer32 *prevLayer = nullptr;
  try
  {
    prevLayer = dynamic_cast<Convolutional::ConvolutionalLayer32*>(
        ↪ m_Layers[prevLayerIndex].get());
  }
  catch (...)
  {
```

```
  }
  if (!prevLayer)
    throw std::runtime_error("Previous layer is not a convolutional
        ↪ layer.");

  retval = std::shared_ptr<Layer32>(
    new Convolutional::MaxPoolLayer32(*this, uiPoolingWidth,
        ↪ uiPoolingHeight, uiPoolingStep, f,
      *prevLayer, m_uiMinibatchSize, m_LayerKey)
    );

  addLayer(retval);

  return (Convolutional::MaxPoolLayer32&)*retval;
}

NeuralNetwork::Convolutional::MaxPoolLayer32 & NeuralNetwork::Network32
    ↪ ::CreateMaxPoolLayer(std::size_t uiPoolingWidth, std::size_t
    ↪ uiPoolingHeight, std::size_t uiPoolingStep, const NeuralNetwork::
    ↪ ActivationFunctions::IActivationFunction<float> & f, std::size_t
    ↪ uiInputWidth, std::size_t uiInputHeight, std::size_t uiInputDepth,
    ↪  std::size_t prevLayerIndex)
{
  std::shared_ptr<Layer32> retval;

  if (m_Layers.empty())
    throw std::runtime_error("Only Input layer allowed as first layer in
        ↪ the network.");
  if (prevLayerIndex >= getLayerCount())
    throw std::out_of_range("prevLayerIndex outside of range of layers."
        ↪ );

  retval = std::shared_ptr<Layer32>(
    new Convolutional::MaxPoolLayer32(*this, uiPoolingWidth,
        ↪ uiPoolingHeight, uiPoolingStep, f,
      uiInputWidth, uiInputHeight, uiInputDepth,
      *m_Layers[prevLayerIndex], m_uiMinibatchSize, m_LayerKey)
    );

  addLayer(retval);

  return (Convolutional::MaxPoolLayer32&)*retval;
}

NeuralNetwork::Convolutional::ConvolutionalLayer32 & NeuralNetwork::
    ↪ Network32::CreateConvolutionalLayer(
  const std::shared_ptr<Convolutional::Filter32>* arrFilters,
  const std::size_t *arrFeatureSteps, std::size_t uiNumFeatures,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<float> &
      ↪ f,
  std::size_t prevLayerIndex)
```

```
{
  std::shared_ptr<Layer32> retval;

  if (m_Layers.empty())
    throw std::runtime_error("Only Input layer allowed as first layer in
        ↪ the network.");
  if (prevLayerIndex >= getLayerCount())
    throw std::out_of_range("prevLayerIndex outside of range of layers."
        ↪ );

  Convolutional::ConvolutionalLayer32 *prevLayer = nullptr;
  try
  {
    prevLayer = dynamic_cast<Convolutional::ConvolutionalLayer32*>(
        ↪ m_Layers[prevLayerIndex].get());
  }
  catch (...)
  {
  }
  if (!prevLayer)
    throw std::runtime_error("Previous layer is not a convolutional
        ↪ layer.");

  retval = std::shared_ptr<Layer32>(
    new Convolutional::ConvolutionalLayer32(*this, arrFilters,
        ↪ arrFeatureSteps, uiNumFeatures, f,
      *prevLayer, m_uiMinibatchSize, m_LayerKey)
    );

  addLayer(retval);

  return (Convolutional::ConvolutionalLayer32&)*retval;
}

NeuralNetwork::Convolutional::ConvolutionalLayer32 & NeuralNetwork::
    ↪ Network32::CreateConvolutionalLayer(
  const std::shared_ptr<Convolutional::Filter32>* arrFilters,
  const std::size_t *arrFeatureSteps, std::size_t uiNumFeatures,
  const NeuralNetwork::ActivationFunctions::IActivationFunction<float> &
      ↪ f,
  std::size_t uiInputWidth, std::size_t uiInputHeight, std::size_t
      ↪ uiInputDepth,
  std::size_t prevLayerIndex)
{
  std::shared_ptr<Layer32> retval;

  if (m_Layers.empty())
    throw std::runtime_error("Only Input layer allowed as first layer in
        ↪ the network.");
  if (prevLayerIndex >= getLayerCount())
```

```
    throw std::out_of_range("prevLayerIndex outside of range of layers.");

  retval = std::shared_ptr<Layer32>(
    new Convolutional::ConvolutionalLayer32(*this, arrFilters,
      arrFeatureSteps, uiNumFeatures, f,
      uiInputWidth, uiInputHeight, uiInputDepth,
      *m_Layers[prevLayerIndex], m_uiMinibatchSize, m_LayerKey)
    );

  addLayer(retval);

  return (Convolutional::ConvolutionalLayer32&)*retval;
}

NeuralNetwork::FullyConnectedLayer32 & NeuralNetwork::Network32::
    CreateFullyConnectedLayer(std::size_t numNeurons, const
    NeuralNetwork::ActivationFunctions::IActivationFunction<float> & f
    , std::size_t prevLayerIndex)
{
  std::shared_ptr<Layer32> retval;

  if (numNeurons <= 0)
    throw std::invalid_argument("Number of neurons must be positive.");
  if (m_Layers.empty())
    throw std::runtime_error("Only Input layer allowed as first layer in
      the network.");
  if (prevLayerIndex >= getLayerCount())
    throw std::out_of_range("prevLayerIndex outside of range of layers.");

  retval = std::shared_ptr<Layer32>(
    new FullyConnectedLayer32(*this, numNeurons, f,
      *m_Layers[prevLayerIndex], m_uiMinibatchSize, m_LayerKey)
    );

  addLayer(retval);

  return (FullyConnectedLayer32&)*retval;
}

NeuralNetwork::InputLayer32 & NeuralNetwork::Network32::CreateInputLayer
    (std::size_t numInputs)
{
  std::shared_ptr<Layer32> retval;

  if (numInputs <= 0)
    throw std::invalid_argument("Number of inputs must be positive.");
  if (!m_Layers.empty())
    throw std::runtime_error("Input layer only allowed as first layer in
      the network.");
```

```
  retval = std::shared_ptr<Layer32>(
    new InputLayer32(*this, numInputs, m_uiMinibatchSize, m_LayerKey)
    );

  addLayer(retval);

  return (InputLayer32&)*retval;
}

NeuralNetwork::MergeLayer32 & NeuralNetwork::Network32::CreateMergeLayer
    ↪ (std::size_t *arrPrevLayerIndices, std::size_t size)
{
  std::shared_ptr<Layer32> retval;

  if (size <= 0)
    throw std::invalid_argument("Number of previous layers must be
        ↪ positive.");
  if (m_Layers.empty())
    throw std::runtime_error("Only Input layer allowed as first layer in
        ↪ the network.");

  std::size_t uiNumInputs = 0;
  std::vector<std::shared_ptr<Layer32>> arrLayers(size);
  for (std::size_t i = 0; i < size; i++)
  {
    if (arrPrevLayerIndices[i] >= getLayerCount())
      throw std::out_of_range("prevLayerIndex outside of range of layers
          ↪ .");
    arrLayers[i] = m_Layers[arrPrevLayerIndices[i]];
    uiNumInputs += arrLayers[i]->getNeuronCount();
  }

  retval = std::shared_ptr<Layer32>(
    new MergeLayer32(*this, uiNumInputs, arrLayers.data(), arrLayers.
        ↪ size(), m_uiMinibatchSize, m_LayerKey)
    );

  addLayer(retval);

  return (MergeLayer32&)*retval;
}

void NeuralNetwork::Network32::saveLayer(std::ostream & fNum, const
    ↪ InputLayer32 & layer) const
{
  std::int32_t s_output;
  std::uint64_t u_output;

  s_output = KUtilities::Int32(layer.getLayerType());
  fNum.write((char*)&s_output, sizeof(s_output));
```

```
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const InputLayer32 &layer)]: "
                "Error writing layer type.");

  u_output = KUtilities::UInt64::getLittleEndian(layer.getNeuronCount())
      ↪ ;
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const InputLayer32 &layer)]: "
                "Error writing input size.");

}

void NeuralNetwork::Network32::saveLayer(std::ostream &fNum, const
    ↪ FullyConnectedLayer32 &layer) const
{
  std::int32_t s_output;
  std::uint64_t u_output;

  s_output = KUtilities::Int32::getLittleEndian(layer.getLayerType());
  fNum.write((char*)&s_output, sizeof(s_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const FullyConnectedLayer32 &layer)]: "
                "Error writing layer type.");

  u_output = KUtilities::UInt64::getLittleEndian(layer.getNeuronCount())
      ↪ ;
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const FullyConnectedLayer32 &layer)]: "
                "Error writing neuron count.");

  u_output = KUtilities::UInt64::getLittleEndian(layer.
      ↪ getPreviousLayerIndex());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const FullyConnectedLayer32 &layer)]: "
                "Error writing previous layer index.");


  s_output = KUtilities::Int32::getLittleEndian(layer.
      ↪ getActivationFunction().getID());
  fNum.write((char*)&s_output, sizeof(s_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const FullyConnectedLayer32 &layer)]: "
```

```
                "Error writing activation function ID.");

  layer.getActivationFunction().Save(fNum, { });

  layer.Save(fNum, { });
}

void NeuralNetwork::Network32::saveLayer(std::ostream & fNum, const
    ↪ MergeLayer32 & layer) const
{
  std::int32_t s_output;
  std::uint64_t u_output;

  s_output = KUtilities::Int32::getLittleEndian(layer.getLayerType());
  fNum.write((char*)&s_output, sizeof(s_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const MergeLayer32 &layer)]: "
      "Error writing layer type.");

  u_output = KUtilities::UInt64::getLittleEndian(layer.getNeuronCount())
      ↪ ;
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const MergeLayer32 &layer)]: "
      "Error writing neuron count.");


  u_output = KUtilities::UInt64::getLittleEndian(layer.
      ↪ getPreviousLayerCount());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
        ↪ ostream &fNum, const MergeLayer32 &layer)]: "
      "Error writing previous layer count.");

  for (std::size_t prevLayer_i = 0; prevLayer_i < layer.
      ↪ getPreviousLayerCount(); prevLayer_i++)
  {
    u_output = KUtilities::UInt64::getLittleEndian(layer.
        ↪ getPreviousLayer(prevLayer_i).getNetworkIndex());
    fNum.write((char*)&u_output, sizeof(u_output));
    if (!fNum.good())
      throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
          ↪ ostream &fNum, const MergeLayer32 &layer)]: "
        "Error writing previous layer index.");
  }
```

```
  s_output = KUtilities::Int32::getLittleEndian(layer.
    ↪ getActivationFunction().getID());
  fNum.write((char*)&s_output, sizeof(s_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::saveLayer(
      ↪ ostream &fNum, const MergeLayer32 &layer)]: "
      "Error writing activation function ID.");

  layer.getActivationFunction().Save(fNum, {});

  layer.Save(fNum, {});
}

void NeuralNetwork::Network32::saveConvolutionalLayer(std::ostream &
  ↪ fNum, const NeuralNetwork::Convolutional::ConvolutionalLayer32 &
  ↪ layer) const
{
  std::int32_t s_output;
  std::uint64_t u_output;

  s_output = KUtilities::Int32::getLittleEndian(layer.getLayerType());
  fNum.write((char*)&s_output, sizeof(s_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::
      ↪ saveConvolutionalLayer(ostream &fNum, const Convolutional::
      ↪ ConvolutionalLayer32 &layer)]: "
      "Error writing layer type.");

  u_output = KUtilities::UInt64::getLittleEndian(layer.
    ↪ getPreviousLayerIndex());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::
      ↪ saveConvolutionalLayer(ostream &fNum, const Convolutional::
      ↪ ConvolutionalLayer32 &layer)]: "
      "Error writing previous layer index.");

  s_output = KUtilities::Int32::getLittleEndian(layer.
    ↪ getActivationFunction().getID());
  fNum.write((char*)&s_output, sizeof(s_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::
      ↪ saveConvolutionalLayer(ostream &fNum, const Convolutional::
      ↪ ConvolutionalLayer32 &layer)]: "
      "Error writing activation function ID.");

  layer.getActivationFunction().Save(fNum, {});

  layer.Save(fNum, {});
}
```

```
void NeuralNetwork::Network32::saveMaxpoolLayer(std::ostream & fNum,
    ↪ const NeuralNetwork::Convolutional::MaxPoolLayer32 & layer) const
{
  std::int32_t s_output;
  std::uint64_t u_output;

  s_output = KUtilities::Int32::getLittleEndian(layer.getLayerType());
  fNum.write((char*)&s_output, sizeof(s_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::
        ↪ saveMaxpoolLayer(ostream &fNum, const Convolutional::
        ↪ MaxPoolLayer32 &layer)]: "
      "Error writing layer type.");

  u_output = KUtilities::UInt64::getLittleEndian(layer.
      ↪ getPreviousLayerIndex());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::
        ↪ saveMaxpoolLayer(ostream &fNum, const Convolutional::
        ↪ MaxPoolLayer32 &layer)]: "
      "Error writing previous layer index.");

  s_output = KUtilities::Int32::getLittleEndian(layer.
      ↪ getActivationFunction().getID());
  fNum.write((char*)&s_output, sizeof(s_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::
        ↪ saveMaxpoolLayer(ostream &fNum, const Convolutional::
        ↪ MaxPoolLayer32 &layer)]: "
      "Error writing activation function ID.");

  layer.getActivationFunction().Save(fNum, {});

  layer.Save(fNum, {});
}

void NeuralNetwork::Network32::Save(std::ostream &fNum, const NetworkKey
    ↪ &key) const
{
  std::uint64_t u_output;

  u_output = KUtilities::UInt64::getLittleEndian(this->VERSION);
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::Save(ostream &
        ↪ fNum, const NetworkKey &key)]: "
      "Error writing version.");

  u_output = KUtilities::UInt64::getLittleEndian(sizeof(float));
  fNum.write((char*)&u_output, sizeof(u_output));
```

```
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::Save(ostream &
        ↪ fNum, const NetworkKey &key)]: "
      "Error writing size of float numbers.");

  u_output = KUtilities::UInt64::getLittleEndian(getLayerCount());
  fNum.write((char*)&u_output, sizeof(u_output));
  if (!fNum.good())
    throw std::runtime_error("[NeuralNetwork::Network32::Save(ostream &
        ↪ fNum, const NetworkKey &key)]: "
                "Error writing layer count.");

  for (std::size_t layer_i = 0; layer_i < getLayerCount(); layer_i++)
  {
    switch (m_Layers[layer_i]->getLayerType())
    {
    case InputLayer32::LAYER_TYPE:
      saveLayer(fNum, (const InputLayer32&)*m_Layers[layer_i]);
      break;

    case FullyConnectedLayer32::LAYER_TYPE:
      saveLayer(fNum, (const FullyConnectedLayer32&)*m_Layers[layer_i]);
      break;

    case Convolutional::ConvolutionalLayer32::LAYER_TYPE:
      saveConvolutionalLayer(fNum, (const NeuralNetwork::Convolutional::
          ↪ ConvolutionalLayer32&)*m_Layers[layer_i]);
      break;

    case Convolutional::MaxPoolLayer32::LAYER_TYPE:
      saveMaxpoolLayer(fNum, (const NeuralNetwork::Convolutional::
          ↪ MaxPoolLayer32&)*m_Layers[layer_i]);
      break;

    case MergeLayer32::LAYER_TYPE:
      saveLayer(fNum, (const NeuralNetwork::MergeLayer32&)*m_Layers[
          ↪ layer_i]);
      break;

    default:
      throw std::runtime_error("[NeuralNetwork::Network32::Save(ostream
          ↪ &fNum, const NetworkKey &key)]: "
                "Invalid layer type detected.");
      break;
    }
  }
}

std::shared_ptr<NeuralNetwork::ActivationFunctions::IActivationFunction<
    ↪ float>>
  NeuralNetwork::Network32::loadActivationFunction(std::istream & fNum)
```

```
{
  std::shared_ptr<NeuralNetwork::ActivationFunctions::
    ↪ IActivationFunction<float>> retval;

  char buffer[8];

  fNum.read(buffer, sizeof(std::int32_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::int32_t))
    throw std::runtime_error("[NeuralNetwork::Network32::
      ↪ loadActivationFunction(istream &fNum)]: "
                "Error reading activation function ID.");
  int iID = KUtilities::Int32::ToEndian(buffer, true);

  switch (iID)
  {
  case NeuralNetwork::ActivationFunctions::ActivationFunctionID::
    ↪ IDENTITY_ID:
    retval = std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>>(
      NeuralNetwork::ActivationFunctions::Identity32::Load(fNum, { }));
    break;

  case NeuralNetwork::ActivationFunctions::ActivationFunctionID::STEP_ID
    ↪ :
    retval = std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>>(
      NeuralNetwork::ActivationFunctions::Step32::Load(fNum, { }));
    break;

  case NeuralNetwork::ActivationFunctions::ActivationFunctionID::
    ↪ INVERTED_STEP_ID:
    retval = std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>>(
      NeuralNetwork::ActivationFunctions::InvertedStep32::Load(fNum, {
        ↪ }));
    break;

  case NeuralNetwork::ActivationFunctions::ActivationFunctionID::
    ↪ HYPERBOLIC_TANGENT_ID:
    retval = std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>>(
      NeuralNetwork::ActivationFunctions::HyperbolicTangent32::Load(fNum
        ↪ , { }));
    break;

  case NeuralNetwork::ActivationFunctions::ActivationFunctionID::
    ↪ SIGMOID_ID:
    retval = std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>>(
      NeuralNetwork::ActivationFunctions::Sigmoid32::Load(fNum, { }));
    break;
```

```
  case NeuralNetwork::ActivationFunctions::ActivationFunctionID::RELU_ID
    ↪ :
    retval = std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>>(
      NeuralNetwork::ActivationFunctions::ReLU32::Load(fNum, { }));
    break;

  case NeuralNetwork::ActivationFunctions::ActivationFunctionID::
    ↪ SOFTMAX_ID:
    retval = std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>>(
      NeuralNetwork::ActivationFunctions::Softmax32::Load(fNum,
        ↪ getMinibatchSize(), { }));
    break;

  default:
    throw std::runtime_error("[NeuralNetwork::Network32::
      ↪ loadActivationFunction(istream &fNum)]: "
                "Invalid activation function ID detected.");
    break;
  }

  return retval;
}

void NeuralNetwork::Network32::loadInputLayer(std::istream & fNum)
{
  char buffer[8];

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Network32::loadInputLayer(
      ↪ istream &fNum)]: "
                "Error reading input size.");

  std::size_t uiInputSize = KUtilities::UInt64::ToEndian(buffer, true);

  this->CreateInputLayer(uiInputSize);
}

void NeuralNetwork::Network32::loadFullyConnectedLayer(std::istream &
  ↪ fNum)
{
  std::shared_ptr<NeuralNetwork::ActivationFunctions::
    ↪ IActivationFunction<float>> a_f;
  char buffer[8];

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
```

```
    throw std::runtime_error("[NeuralNetwork::Network32::
        ↪ loadFullyConnectedLayer(std::istream &fNum)]: "
                "Error reading number of neurons.");
  std::size_t uiNumNeurons = KUtilities::UInt64::ToEndian(buffer, true);

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Network32::
        ↪ loadFullyConnectedLayer(std::istream &fNum)]: "
                "Error reading previous layer index.");
  std::size_t uiPrevLayerIndex = KUtilities::UInt64::ToEndian(buffer,
      ↪ true);

  a_f = loadActivationFunction(fNum);

  std::shared_ptr<Layer32> pLayer = NeuralNetwork::FullyConnectedLayer32
      ↪ ::Load(fNum,
    *this, uiNumNeurons, *a_f, *m_Layers[uiPrevLayerIndex],
        ↪ m_uiMinibatchSize, {});
  addLayer(pLayer);
}

void NeuralNetwork::Network32::loadConvolutionalLayer(std::istream &
    ↪ fNum)
{
  std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>> a_f;
  char buffer[8];

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Network32::
        ↪ loadFullyConnectedLayer(std::istream &fNum)]: "
      "Error reading previous layer index.");
  std::size_t uiPrevLayerIndex = KUtilities::UInt64::ToEndian(buffer,
      ↪ true);

  a_f = loadActivationFunction(fNum);

  std::shared_ptr<Layer32> pLayer = NeuralNetwork::Convolutional::
      ↪ ConvolutionalLayer32::Load(fNum,
    *this,
    *a_f, *m_Layers[uiPrevLayerIndex],
    m_uiMinibatchSize, {});
  addLayer(pLayer);
}

void NeuralNetwork::Network32::loadMaxpoolLayer(std::istream & fNum)
{
  std::shared_ptr<NeuralNetwork::ActivationFunctions::
      ↪ IActivationFunction<float>> a_f;
```

```
  char buffer[8];

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Network32::
      ↪ loadFullyConnectedLayer(std::istream &fNum)]: "
      "Error reading previous layer index.");
  std::size_t uiPrevLayerIndex = KUtilities::UInt64::ToEndian(buffer,
    ↪ true);

  a_f = loadActivationFunction(fNum);

  std::shared_ptr<Layer32> pLayer = NeuralNetwork::Convolutional::
    ↪ MaxPoolLayer32::Load(fNum,
    *this, m_uiMinibatchSize,
    *a_f, *m_Layers[uiPrevLayerIndex], {});
  addLayer(pLayer);
}

void NeuralNetwork::Network32::loadMergeLayer(std::istream &fNum)
{
  std::shared_ptr<NeuralNetwork::ActivationFunctions::
    ↪ IActivationFunction<float>> a_f;
  char buffer[8];

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Network32::loadMergeLayer(
      ↪ std::istream &fNum)]: "
      "Error reading number of neurons.");
  std::size_t uiNumInputs = KUtilities::UInt64::ToEndian(buffer, true);
  if (uiNumInputs <= 0)
    throw std::runtime_error("[NeuralNetwork::Network32::loadMergeLayer(
      ↪ std::istream &fNum)]: "
      "Invalid number of neurons read.");


  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Network32::loadMergeLayer(
      ↪ std::istream &fNum)]: "
      "Error reading previous layer count.");
  std::size_t uiPrevLayerCount = KUtilities::UInt64::ToEndian(buffer,
    ↪ true);
  if (uiPrevLayerCount <= 0)
    throw std::runtime_error("[NeuralNetwork::Network32::loadMergeLayer(
      ↪ std::istream &fNum)]: "
      "Invalid previous layer count read.");

  std::size_t uiActualNumInputs = 0;
  std::vector<std::shared_ptr<Layer32>> arrLayers(uiPrevLayerCount);
```

```
  for (std::size_t i = 0; i < uiPrevLayerCount; i++)
  {
    fNum.read(buffer, sizeof(std::uint64_t));
    if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
      throw std::runtime_error("[NeuralNetwork::Network32::
        ↪ loadMergeLayer(std::istream &fNum)]: "
        "Error reading previous layer index.");
    std::size_t uiPrevLayerIndex = KUtilities::UInt64::ToEndian(buffer,
      ↪ true);

    if (uiPrevLayerIndex >= getLayerCount())
      throw std::out_of_range("[NeuralNetwork::Network32::loadMergeLayer
        ↪ (std::istream &fNum)]: "
        "Previous layer index outside of range of layers.");

    arrLayers[i] = m_Layers[uiPrevLayerIndex];
    uiActualNumInputs += arrLayers[i]->getNeuronCount();
  }

  if (uiNumInputs != uiActualNumInputs)
    throw std::out_of_range("[NeuralNetwork::Network32::loadMergeLayer(
      ↪ std::istream &fNum)]: "
      "Actual number of inputs and number of neurons read do not match."
        ↪ );


  a_f = loadActivationFunction(fNum);

  std::shared_ptr<Layer32> pLayer = NeuralNetwork::MergeLayer32::Load(
    ↪ fNum,
    *this, uiNumInputs, arrLayers.data(), arrLayers.size(),
      ↪ m_uiMinibatchSize, {});
  addLayer(pLayer);
}

void NeuralNetwork::Network32::Load(std::istream & fNum, const
  ↪ NetworkKey & key)
{
  try
  {
    char buffer[8];

    m_Layers.clear();

    fNum.read(buffer, sizeof(std::uint64_t));
    if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
      throw std::runtime_error("[NeuralNetwork::Network32::Load(istream
        ↪ &fNum, const NetworkKey &key)]: "
        "Error reading layer count.");
    if (KUtilities::UInt64::ToEndian(buffer, true) != this->VERSION)
```

```
    throw std::runtime_error("[NeuralNetwork::Network32::Load(istream
        ↪ &fNum, const NetworkKey &key)]: "
      "Invalid data version.");

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Network32::Load(istream
        ↪ &fNum, const NetworkKey &key)]: "
      "Error reading layer count.");
  if (KUtilities::UInt64::ToEndian(buffer, true) != sizeof(float))
    throw std::runtime_error("[NeuralNetwork::Network32::Load(istream
        ↪ &fNum, const NetworkKey &key)]: "
      "Invalid floating point data size.");

  fNum.read(buffer, sizeof(std::uint64_t));
  if (!fNum.good() || fNum.gcount() < sizeof(std::uint64_t))
    throw std::runtime_error("[NeuralNetwork::Network32::Load(istream
        ↪ &fNum, const NetworkKey &key)]: "
      "Error reading layer count.");
  std::size_t uiNumLayers = KUtilities::UInt64::ToEndian(buffer, true)
      ↪ ;

  for (std::size_t layer_i = 0; layer_i < uiNumLayers; layer_i++)
  {
    fNum.read(buffer, sizeof(std::int32_t));
    if (!fNum.good() || fNum.gcount() < sizeof(std::int32_t))
      throw std::runtime_error("[NeuralNetwork::Network32::Load(
          ↪ istream &fNum, const NetworkKey &key)]: "
        "Error reading next layer type.");
    int iLayerType = KUtilities::Int32::ToEndian(buffer, true);

    switch (iLayerType)
    {
    case InputLayer32::LAYER_TYPE:
      loadInputLayer(fNum);
      break;

    case FullyConnectedLayer32::LAYER_TYPE:
      loadFullyConnectedLayer(fNum);
      break;

    case Convolutional::ConvolutionalLayer32::LAYER_TYPE:
      loadConvolutionalLayer(fNum);
      break;

    case Convolutional::MaxPoolLayer32::LAYER_TYPE:
      loadMaxpoolLayer(fNum);
      break;

    case MergeLayer32::LAYER_TYPE:
      loadMergeLayer(fNum);
```

```
          break;

        default:
          throw std::runtime_error("[NeuralNetwork::Network32::Save(
              ↪ ostream &fNum, const NetworkKey &key)]: "
            "Invalid layer type detected.");
          break;
        }
      }
    }
    catch (...)
    {
      m_Layers.clear();

      throw;
    }
}
```

B.22 File: kneuron.h

```
#ifndef K_FFANN_NEURON_CONTAINER_H
#define K_FFANN_NEURON_CONTAINER_H

#include <vector>
#include <stdexcept>
#include "mkl.h"

#include "ksafevector.h"
#include "knnkeys.h"
#include "ktypeutils.h"


namespace NeuralNetwork
{
  class Neuron32
  {
  public:

    static const int VERSION = 1;

    Neuron32(std::vector<float> &activation, KUtilities::atomic_vector<
        ↪ float> &biases, std::vector<float> &weighted_inputs,
        KUtilities::atomic_vector<float> &deltas, KUtilities::
            ↪ atomic_vector<float> &weights, std::size_t uiNumWeights,
        std::size_t uiLayerIndex, std::size_t uiLayerNeuronCount, std::
            ↪ size_t uiMinibatchSize, const NeuronKey &key) :
      Neuron32(activation, biases, uiLayerIndex, weighted_inputs, deltas
          ↪ ,
```

```
        weights, uiLayerIndex * uiNumWeights, uiNumWeights,
        uiLayerIndex, uiLayerNeuronCount, uiMinibatchSize, key)
    { }

    Neuron32(std::vector<float> &activation, KUtilities::atomic_vector<
        ↪ float> &biases, std::size_t uiBiasIndex, std::vector<float> &
        ↪ weighted_inputs,
          KUtilities::atomic_vector<float> &deltas,
          KUtilities::atomic_vector<float> &weights, std::size_t
              ↪ uiStartWeight, std::size_t uiNumWeights,
          std::size_t uiLayerIndex, std::size_t uiLayerNeuronCount, std::
              ↪ size_t uiMinibatchSize, const NeuronKey &key) :
      m_activation(activation),
      m_biases(biases),
      m_uiBiasIndex(uiBiasIndex),
      m_weighted_inputs(weighted_inputs),
      m_deltas(deltas),
      m_deltaWeights(uiNumWeights),
      m_weights(weights),
      m_uiStartWeight(uiStartWeight),
      m_uiNumWeights(uiNumWeights),
      m_uiLayerIndex(uiLayerIndex),
      m_uiLayerNeuronCount(uiLayerNeuronCount),
      m_uiMinibatchSize(uiMinibatchSize),
      DeltaBias(0.0f),
      LearningRate(1.0f),
      Regularization(0.0f),
      FinalMomentum(0.0f),
      Momentum(0.0f),
      MomentumThreshold(0.0f)
    {
      if (uiStartWeight >= m_weights.size())
        throw std::out_of_range("Start weight index is out of bounds of
            ↪ array of weights.");
      if (uiStartWeight + uiNumWeights > m_weights.size())
        throw std::out_of_range("Start weight index plus number of
            ↪ weights falls out of bounds of array of weights.");
    }

    Neuron32(const Neuron32&) = delete;
    Neuron32& operator=(const Neuron32&) = delete;

    virtual ~Neuron32()
    { }


    /// <summary>
    /// Returns the last activation for this neuron.
    /// </summary>
    float getActivation(std::size_t sample = 0) const
    {
```

```
    return m_activation[sample * m_uiLayerNeuronCount + m_uiLayerIndex
        ↪ ];
}

/// <summary>
/// Sets a value for the activation for this neuron.
/// </summary>
void setActivation(float value, std::size_t sample = 0)
{
    m_activation[sample * m_uiLayerNeuronCount + m_uiLayerIndex] =
        ↪ value;
}


/// <summary>
/// Returns the value of the bias for this neuron.
/// </summary>
float getBias() const
{
    return m_biases[m_uiBiasIndex];
}

/// <summary>
/// Sets a value for the bias for this neuron.
/// </summary>
void setBias(float value)
{
    m_biases.store(m_uiBiasIndex, value);
}

/// <summary>
/// Updates the bias by adding the product of the
/// negative learning rate and the value of the delta bias.
/// </summary>
void UpdateBias(float fInvMinibatchSize = 1.0)
{
    m_biases.accumulate(m_uiBiasIndex, -LearningRate * DeltaBias *
        ↪ fInvMinibatchSize);
}


/// <summary>
/// Scales the delta bias by the value of the momentum.
/// </summary>
void ScaleDeltaBias()
{
    ScaleDeltaBias(Momentum);
}

/// <summary>
/// Scales the delta bias by the specified value.
```

```
    /// </summary>
    void ScaleDeltaBias(float scale)
    {
      DeltaBias *= scale;
    }


    /// <summary>
    /// Returns the last weighted input value for this neuron.
    /// </summary>
    float getWeightedInput(std::size_t sample = 0) const
    {
      return m_weighted_inputs[sample * m_uiLayerNeuronCount +
          ↪ m_uiLayerIndex];
    }

    /// <summary>
    /// Sets the weighted input value for this neuron.
    /// </summary>
    void setWeightedInput(float value, std::size_t sample = 0)
    {
      m_weighted_inputs[sample * m_uiLayerNeuronCount + m_uiLayerIndex]
          ↪ = value;
    }


    /// <summary>
    /// Returns the value of delta for this neuron.
    /// </summary>
    float getDelta(std::size_t sample = 0) const
    {
      return m_deltas[sample * m_uiLayerNeuronCount + m_uiLayerIndex];
    }

    /// <summary>
    /// Sets the value of delta for this neuron.
    /// </summary>
    void setDelta(float value, std::size_t sample = 0)
    {
      m_deltas.store(sample * m_uiLayerNeuronCount + m_uiLayerIndex,
          ↪ value);
    }


    /// <summary>
    /// Returns the index of this neuron in its container layer.
    /// </summary>
    std::size_t getLayerIndex() const
    {
      return m_uiLayerIndex;
    }
```

```
/// <summary>
/// Returns the number of connections incoming into this neuron.
/// </summary>
std::size_t getConnectionsInCount() const
{
  return getWeightCount();
}

/// <summary>
/// Returns the number of weights for this neuron's incoming
    ↪ connections.
/// </summary>
std::size_t getWeightCount() const
{
  return m_uiNumWeights;
}


/// <summary>
/// Returns the array of delta weights for this neuron.
/// </summary>
const float *getDeltaWeights() const
{
  return (m_deltaWeights.size() > 0 ?
    m_deltaWeights.data() : nullptr);
}

/// <summary>
/// Returns the array of delta weights for this neuron.
/// </summary>
float *getDeltaWeights()
{
  return (m_deltaWeights.size() > 0 ?
    m_deltaWeights.data() : nullptr);
}

/// <summary>
/// Scales all the delta weights for this neuron by the momentum.
/// </summary>
void ScaleDeltaWeights()
{
  ScaleDeltaWeights(Momentum);
}

/// <summary>
/// Scales all the delta weights for this neuron by the specified
    ↪ value.
/// </summary>
void ScaleDeltaWeights(float scale)
```

```
    {
      cblas_sscal((int)getWeightCount(), scale, getDeltaWeights(), 1);
    }


    /// <summary>
    /// Returns an array of constant weights for the incoming
        ↪ connections to this neuron.
    /// </summary>
    const float *getWeights() const
    {
      return (m_weights.size() > 0 && m_uiNumWeights > 0 ?
        m_weights.data() + m_uiStartWeight : nullptr);
    }

    /// <summary>
    /// Sets weights for the incoming connections to this neuron to the
        ↪ values
    /// specified by the input array.
    /// </summary>
    void setWeights(const float *input)
    {
      #pragma omp parallel for
      for (int i = 0; i < (int)m_uiNumWeights; i++)
        setWeight(i, input[i]);
    }

    /// <summary>
    /// Returns the weight of the incoming connection with specified
        ↪ index.
    /// </summary>
    float getWeight(std::size_t index) const
    {
      return m_weights[m_uiStartWeight + index];
    }

    /// <summary>
    /// Sets the weight of the incoming connection with specified index.
    /// </summary>
    void setWeight(std::size_t index, float value)
    {
      m_weights.store(m_uiStartWeight + index, value);
    }

    /// <summary>
    /// Updates the weight of the incoming connection with specified
        ↪ index
    /// by adding the product of the negative learning rate and the
        ↪ value
    /// of the corresponding delta weight.
    /// </summary>
```

```
void UpdateWeight(std::size_t index, float fInvMinibatchSize = 1.0,
    ↪ float fInvTotalSamples = 1.0)
{
  m_weights.linear_transform(m_uiStartWeight + index,
    1 - LearningRate * Regularization * fInvTotalSamples,
    -LearningRate * m_deltaWeights[index] * fInvMinibatchSize);
}

/// <summary>
/// Updates the weights of all the incoming connection
/// by adding the product of the negative learning rate and the
    ↪ value
/// of the corresponding delta weight.
/// </summary>
void UpdateWeights(float fInvMinibatchSize = 1.0, float
    ↪ fInvTotalSamples = 1.0)
{
  #pragma omp parallel for
  for (int i = 0; i < (int)m_uiNumWeights; i++)
    UpdateWeight(i, fInvMinibatchSize, fInvTotalSamples);
}


void setLearningRate(float value)
{
  LearningRate = value;
  if (LearningRate < MomentumThreshold)
    Momentum = FinalMomentum;
}

void ScaleLearningRate(float scale)
{
  setLearningRate(LearningRate * scale);
}

/// <summary>
/// Learning rate for this neuron.
/// </summary>
float LearningRate;
/// <summary>
/// Final momentum for this neuron.
/// </summary>
float FinalMomentum;
/// <summary>
/// Threshold for momentum for this neuron.
/// </summary>
float MomentumThreshold;
/// <summary>
/// Momentum for this neuron.
/// </summary>
float Momentum;
```

```
    /// <summary>
    /// Regularization parameter for this neuron.
    /// </summary>
    float Regularization;

    /// <summary>
    /// Delta bias for this neuron.
    /// </summary>
    float DeltaBias;

  private:
    std::vector<float> &m_activation;
    KUtilities::atomic_vector<float> &m_biases;
    std::size_t m_uiBiasIndex;
    std::vector<float> &m_weighted_inputs;
    KUtilities::atomic_vector<float> &m_deltas;
    std::vector<float> m_deltaWeights;
    KUtilities::atomic_vector<float> &m_weights;
    std::size_t m_uiStartWeight;
    std::size_t m_uiNumWeights;
    std::size_t m_uiLayerIndex;
    std::size_t m_uiLayerNeuronCount;
    std::size_t m_uiMinibatchSize;
  };

  typedef Neuron32 Neuron;
}

#endif
```

B.23 File: knnkeys.h

```
#ifndef K_FFANN_FRIEND_KEYS_H
#define K_FFANN_FRIEND_KEYS_H

namespace NeuralNetwork
{
  /// <summary>
  /// Defines the friend key for the Network class.
  /// </summary>
  class NetworkKey
  {
    friend class Loader32;

    NetworkKey(const NetworkKey&) = delete;
    NetworkKey& operator =(const NetworkKey&) = delete;

    NetworkKey() { }
```

```
};

/// <summary>
/// Defines the friend key for the Layer class.
/// </summary>
class LayerKey
{
  friend class Network32;

  LayerKey(const LayerKey&) = delete;
  LayerKey& operator =(const LayerKey&) = delete;

  LayerKey() { }
};

/// <summary>
/// Defines the friend key for the Neuron class.
/// </summary>
class NeuronKey
{
  friend class Layer32;

  NeuronKey(const NeuronKey&) = delete;
  NeuronKey& operator =(const NeuronKey&) = delete;

  NeuronKey() { }
};


/// <summary>
/// Defines the friend key for the ActivationFunction class.
/// </summary>
class ActivationFunctionKey
{
  friend class Network32;

  ActivationFunctionKey(const ActivationFunctionKey&) = delete;
  ActivationFunctionKey& operator =(const ActivationFunctionKey&) =
     ↪ delete;

  ActivationFunctionKey() { }
};

namespace Convolutional
{

  /// <summary>
  /// Defines the friend key for the Feature class.
  /// </summary>
  class FeatureKey
  {
```

```
      friend class ConvolutionalLayer32;

      FeatureKey(const FeatureKey&) = delete;
      FeatureKey& operator =(const FeatureKey&) = delete;

      FeatureKey() { }
    };
  }
}

#endif
```

B.24 File: kopencl.h

```
#ifndef K_OPEN_CL_UTILS__H
#define K_OPEN_CL_UTILS__H

#include <cstdint>
#include <cl/cl.hpp>

namespace KNNUtils
{
  /// <summary>
  /// This class encapsulates OpenCL functions for
  /// sparse matrix operations.
  /// </summary>
  class KBLASOpenCL
  {
  public:
    /// <summary>
    /// Initializes OpenCL to be used by this class.
    /// </summary>
    static void Init(bool bUseDefault = true);

    /// <summary>
    /// Computes R = A * D, where A is a CSR sparse matrix and D is a
      ↪ dense matrix.
    /// </summary>
    static void scsrmm(float *Result,
      uint32_t nColsResult,
      bool bTransposeA,
      const float *A, const uint32_t *col_index_A, const uint32_t *
        ↪ row_ptr_A,
      uint32_t nRowsA, uint32_t nColsA,
      const float *D);

  private:
```

```
    static const std::string m_sSource;

    /// <summary>
    /// Initializes the platform.
    /// </summary>
    static void InitClPlatform(bool bUseDefault);

    /// <summary>
    /// Initializes the device to be used for computations preferring
    /// GPUs over CPUs if any OpenCL-capable GPU is present.
    /// </summary>
    static bool InitClDevice(bool bUseDefault);

    static bool m_bInitialized;

    static cl::Platform m_platform;
    static std::vector<cl::Device> m_arrDevices;
    static cl::Context m_context;
    static cl::CommandQueue m_queue;
    static cl::Program m_program;
    static cl::Kernel m_kernel_scsrmm;
    static cl::Kernel m_kernel_scsrmm_t;
  };

}

#endif
```

B.25 File: kopencl.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>
#include <stdexcept>

#define __CL_ENABLE_EXCEPTIONS
#include "kopencl.h"

bool KNNUtils::KBLASOpenCL::m_bInitialized = false;
cl::Platform KNNUtils::KBLASOpenCL::m_platform;
std::vector<cl::Device> KNNUtils::KBLASOpenCL::m_arrDevices;
cl::Context KNNUtils::KBLASOpenCL::m_context;
cl::CommandQueue KNNUtils::KBLASOpenCL::m_queue;
cl::Program KNNUtils::KBLASOpenCL::m_program;
cl::Kernel KNNUtils::KBLASOpenCL::m_kernel_scsrmm;
cl::Kernel KNNUtils::KBLASOpenCL::m_kernel_scsrmm_t;
```

```
// OpenCL code: modified from ViennaCL sparse matrix
//              multiplication with added optimizations
const std::string KNNUtils::KBLASOpenCL::m_sSource =
"__kernel void trans_mat_mult(___global const unsigned int *
    ↪ sp_mat_row_indices,\n"
"____global const unsigned int *sp_mat_col_indices,\n"
"____global const float *sp_mat_elements,\n"
"____global const float *d_mat,\n"
"__unsigned int d_mat_row_start,\n"
"__unsigned int d_mat_col_start,\n"
"__unsigned int d_mat_row_inc,\n"
"__unsigned int d_mat_col_inc,\n"
"__unsigned int d_mat_row_size,\n"
"__unsigned int d_mat_col_size,\n"
"__unsigned int d_mat_internal_rows,\n"
"__unsigned int d_mat_internal_cols,\n"
"____global float *result,\n"
"__unsigned int result_row_start,\n"
"__unsigned int result_col_start,\n"
"__unsigned int result_row_inc,\n"
"__unsigned int result_col_inc,\n"
"__unsigned int result_row_size,\n"
"__unsigned int result_col_size,\n"
"__unsigned int result_internal_rows,\n"
"__unsigned int result_internal_cols)\n"
"{\n"
"__for (unsigned int row = get_group_id(0); row < result_row_size; row
    ↪ += get_num_groups(0))\n"
"__{\n"
"____unsigned int row_start = sp_mat_row_indices[row];\n"
"____unsigned int row_end = sp_mat_row_indices[row+1];\n"
"____for (unsigned int col = get_local_id(0); col < result_col_size; col
    ↪ += get_local_size(0))\n"
"____{\n"
"______float r = 0;\n"
"______for (unsigned int k = row_start; k < row_end; k++)\n"
"______{\n"
"________unsigned int j = sp_mat_col_indices[k];\n"
"________float x = sp_mat_elements[k];\n"
"________float y = d_mat[ (d_mat_row_start + col * d_mat_row_inc) *
    ↪ d_mat_internal_cols + d_mat_col_start + j * d_mat_col_inc ];\n"
"________r += x * y;\n"
"______}\n"
"______result[ (result_row_start + row * result_row_inc) *
    ↪ result_internal_cols + result_col_start + col * result_col_inc ] =
    ↪ r;\n"
"____}\n"
"__}\n"
"}\n"
"\n"
"\n"
```

```
"__kernel void mat_mult(___global const unsigned int *
    ↪ sp_mat_row_indices,\n"
"____global const unsigned int *sp_mat_col_indices,\n"
"____global const float *sp_mat_elements,\n"
"____global const float *d_mat,\n"
"__unsigned int d_mat_row_start,\n"
"__unsigned int d_mat_col_start,\n"
"__unsigned int d_mat_row_inc,\n"
"__unsigned int d_mat_col_inc,\n"
"__unsigned int d_mat_row_size,\n"
"__unsigned int d_mat_col_size,\n"
"__unsigned int d_mat_internal_rows,\n"
"__unsigned int d_mat_internal_cols,\n"
"____global float *result,\n"
"__unsigned int result_row_start,\n"
"__unsigned int result_col_start,\n"
"__unsigned int result_row_inc,\n"
"__unsigned int result_col_inc,\n"
"__unsigned int result_row_size,\n"
"__unsigned int result_col_size,\n"
"__unsigned int result_internal_rows,\n"
"__unsigned int result_internal_cols)\n"
"{\n"
"__for (unsigned int row = get_group_id(0); row < result_row_size; row
    ↪ += get_num_groups(0))\n"
"__{\n"
"____unsigned int row_start = sp_mat_row_indices[row];\n"
"____unsigned int row_end = sp_mat_row_indices[row+1];\n"
"____for (unsigned int col = get_local_id(0); col < result_col_size; col
    ↪ += get_local_size(0))\n"
"____{\n"
"______float r = 0;\n"
"______for (unsigned int k = row_start; k < row_end; k++)\n"
"______{\n"
"________unsigned int j = sp_mat_col_indices[k];\n"
"________float x = sp_mat_elements[k];\n"
"________float y = d_mat[ (d_mat_row_start + j * d_mat_row_inc) *
    ↪ d_mat_internal_cols + d_mat_col_start + col * d_mat_col_inc ];\n"
"________r += x * y;\n"
"______}\n"
"______result[ (result_row_start + row * result_row_inc) *
    ↪ result_internal_cols + result_col_start + col * result_col_inc ] =
    ↪ r;\n"
"____}\n"
"__}\n"
"}\n"
;

void KNNUtils::KBLASOpenCL::Init(bool bUseDefault)
{
  if (m_sSource.empty())
```

```
    throw std::runtime_error("Invalid OpenCL source code.");

  InitClPlatform(bUseDefault);
  InitClDevice(bUseDefault);

  m_context = cl::Context(m_arrDevices[0]);

  m_queue = cl::CommandQueue(m_context, m_arrDevices[0]);


  cl::Program::Sources source;
  source.emplace_back(m_sSource.c_str(), m_sSource.length());

  m_program = cl::Program(m_context, source);
  m_program.build(m_arrDevices);

  m_kernel_scsrmm = cl::Kernel(m_program, "mat_mult");
  m_kernel_scsrmm_t = cl::Kernel(m_program, "trans_mat_mult");

  m_bInitialized = true;
}

void KNNUtils::KBLASOpenCL::scsrmm(
  float *Result,
  uint32_t nColsResult,
  bool bTransposeA,
  const float *A, const uint32_t *col_index_A, const uint32_t *row_ptr_A
    ↪ ,
  uint32_t nRowsA, uint32_t nColsA,
  const float *D)
{

  if (!m_bInitialized)
    Init(true);

  std::size_t sizeRowPtrA = nRowsA + 1;
  std::size_t sizeColIndexA = row_ptr_A[nRowsA];
  std::size_t sizeA = sizeColIndexA;
  uint32_t &nRowsD = (bTransposeA ? nRowsA : nColsA);
  uint32_t &nColsD = nColsResult;
  uint32_t &nRowsResult = (bTransposeA ? nColsA : nRowsA);
  std::size_t sizeResult = nRowsResult * nColsResult;

  std::size_t size = ((sizeResult >> 6) + (sizeResult & 63 ? 1 : 0)) <<
    ↪ 6;


  cl::Buffer dSparseRowPtr_A;
  cl::Buffer dSparseColIndices_A;
  cl::Buffer dSparseNNZ_A;
  cl::Buffer dDenseMat_D;
```

```
  cl::Buffer dDenseMat_Result;


  dSparseNNZ_A = cl::Buffer(m_context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeA * sizeof(float),
    (void*)A);

  dSparseColIndices_A = cl::Buffer(m_context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeColIndexA * sizeof(uint32_t),
    (void*)col_index_A);

  dSparseRowPtr_A = cl::Buffer(m_context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeRowPtrA * sizeof(uint32_t),
    (void*)row_ptr_A);


  dDenseMat_D = cl::Buffer(m_context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeof(float) * nRowsD * nColsD,
    (void*)D);


  dDenseMat_Result = cl::Buffer(m_context,
    CL_MEM_WRITE_ONLY, sizeResult * sizeof(float));

  cl::Kernel &kernel = (bTransposeA ? m_kernel_scsrmm_t :
    ↪ m_kernel_scsrmm);

  kernel.setArg(0, dSparseRowPtr_A);
  kernel.setArg(1, dSparseColIndices_A);
  kernel.setArg(2, dSparseNNZ_A);

#define offset 2
  kernel.setArg(5 - offset, dDenseMat_D);
  kernel.setArg(6 - offset, 0);
  kernel.setArg(7 - offset, 0);
  kernel.setArg(8 - offset, 1);
  kernel.setArg(9 - offset, 1);
  kernel.setArg(10 - offset, nRowsD);
  kernel.setArg(11 - offset, nColsD);
  kernel.setArg(12 - offset, nRowsD);
  kernel.setArg(13 - offset, nColsD);
  kernel.setArg(14 - offset, dDenseMat_Result);
  kernel.setArg(15 - offset, 0);
  kernel.setArg(16 - offset, 0);
  kernel.setArg(17 - offset, 1);
  kernel.setArg(18 - offset, 1);
  kernel.setArg(19 - offset, nRowsResult);
```

```
  kernel.setArg(20 - offset, nColsResult);
  kernel.setArg(21 - offset, nRowsResult);
  kernel.setArg(22 - offset, nColsResult);

  m_queue.enqueueNDRangeKernel(kernel,
    cl::NDRange(),
    cl::NDRange(size),
    cl::NDRange(64));

  m_queue.enqueueReadBuffer(dDenseMat_Result,
    CL_TRUE,
    0, sizeResult * sizeof(float),
    (void*)Result);
}

void KNNUtils::KBLASOpenCL::InitClPlatform(bool bUseDefault)
{
  int iChoice = 0;
  std::vector<cl::Platform> arrPlatforms;

  try
  {
    cl::Platform::get(&arrPlatforms);
  }
  catch (...)
  {
    throw std::runtime_error("Could not retrieve OpenCL platforms in
      ↪ this system.");
  }

  if (arrPlatforms.size() > 1 && !bUseDefault)
  {
    std::cout << "Select platform:" << std::endl;
    for (std::size_t i = 0; i < arrPlatforms.size(); i++)
    {
      std::string sPlatformName;
      arrPlatforms[i].getInfo(CL_PLATFORM_NAME, &sPlatformName);
      std::cout << "\t" << i + 1 << ". " << sPlatformName << std::endl;
    }
    std::cin >> iChoice;
    if (iChoice < 1 || iChoice > arrPlatforms.size())
      throw std::runtime_error("Invalid platform choice.");
    iChoice--;
  }
  else if (arrPlatforms.empty())
    throw std::runtime_error("No OpenCL platforms detected in this
      ↪ system.");

  m_platform = arrPlatforms[iChoice];
}
```

```
bool KNNUtils::KBLASOpenCL::InitClDevice(bool bUseDefault)

{
  bool retval = true;
  int iChoice = 0;
  std::vector<cl::Device> arrDevices;

  try
  {
    m_platform.getDevices(CL_DEVICE_TYPE_GPU, &arrDevices);
  }
  catch (...)
  {
    arrDevices.clear();
  }
  if (arrDevices.empty())
  {
    retval = false;
    try
    {
      m_platform.getDevices(CL_DEVICE_TYPE_CPU, &arrDevices);
    }
    catch (...)
    {
      throw std::runtime_error("Could not find OpenCL-capable devices.")
          ↪ ;
    }
  }

  if (arrDevices.size() > 1 && !bUseDefault)
  {
    std::cout << "Select OpenCL device:" << std::endl;
    for (std::size_t device_i = 0; device_i < arrDevices.size();
        ↪ device_i++)
    {
      std::string sDeviceName;
      arrDevices[device_i].getInfo(CL_DEVICE_NAME, &sDeviceName);
      std::cout << "\t" << device_i + 1 << ". " << sDeviceName << std::
          ↪ endl;
    }
    std::cin >> iChoice;
    if (iChoice < 1 || iChoice > arrDevices.size())
      throw std::runtime_error("Invalid device choice.");
    iChoice--;
  }

  m_arrDevices.resize(1, arrDevices[iChoice]);

  return retval;
}
```

B.26 File: ksafevector.h

```
#ifndef K_UTILITIES_SAFE_VECTOR__H
#define K_UTILITIES_SAFE_VECTOR__H

#include <vector>
#include <mutex>
#include <memory>
#include <algorithm>
#include <utility>
#include <cstring>
#include <cstdint>
#include <iterator>

namespace KUtilities
{
  template <class T> class atomic_vector_iter;

  /// <summary>
  ///
  /// </summary>
  template <class T>
  class atomic_vector
  {
    friend class atomic_vector_iter<T>;

  public:

    typedef atomic_vector_iter<T> const_iterator;
    typedef std::ptrdiff_t difference_type;
    typedef std::intptr_t pointer_type;
    typedef std::size_t size_type;
    typedef T value_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;

    template <class iterT>
    class atomic_vector_iter
    {
    public:
      atomic_vector_iter(atomic_vector<iterT> &v, size_type position) :
        m_atomic_vector(v), m_position(position) { }

      /// <summary>
      /// Copy constructor.
      /// </summary>
      atomic_vector_iter(const atomic_vector_iter<iterT> &src) :
```

```
      m_atomic_vector(src.m_atomic_vector), m_position(src.m_position)
        ↪ { }

    friend bool operator==(const atomic_vector_iter<iterT> &lhs, const
      ↪ atomic_vector_iter<iterT> &rhs)
    {
      if (&lhs == &rhs)
        return true;
      else
        return (&(lhs.m_atomic_vector) == &(rhs.m_atomic_vector)) &&
        (lhs.m_position == rhs.m_position);
    }
    friend bool operator!=(const atomic_vector_iter<iterT> &lhs, const
      ↪ atomic_vector_iter<iterT> &rhs)
    {
      return !(lhs == rhs);
    }

    const iterT& operator *() const
    {
      return m_atomic_vector.at(m_position);
    }

    atomic_vector_iter<iterT>& operator++()
    {
      ++m_position;
      return *this;
    }

    atomic_vector_iter<iterT> operator++(int)
    {
      return atomic_vector_iter<iterT>(m_atomic_vector, m_position++);
    }

  private:
    size_type m_position;
    atomic_vector<iterT> &m_atomic_vector;
  };


  /// <summary>
  /// Default constructor. Constructs an empty vector.
  /// </summary>
  atomic_vector()
  { }

  /// <summary>
  /// Copy constructor.
  /// </summary>
  atomic_vector(const atomic_vector<T> &src)
  {
```

```
    reserve(src.capacity());
    resize(src.size());
    m_arr = src.m_arr;
  }

  /// <summary>
  /// Move constructor.
  /// </summary>
  atomic_vector(atomic_vector<T> &&src)
  {
    m_arrMutex = std::move(src.m_arrMutex);
    m_arr = std::move(src.m_arr);
  }

  /// <summary>
  /// Move constructor.
  /// </summary>
  atomic_vector(std::vector<T> &&src)
  {
    reserve(src.capacity());
    resize(src.size());
    m_arr = std::move(src.m_arr);
  }

  /// <summary>
  /// Fill constructor. Constructs a new vector with the specified
  /// number of elements (the elements are default constrcuted).
  /// </summary>
  atomic_vector(std::size_t size)
  {
    if (size > 0)
    {
      reserve(size);
      resize(size);
    }
  }

  /// <summary>
  /// Fill constructor. Constructs a new vector with the specified
  /// number of elements (the elements are copy constrcuted).
  /// </summary>
  atomic_vector(size_type size, const_reference value)
  {
    if (size > 0)
    {
      reserve(size);
      resize(size, value);
    }
  }

  atomic_vector(size_type size, const T *array)
```

```
  {
    assign(size, array);
  }

  atomic_vector(const_pointer start, const_pointer end)
  {
    assign(start, end);
  }

  /// <summary>
  /// Range constructor.
  /// </summary>
  template <class InputIter>
  atomic_vector(InputIter start, InputIter end)
  {
    assign(start, end);
  }

  /// <summary>
  /// Assignment operator.
  /// </summary>
  atomic_vector<T>& operator =(const atomic_vector<T> &src)
  {
    if (this != &src)
    {
      reserve(src.capacity());
      resize(src.size());
      m_arr = src.m_arr;
    }
    return *this;
  }

  /// <summary>
  /// Assignment operator.
  /// </summary>
  atomic_vector<T>& operator =(const std::vector<T> &src)
  {
    reserve(src.capacity());
    resize(src.size());
    m_arr = src;
    return *this;
  }

  /// <summary>
  /// Move assignment operator.
  /// </summary>
  atomic_vector<T>& operator= (atomic_vector<T>&& src)
  {
    if (this != &src)
    {
      m_arrMutex = std::move(src.m_arrMutex);
```

```
      m_arr = std::move(src.m_arr);
    }
    return *this;
  }

  /// <summary>
  /// Move assignment operator.
  /// </summary>
  atomic_vector<T>& operator= (std::vector<T>&& src)
  {
    reserve(src.capacity());
    resize(src.size());
    m_arr = std::move(src);
    return *this;
  }



  /// <summary>
  /// Returns an input iterator pointing to the first element in the
    ↪ vector.
  /// </summary>
  const_iterator begin() const
  {
    return const_iterator(*this, 0);
  }

  /// <summary>
  /// Returns an input iterator pointing past the last element in the
    ↪ vector.
  /// </summary>
  const_iterator end() const
  {
    return const_iterator(*this, size());
  }



  /// <summary>
  /// Returns the number of elements in the vector.
  /// </summary>
  size_type size() const
  {
    return m_arr.size();
  }

  /// <summary>
  /// Returns the maximum number of elements that the vector can hold.
  /// </summary>
  size_type max_size() const
  {
```

```
    return std::min(m_arr.max_size(), m_arrMutex.max_size());
  }

  /// <summary>
  /// Resizes the container so that it contains n elements.
  /// </summary>
  void resize(size_type n)
  {
    size_type old_size = m_arr.size();

    m_arr.resize(n);
    m_arrMutex.resize(n);

    for (size_type i = old_size; i < m_arrMutex.size(); i++)
      m_arrMutex[i].reset(new std::mutex());
  }

  /// <summary>
  /// Resizes the container so that it contains n elements.
  /// New elements are initialized to specified value.
  /// </summary>
  void resize(size_type n, const_reference value)
  {
    size_type old_size = m_arr.size();

    m_arr.resize(n, val);
    m_arrMutex.resize(n);

    for (int i = old_size; i < m_arrMutex.size(); i++)
      m_arrMutex[i].reset(new std::mutex());
  }

  /// <summary>
  /// Returns the size of the storage space currently allocated
  /// for the vector, expressed in terms of elements.
  /// </summary>
  size_type capacity() const
  {
    return std::min(m_arr.capacity(), m_arrMutex.capacity());
  }

  /// <summary>
  /// Returns whether the vector is empty (i.e. whether its size is 0)
      ↪ .
  /// </summary>
  bool empty() const
  {
    return m_arr.empty();
  }

  /// <summary>
```

```
    /// Requests that the vector capacity be at least enough to contain
        ↪ n elements.
    /// </summary>
    void reserve(size_type n)
    {
      m_arr.reserve(n);
      m_arrMutex.reserve(n);
    }

    /// <summary>
    /// Requests the container to reduce its capacity to fit its size.
    /// </summary>
    void shrink_to_fit()
    {
      m_arr.shrink_to_fit(n);
      m_arrMutex.shrink_to_fit(n);
    }



    /// <summary>
    /// Retrieves the value of the specified element in the vector.
    /// </summary>
    const_reference operator[](size_type index) const
    {
      return m_arr[index];
    }

    /// <summary>
    /// Retrieves the value of the specified element in the vector (
        ↪ alias of at()).
    /// </summary>
    const_reference load(size_type index) const
    {
      return at(index);
    }

    /// <summary>
    /// Retrieves the value of the specified element in the vector.
    /// </summary>
    const_reference at(size_type index) const
    {
      return m_arr.at(index);
    }

    /// <summary>
    /// Sets the value of the specified element in the vector.
    /// </summary>
    void store(size_type index, const_reference value)
    {
      std::unique_lock<std::mutex> mutexLock(*m_arrMutex.at(index));
```

```
    m_arr[index] = value;
  }

  /// <summary>
  /// Performs a compare and store operation.
  /// </summary>
  bool compare_store(size_type index, const_reference expected,
      ↪ const_reference value)
  {
    bool retval = false;

    std::unique_lock<std::mutex> mutexLock(*m_arrMutex.at(index));
    if (retval = (std::memcmp(&expected, &m_arr[index], sizeof(
        ↪ expected)) == 0))
      m_arr[index] = value;

    return retval;

  }

  /// <summary>
  /// Performs a compare and exchange operation.
  /// </summary>
  void compare_xchange(size_type index, reference expected,
      ↪ const_reference value, reference out)
  {
    bool retval = false;

    std::unique_lock<std::mutex> mutexLock(*m_arrMutex.at(index));
    if (retval = (std::memcmp(&expected, &m_arr[index], sizeof(
        ↪ expected)) == 0))
      m_arr[index] = value;
    else
      expected = m_arr[index];

    return retval;

  }

  /// <summary>
  /// Atomically performs a linear transformation on the specified
      ↪ element in the vector.
  /// </summary>
  void linear_transform(size_type index, const_reference scale,
      ↪ const_reference value)
  {
    std::unique_lock<std::mutex> mutexLock(*m_arrMutex.at(index));
    m_arr[index] = scale * m_arr[index] + value;
  }

  /// <summary>
```

```
    /// Adds value to the specified element in the vector.
    /// </summary>
    void accumulate(size_type index, const_reference value)
    {
      std::unique_lock<std::mutex> mutexLock(*m_arrMutex.at(index));
      m_arr[index] += value;
    }

    /// <summary>
    /// Sets the specified element in the vector to the multiplication
       ↪ of
    /// itself times value.
    /// </summary>
    void multiply(size_type index, const_reference value)
    {
      std::unique_lock<std::mutex> mutexLock(*m_arrMutex.at(index));
      m_arr[index] *= value;
    }

    /// <summary>
    /// Retrieves the value of the first element in the vector.
    /// </summary>
    const_reference front() const
    {
      return m_arr.front();
    }

    /// <summary>
    /// Sets the value of the first element in the vector.
    /// </summary>
    void front_store(const_reference value)
    {
      std::unique_lock<std::mutex> mutexLock(*m_arrMutex.front());
      m_arr.front() = value;
    }

    /// <summary>
    /// Adds value to the first element in the vector.
    /// </summary>
    void front_accumulate(const_reference value)
    {
      std::unique_lock<std::mutex> mutexLock(*m_arrMutex.front());
      m_arr.front() += value;
    }

    /// <summary>
    /// Sets the first element in the vector to the multiplication of
    /// itself times value.
    /// </summary>
    void front_multiply(const_reference value)
    {
```

```
    std::unique_lock<std::mutex> mutexLock(*m_arrMutex.front());
    m_arr.front() *= value;
  }

  /// <summary>
  /// Retrieves the value of the last element in the vector.
  /// </summary>
  const_reference back() const
  {
    return m_arr.back();
  }

  /// <summary>
  /// Sets the value of the last element in the vector.
  /// </summary>
  void back_store(const_reference value)
  {
    std::unique_lock<std::mutex> mutexLock(*m_arrMutex.back());
    m_arr.back() = value;
  }

  /// <summary>
  /// Adds value to the last element in the vector.
  /// </summary>
  void back_accumulate(const_reference value)
  {
    std::unique_lock<std::mutex> mutexLock(*m_arrMutex.back());
    m_arr.back() += value;
  }

  /// <summary>
  /// Sets the last element in the vector to the multiplication of
  /// itself times value.
  /// </summary>
  void back_multiply(const_reference value)
  {
    std::unique_lock<std::mutex> mutexLock(*m_arrMutex.back());
    m_arr.back() *= value;
  }

  /// <summary>
  /// Returns a direct pointer to the memory array used internally
  /// by the vector to store its owned elements.
  /// </summary>
  const T *data() const
  {
    return m_arr.data();
  }

  /// <summary>
  /// Returns a direct pointer to the memory array used internally
```

```
    /// by the vector to store its owned elements.
    /// </summary>
    T *unsafe_data()
    {
      return m_arr.data();
    }



    /// <summary>
    /// Assigns new contents to the vector, replacing its current
    /// contents, and modifying its size accordingly.
    /// </summary>
    void assign(size_type size, const T *array)
    {
      this->resize(size);
      #pragma omp parallel for
      for (size_type i = 0; i < size; ++i)
        m_arr[i] = array[i];
    }

    /// <summary>
    /// Assigns new contents to the vector, replacing its current
    /// contents, and modifying its size accordingly.
    /// </summary>
    template <class InputIter>
    void assign(InputIter start, InputIter end)
    {
      this->resize(std::distance(start, end));
      auto my_it = m_arr.begin();
      for (InputIter it = start; it != end; ++it, ++my_it)
        *my_it = *it;
    }

    /// <summary>
    /// Assigns new contents to the vector, replacing its current
    /// contents, and modifying its size accordingly.
    /// </summary>
    void assign(const_pointer start, const_pointer end)
    {
      assign((size_type)std::distance(start, end), start);
    }

    /// <summary>
    /// Assigns new contents to the vector, replacing its current
    /// contents, and modifying its size accordingly.
    /// </summary>
    void assign(size_type n, const_reference value)
    {
      this->resize(n);
      m_arr.assign(n, value);
```

```
    }

    /// <summary>
    /// Removes all elements from the vector (which are destroyed),
    /// leaving the container with a size of 0.
    /// </summary>
    void clear()
    {
      m_arr.clear();
      m_arrMutex.clear();
    }

    /// <summary>
    /// Adds a new element at the end of the vector, after its current
    /// last element.The content of value is copied to the new element.
    /// </summary>
    void push_back(const_reference value)
    {
      std::shared_ptr<std::mutex> pmutex(new std::mutex());
      std::unique_lock<std::mutex> mutexLock(*pmutex);
      m_arrMutex.push_back(pmutex);

      m_arr.push_back(value);

    }

    /// <summary>
    /// Removes and destroys the last element in the vector, effectively
    /// reducing the container size by one.
    /// </summary>
    void pop_back()
    {
      m_arrMutex.pop_back();
      m_arr.pop_back();
    }

  protected:
    std::vector<T> m_arr;
    std::vector<std::shared_ptr<std::mutex>> m_arrMutex;
  };


  /// <summary>
  /// Assigns new contents to the vector, replacing its current
  /// contents, and modifying its size accordingly.
  /// </summary>
  template<>
  inline void atomic_vector<double>::assign(size_type size, const double
      ↪ *array)
  {
    this->resize(size);
```

```
  std::memcpy(m_arr.data(), array, size * sizeof(double));
}

/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
/// </summary>
template<>
inline void atomic_vector<float>::assign(size_type size, const float *
    ↪ array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(float));
}

/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
/// </summary>
template<>
inline void atomic_vector<std::int8_t>::assign(size_type size, const
    ↪ std::int8_t *array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(std::int8_t));
}

/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
/// </summary>
template<>
inline void atomic_vector<std::int16_t>::assign(size_type size, const
    ↪ std::int16_t *array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(std::int16_t));
}

/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
/// </summary>
template<>
inline void atomic_vector<std::int32_t>::assign(size_type size, const
    ↪ std::int32_t *array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(std::int32_t));
}
```

```
/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
/// </summary>
template<>
inline void atomic_vector<std::int64_t>::assign(size_type size, const
    ↪ std::int64_t *array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(std::int64_t));
}

/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
/// </summary>
template<>
inline void atomic_vector<std::uint8_t>::assign(size_type size, const
    ↪ std::uint8_t *array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(std::uint8_t));
}

/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
/// </summary>
template<>
inline void atomic_vector<std::uint16_t>::assign(size_type size, const
    ↪ std::uint16_t *array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(std::uint16_t));
}

/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
/// </summary>
template<>
inline void atomic_vector<std::uint32_t>::assign(size_type size, const
    ↪ std::uint32_t *array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(std::uint32_t));
}

/// <summary>
/// Assigns new contents to the vector, replacing its current
/// contents, and modifying its size accordingly.
```

```
/// </summary>
template<>
inline void atomic_vector<std::uint64_t>::assign(size_type size, const
    ↪ std::uint64_t *array)
{
  this->resize(size);
  std::memcpy(m_arr.data(), array, size * sizeof(std::uint64_t));
}

/// <summary>
/// Replicates the first small_size elements of the array
/// by copying them to fill up the whole array.
/// </summary>
inline void replicate_array(double * arr, std::size_t small_size, std
    ↪ ::size_t array_size)
{
  if (arr && small_size < array_size)
  {
    std::size_t next_copy_point = small_size;
    while (array_size > (next_copy_point << 1) - 1)
    {
      std::memcpy(arr + next_copy_point, arr, next_copy_point * sizeof
          ↪ (double));

      next_copy_point <<= 1;
    }

    if (array_size > next_copy_point)
      std::memcpy(arr + next_copy_point, arr, (array_size -
          ↪ next_copy_point) * sizeof(double));
  }
}

/// <summary>
/// Replicates the first small_size elements of the array
/// by copying them to fill up the whole array.
/// </summary>
inline void replicate_array(float * arr, std::size_t small_size, std::
    ↪ size_t array_size)
{
  if (arr && small_size < array_size)
  {
    std::size_t next_copy_point = small_size;
    while (array_size >(next_copy_point << 1) - 1)
    {
      std::memcpy(arr + next_copy_point, arr, next_copy_point * sizeof
          ↪ (float));

      next_copy_point <<= 1;
    }
```

```
      if (array_size > next_copy_point)
        std::memcpy(arr + next_copy_point, arr, (array_size -
            ↪ next_copy_point) * sizeof(float));
    }
  }
}

#endif
```

B.27 File: ktrainer.h

```
#ifndef K_FFANN_TRAINER_H
#define K_FFANN_TRAINER_H

#include <ostream>

#include "knetwork.h"
#include "kfclayer.h"
#include "kconvlayer.h"
#include "kmaxpoollayer.h"

#include "ktypeutils.h"

namespace NeuralNetwork
{
  class Trainer32
  {
  public:

    class IObjectiveFunction
    {
    public:

      IObjectiveFunction()
      { }

      virtual ~IObjectiveFunction()
      { }

      IObjectiveFunction(const IObjectiveFunction&) = delete;
      IObjectiveFunction& operator=(const IObjectiveFunction&) = delete;


      /// <summary>
      /// Computes the error value of the objective function
      /// based on the input values and the target values.
      /// </summary>
```

```
    virtual float error(const float *values, const float *target, std
        ↪ ::size_t size) = 0;

    /// <summary>
    /// Returns the partial derivative of the objective function
    /// with respect to the objective function's input vector's
    /// element index-th.
    /// </summary>
    virtual void derror(float *result, const float *values, const
        ↪ float *target, std::size_t size) = 0;
};


class ITrainingInput
{
public:
    virtual void ReadInput(float *input, std::size_t input_size,
                           float *target, std::size_t target_size,
                           std::size_t sample_index) = 0;
    virtual std::size_t getNumberOfSamples() = 0;
};

Trainer32(Network32 &network, float fInitialLearningRate) :
    Trainer32(network, fInitialLearningRate, 0.5, 0.0, 0.0, 0.0)
{ }

Trainer32(Network32 &network, float fInitialLearningRate, float
    ↪ fLearningRateReduction,
    float fMomentumThreshold, float fInitialMomentum, float
        ↪ fFinalMomentum) :
    m_network(network),
    InitialLearningRate(fInitialLearningRate),
    LearningRateReduction(fLearningRateReduction),
    MomentumThreshold(fMomentumThreshold),
    InitialMomentum(fInitialMomentum),
    FinalMomentum(fFinalMomentum),
    RandomizeSamples(true)
{ }

Trainer32(const Trainer32&) = delete;
Trainer32& operator=(const Trainer32&) = delete;

/// <summary>
/// Train the network for a full epoch.
/// </summary>
float Train(ITrainingInput &training, IObjectiveFunction &objFunc)
{
    return Train(training, objFunc, nullptr, 0);
}

/// <summary>
```

```
    /// Train the network for a full epoch.
    /// </summary>
    float Train(ITrainingInput &training, IObjectiveFunction &objFunc,
      std::ostream &log, int iReportAfterPercent)
    {
      return Train(training, objFunc, &log, iReportAfterPercent);
    }

    virtual float Test(ITrainingInput &testing, std::ostream &log);

    virtual float Backpropagation(const float *input, const float *
        ↪ target, IObjectiveFunction &objFunc);
    void PostBackpropagation(std::size_t uiTotalSamples);

    /// <summary>
    /// Resets the learning rate of all the neurons in the network to
        ↪ the
    /// value specified by InitialLearningRate member.
    /// </summary>
    void ResetLearningRate();

    float InitialLearningRate;
    float LearningRateReduction;
    float MomentumThreshold;
    float InitialMomentum;
    float FinalMomentum;

    bool RandomizeSamples;

  protected:
    /// <summary>
    /// Performs the training of the network for a complete epoch.
    /// </summary>
    virtual float Train(ITrainingInput &training, IObjectiveFunction &
        ↪ objFunc,
      std::ostream *log, int iReportAfterPercent);

  private:

    std::size_t readMinibatchData(ITrainingInput &training,
                std::vector<std::size_t> &sampleOrder,
                std::size_t uiSampleIndex,
                float *bufferInput,
                std::size_t uiSampleInputSize,
                float *bufferTarget,
                std::size_t uiSampleOutputSize);

    std::vector<float> m_buffer;

    Network32 &m_network;
  };
```

```
  typedef Trainer32 Trainer;
}

#endif
```

B.28 File: ktrainer.cpp

```
#include <iostream>

#include <stdexcept>
#include <cstring>
#include <algorithm>
#include <iterator>
#include <cstdlib>
#include <ostream>
#include <iomanip>
#include <limits>
#include "mkl.h"

#include "ktrainer.h"
#include "klayer.h"
#include "kinputlayer.h"
#include "kconvneuron.h"

#include "ksafevector.h"

std::size_t NeuralNetwork::Trainer32::readMinibatchData(ITrainingInput &
    ↪ training,
  std::vector<std::size_t> &sampleOrder, std::size_t uiSampleIndex,
  float * bufferInput, std::size_t uiSampleInputSize,
  float *bufferTarget, std::size_t uiSampleOutputSize)
{
  std::size_t uiSamplesCount = 0;

  while (uiSamplesCount < m_network.getMinibatchSize() &&
    uiSamplesCount + uiSampleIndex < sampleOrder.size())
  {
    training.ReadInput(bufferInput + uiSamplesCount * uiSampleInputSize,
        ↪ uiSampleInputSize,
      bufferTarget + uiSamplesCount * uiSampleOutputSize,
          ↪ uiSampleOutputSize,
      sampleOrder[uiSamplesCount + uiSampleIndex]);

    uiSamplesCount++;
  }

  return uiSamplesCount;
```

```
}


float NeuralNetwork::Trainer32::Train(ITrainingInput & training,
    ↪ IObjectiveFunction &objFunc,
  std::ostream *log, int iReportAfterPercent)
{

  std::vector<float> bufferInput;
  std::vector<float> bufferTarget;
  std::size_t uiTotalSamples =
    training.getNumberOfSamples() - (training.getNumberOfSamples() %
       ↪ m_network.getMinibatchSize());
  std::vector<std::size_t> sampleOrder(uiTotalSamples);

  #pragma omp parallel for
  for (int i = 0; i < (int)sampleOrder.size(); i++)
    sampleOrder[i] = i;
  if (RandomizeSamples)
    for (std::size_t i = 0; i < sampleOrder.size(); i++)
    {
      std::size_t newPosition = std::rand() % sampleOrder.size();
      std::swap(sampleOrder[i], sampleOrder[newPosition]);
    }

  std::vector<float> input(m_network.getInputLayer().
     ↪ getActivationsVector().size());
  std::vector<float> target(m_network.getOutputLayer().
     ↪ getActivationsVector().size());

  if (log)
  {
    if (iReportAfterPercent > 100 || iReportAfterPercent <= 0)
      iReportAfterPercent = 100;

    (*log) << std::setw(20) << "Progress" << std::setw(20) << "Ave_Cost"
      << std::endl;
    (*log) << std::setw(20) << "0%" << std::setw(20) << "Unknown"
      << std::endl;
    log->flush();
  }

  int iStrikes = 0;
  float fLastErrorAve = std::numeric_limits<float>::max();
  float fCurrentError = 0.0;
  float fCurrentErrorAve = 0.0;
  std::size_t uiReductionsCnt = 0;
  std::size_t uiIndexThreshold = sampleOrder.size() / 5;

  std::size_t uiTotalMinibatches = sampleOrder.size() / m_network.
     ↪ getMinibatchSize();
```

```
  std::size_t uiMinibatchCnt = 0;


  std::size_t uiSampleIndex = 0;
  while (uiSampleIndex < sampleOrder.size())
  {
    if (readMinibatchData(training,
      sampleOrder, uiSampleIndex,
      input.data(), m_network.getInputLayer().getNeuronCount(),
      target.data(), m_network.getOutputLayer().getNeuronCount())
      == m_network.getMinibatchSize())
    {
      fCurrentError += Backpropagation(input.data(), target.data(),
          ↪ objFunc);
      PostBackpropagation(uiTotalSamples);
    }

    uiSampleIndex += m_network.getMinibatchSize();
    uiMinibatchCnt++;

    if (uiSampleIndex <= sampleOrder.size())
      fCurrentErrorAve = fCurrentError / uiSampleIndex;

    if (log)
    {
      if (uiTotalMinibatches > 0 && (uiMinibatchCnt * 100 /
          ↪ iReportAfterPercent) % uiTotalMinibatches == 0)
      {
        (*log) << std::setw(19) << (uiMinibatchCnt * 100) /
            ↪ uiTotalMinibatches << "%" << std::setw(20) <<
            ↪ fCurrentErrorAve
          << std::endl;
        log->flush();
      }
    }
  }

  return fCurrentErrorAve;
}

float NeuralNetwork::Trainer32::Test(ITrainingInput & testing, std::
    ↪ ostream & log)
{

  std::size_t uiTotalSamples = testing.getNumberOfSamples() - (testing.
      ↪ getNumberOfSamples() % m_network.getMinibatchSize());
  std::size_t uiTotalMinibatches = uiTotalSamples / m_network.
      ↪ getMinibatchSize();
  std::size_t uiMinibatchCnt = 0;
  std::size_t uiNumCorrect;
  InputLayer32 &inputLayer = m_network.getInputLayer();
```

```
Layer32 &outputLayer = m_network.getOutputLayer();
std::vector<float> input(inputLayer.getActivationsVector().size());
std::vector<float> target(outputLayer.getActivationsVector().size());
std::vector<std::size_t> sampleOrder(uiTotalSamples);

#pragma omp parallel for
for (int i = 0; i < (int)sampleOrder.size(); i++)
  sampleOrder[i] = i;

log << std::endl << "Test completion percent: 0";
log.flush();

uiNumCorrect = 0;
for (std::size_t sample_i = 0; sample_i < uiTotalSamples; sample_i +=
    ↪ m_network.getMinibatchSize())
{
  if (((uiMinibatchCnt + 1) * 10) % uiTotalMinibatches == 0)
  {
    log << ", " << ((uiMinibatchCnt + 1) * 100) / uiTotalMinibatches;
    log.flush();
  }

  readMinibatchData(testing, sampleOrder, sample_i,
    input.data(), inputLayer.getNeuronCount(),
    target.data(), outputLayer.getNeuronCount());

  inputLayer.InitializeInput(input.data());

  m_network.PropagatePerLayer();


  for (std::size_t minibatch_sample_i = 0; minibatch_sample_i <
      ↪ m_network.getMinibatchSize(); minibatch_sample_i++)
  {
    const float *actualOutput = outputLayer.getActivations(
        ↪ minibatch_sample_i);
    std::size_t maxIndexOutput =
      std::distance(actualOutput, std::max_element(actualOutput,
          ↪ actualOutput + outputLayer.getNeuronCount()));
    const float *targetOutput = target.data() + minibatch_sample_i *
        ↪ outputLayer.getNeuronCount();
    std::size_t maxIndexTarget =
      std::distance(targetOutput, std::max_element(targetOutput,
          ↪ targetOutput + outputLayer.getNeuronCount()));

    if (maxIndexTarget == maxIndexOutput)
      uiNumCorrect++;
  }

  uiMinibatchCnt++;
}
```

```
  log << "%" << std::endl;
  log << "Results:" << std::endl;
  log << "Correct hits: " << uiNumCorrect << std::endl;
  log << "Total samples: " << uiTotalSamples << std::endl;
  log << "Accuracy (correct): " << 100.0 * uiNumCorrect / (float)
      ↪ uiTotalSamples << "%" << std::endl;
  log.flush();

  return uiNumCorrect / (float)uiTotalSamples;
}

float NeuralNetwork::Trainer32::Backpropagation(const float *input,
                          const float *target,
                          IObjectiveFunction &objFunc)
{
  float retval;

  if (!input)
    throw std::invalid_argument("input cannot be null.");
  if (!target)
    throw std::invalid_argument("target cannot be null.");
  if (m_network.getLayerCount() <= 0)
    throw std::invalid_argument("Network is empty.");

  m_network.getInputLayer().InitializeInput(input);
  m_network.PropagatePerLayer();

  Layer32 &outputLayer = m_network.getOutputLayer();

  retval = 0.0;
  #pragma omp parallel for reduction (+:retval)
  for (int sample_i = 0; sample_i < m_network.getMinibatchSize();
      ↪ sample_i++)
    retval += objFunc.error(outputLayer.getActivations(sample_i),
                target + sample_i * outputLayer.getNeuronCount(),
                outputLayer.getNeuronCount());

  m_buffer.resize(outputLayer.getActivationsVector().size());
  objFunc.derror(m_buffer.data(), outputLayer.getActivations(), target,
      ↪ m_buffer.size());
  outputLayer.getActivationFunction().df(outputLayer.getDeltaVector().
      ↪ unsafe_data(), outputLayer.getWeightedInputValues(), m_buffer.
      ↪ size());
  vsMul((int)m_buffer.size(), outputLayer.getDeltaVector().data(),
      ↪ m_buffer.data(), outputLayer.getDeltaVector().unsafe_data());



  for (int l = (int)m_network.getLayerCount() - 1; l > 0; l--)
  {
```

```
    TrainableLayer32 *toLayer = (TrainableLayer32*)(m_network.getLayers
       ↪ ()[l].get());
    Layer32 *fromLayer = (toLayer->getPreviousLayerIndex() < l ?
                m_network.getLayers()[toLayer->getPreviousLayerIndex()].
                   ↪ get() :
                nullptr);


    if (toLayer)
      toLayer->Backpropagate(*fromLayer);
  }

  return retval;
}

void NeuralNetwork::Trainer32::PostBackpropagation(std::size_t
   ↪ uiTotalSamples)
{

  float fInvMinibatchSize, fInvTotalSamples;

  fInvMinibatchSize = 1.0f / m_network.getMinibatchSize();
  fInvTotalSamples = (uiTotalSamples == 0 ? 1.0f : 1.0f / uiTotalSamples
     ↪ );

  for (int l = 1;
    l < m_network.getLayerCount(); l++)
  {
    TrainableLayer32 *currentLayer = (TrainableLayer32*)(m_network.
       ↪ getLayers()[l].get());
    if (currentLayer)
      currentLayer->PostBackpropagate(fInvMinibatchSize,
         ↪ fInvTotalSamples);
  }
}

void NeuralNetwork::Trainer32::ResetLearningRate()
{
  #pragma omp parallel for
  for (int layer_i = 0; layer_i < (int)m_network.getLayerCount();
     ↪ layer_i++)
  {
    for (std::size_t neuron_i = 0; neuron_i < m_network.getLayers()[
       ↪ layer_i]->getNeuronCount(); neuron_i++)
      m_network.getLayers()[layer_i]->getNeuron(neuron_i).LearningRate =
         ↪  InitialLearningRate;
  }
}
```

B.29 File: ktypeutils.h

```
#ifndef K_TYPE_UTILITIES__H
#define K_TYPE_UTILITIES__H

#include <cstdint>
#include <algorithm>

#define REAL_32BITS

namespace NeuralNetwork
{
#ifdef REAL_32BITS
  #define real_type     float
#else
  #define real_type     double
#endif

   extern bool bUseGPU;
}

namespace KUtilities
{

   class BitUtilities
   {
   public:
     static bool _isLittleEndian()
     {
       int n = 1;
       return *(char*)&n == 1;
     }

     static void swap4(void *v)
     {
       char *in = (char*)v;
       char out[4];

       out[0] = in[3];
       out[1] = in[2];
       out[2] = in[1];
       out[3] = in[0];
       std::memcpy(v, out, 4);
     }

     static void swap8(void *v)
     {
       char *in = (char*)v;
       char out[8];
```

```
      out[0] = in[7];
      out[1] = in[6];
      out[2] = in[5];
      out[3] = in[4];
      out[4] = in[3];
      out[5] = in[2];
      out[6] = in[1];
      out[7] = in[0];
      std::memcpy(v, out, 8);
    }

    /// <summary>
    /// Flag that specifies whether the current system is little endian
        ↪ or big endian.
    /// </summary>
    static const bool IsLittleEndian;
};


class Int32
{
public:
  std::int32_t Value;

  Int32() : Int32(0)
  { }

  Int32(std::int32_t value) :
    Value(value)
  { }

  Int32(const Int32 &src) :
    Int32(src.Value)
  { }

  operator std::int32_t() const
  {
    return Value;
  }

  operator std::int32_t&()
  {
    return Value;
  }

  Int32& operator=(const Int32 &src)
  {
    if (&src != this)
      this->Value = src.Value;

    return *this;
```

```
    }

    Int32& operator=(std::int32_t value)
    {
      this->Value = value;

      return *this;
    }

    static std::size_t size()
    {
      return sizeof(std::int32_t);
    }

    static std::int32_t getLittleEndian(std::int32_t value)
    {
      std::int32_t result = value;
      if (!BitUtilities::IsLittleEndian)
      {
        BitUtilities::swap4(&result);
      }

      return result;
    }

    std::int32_t getLittleEndian() const
    {
      return getLittleEndian(Value);
    }

    static std::int32_t getBigEndian(std::int32_t value)
    {
      std::int32_t result = value;
      if (BitUtilities::IsLittleEndian)
      {
        BitUtilities::swap4(&result);
      }

      return result;
    }

    std::int32_t getBigEndian() const
    {
      return getBigEndian();
    }

    static std::int32_t ToEndian(std::int32_t value, bool
      ↪ bIsLittleEndian)
    {
      if (BitUtilities::IsLittleEndian != bIsLittleEndian)
```

```
        std::reverse((char*)&value, (char*)&value + sizeof(std::int32_t)
            ↪ );

      return value;
    }

    static std::int32_t ToEndian(const char *buffer, bool
        ↪ bIsLittleEndian)
    {
      std::int32_t value = *(const std::int32_t*)buffer;

      return ToEndian(value, bIsLittleEndian);
    }

    void set(std::int32_t value, bool bIsLittleEndian)
    {
      Value = ToEndian(value, bIsLittleEndian);
    }
};

class UInt32
{
public:
  std::uint32_t Value;

  UInt32() : UInt32(0)
  { }

  UInt32(std::uint32_t value) :
    Value(value)
  { }

  UInt32(const UInt32 &src) :
    UInt32(src.Value)
  { }

  operator std::int32_t() const
  {
    return Value;
  }

  operator std::uint32_t&()
  {
    return Value;
  }

  UInt32& operator=(const UInt32 &src)
  {
    if (&src != this)
      this->Value = src.Value;
```

```
    return *this;
  }

  UInt32& operator=(std::uint32_t value)
  {
    this->Value = value;

    return *this;
  }

  static std::size_t size()
  {
    return sizeof(std::uint32_t);
  }

  static std::uint32_t getLittleEndian(std::uint32_t value)
  {
    std::uint32_t result = value;
    if (!BitUtilities::IsLittleEndian)
    {
      BitUtilities::swap4(&result);
    }

    return result;
  }

  void getLittleEndian(std::uint32_t &result) const
  {
    result = getLittleEndian(Value);
  }

  std::uint32_t getLittleEndian() const
  {
    return getLittleEndian(Value);
  }

  static std::uint32_t getBigEndian(std::uint32_t value)
  {
    std::uint32_t result = value;
    if (BitUtilities::IsLittleEndian)
    {
      BitUtilities::swap4(&result);
    }

    return result;
  }

  void getBigEndian(std::uint32_t &result) const
  {
    result = getBigEndian(Value);
  }
```

```
  std::uint32_t getBigEndian() const
  {
    return getBigEndian(Value);
  }

  static std::uint32_t ToEndian(std::uint32_t value, bool
    ↪ bIsLittleEndian)
  {
    if (BitUtilities::IsLittleEndian != bIsLittleEndian)
      std::reverse((char*)&value, (char*)&value + sizeof(std::uint32_t
        ↪ ));

    return value;
  }

  static std::uint32_t ToEndian(const char *buffer, bool
    ↪ bIsLittleEndian)
  {
    std::uint32_t value = *(const std::uint32_t*)buffer;

    return ToEndian(value, bIsLittleEndian);
  }

  void set(std::uint32_t value, bool bIsLittleEndian)
  {
    Value = ToEndian(value, bIsLittleEndian);
  }
};

class UInt64
{
public:
  std::uint64_t Value;

  UInt64() : UInt64(0)
  { }

  UInt64(std::uint64_t value) :
    Value(value)
  { }

  UInt64(const UInt64 &src) :
    UInt64(src.Value)
  { }

  operator std::int64_t() const
  {
    return Value;
  }
```

```
operator std::uint64_t&()
{
  return Value;
}

UInt64& operator=(const UInt64 &src)
{
  if (&src != this)
    this->Value = src.Value;

  return *this;
}

UInt64& operator=(std::uint64_t value)
{
  this->Value = value;

  return *this;
}

static std::size_t size()
{
  return sizeof(std::uint64_t);
}

static std::uint64_t getLittleEndian(std::uint64_t value)
{
  std::uint64_t result = value;
  if (!BitUtilities::IsLittleEndian)
  {
    BitUtilities::swap8(&result);
  }

  return result;
}

void getLittleEndian(std::uint64_t &result) const
{
  result = getLittleEndian(Value);
}

std::uint64_t getLittleEndian() const
{
  return getLittleEndian(Value);
}

static std::uint64_t getBigEndian(std::uint64_t value)
{
  std::uint64_t result = value;
  if (BitUtilities::IsLittleEndian)
  {
```

```
        BitUtilities::swap8(&result);
      }

      return result;
    }

    void getBigEndian(std::uint64_t &result) const
    {
      result = getBigEndian(Value);
    }

    std::uint64_t getBigEndian() const
    {
      return getBigEndian(Value);
    }

    static std::uint64_t ToEndian(std::uint64_t value, bool
       ↪ bIsLittleEndian)
    {
      if (BitUtilities::IsLittleEndian != bIsLittleEndian)
        std::reverse((char*)&value, (char*)&value + sizeof(std::uint64_t
           ↪ ));

      return value;
    }

    static std::uint64_t ToEndian(const char *buffer, bool
       ↪ bIsLittleEndian)
    {
      std::uint64_t value = *(std::uint64_t*)buffer;

      return ToEndian(value, bIsLittleEndian);
    }

    void set(std::uint64_t value, bool bIsLittleEndian)
    {
      Value = ToEndian(value, bIsLittleEndian);
    }
  };

}
#endif
```

B.30 File: ktypeutils.cpp

```
#include "ktypeutils.h"

bool NeuralNetwork::bUseGPU = true;
```

```
const bool KUtilities::BitUtilities::IsLittleEndian = BitUtilities::
    ↪ _isLittleEndian();
```

Bibliography

[1] The Mixed National Institute of Standards and Technology (MNIST) Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/. [Online; accessed 2016].

[2] R. Battiti. First and second order methods for learning: Between steepest descent and newton's method. *Neural Computation*, 4(2):141166, 1992.

[3] E. M. L. Beale. A derivation of conjugate gradients. In F. A. Lootsma, editor, *Numerical methods for nonlinear optimization*, pages 39–43. Academic Press, London, UK, 1972.

[4] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *arXiv:1206.5533v2 [cs.LG]*, 2012.

[5] K. W. Boer. Solar Cells. http://www.chemistryexplained.com/Ru-Sp/Solar-Cells.html, 2015. [Online; accessed 2015].

[6] S. M. Botros and C. G. Atkeson. Generalization properties of radial basis functions. In *1990 conference on Advances in neural information processing systems*, pages 707–713, Denver, CO, 1990.

[7] W. L. Buntine and A. S. Weigend. Computing second derivatives in feed-forward networks: A review. *IEEE Transactions on Neural Networks*, 5(3):480–488, 1993.

[8] E. Castillo et al. A very fast learning method for neural networks based on sensitivity analysis. *Journal of Machine Learning Research*, 7:1159–1182, 2006.

[9] P. Chandra, U. Ghose, and A. Sood. A Non-Sigmoidal Activation Function for Feedforward Artificial Neural Networks. In *International Joint Conference on Neural Networks*, Killarney, Ireland, July 2015.

[10] S. D. Chen and R. Ramli. Contrast Enhancement using Recursive Mean-Separate Histogram Equalization for Scalable Brightness Preservation. *IEEE Transactions on Consumer miscs*, 49(4):1301–1309, 2003.

[11] V. CL. ViennaCL - The Vienna Computing Library 1.7.1. http://viennacl.sourceforge.net/doc/. [Online; accessed 2016].

[12] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In *International Conference on Learning Representations*, San Juan, Puerto Rico, May 2016.

[13] C. A. R. de Sousa. An overview on weight initialization methods for feedforward neural networks. In *International Joint Conference on Neural Networks (IJCNN)*, pages 52–59, Vancouver, Canada, July 2016.

[14] J. Dean et al. Large scale distributed deep networks. In *Neural Information Processing Systems*, Lake Tahoe, NV, 2012.

[15] S. Devi et al. Initializing artificial neural networks by genetic algorithm to calculate the resonant frequency of single shorting post rectangular patch antenna. In *Antennas and Propagation Society International Symposium, IEEE*, pages 144–147, June 2003.

[16] G. P. Drago and S. Ridella. Statistically controlled activation weight initialization (scawi). *IEEE Transactions on Neural Networks*, 3:627–631, 1992.

[17] C. Dugas et al. Incorporating Second-Order Functional Knowledge for Better Option Pricing. In *Advances in Neural Information Processing Systems 13*, 2001.

[18] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning Hierarchical Features for Scene Labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2013.

[19] D. Fay et al. Real-time Image Fusion and Target Learning and Detection on a Laptop Attached Processor. In *8th International Conference on Information Fusion*, pages 499–506, 2005.

[20] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS'2010*, pages 249–256, Sardinia, Italy, May 2010.

[21] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proc. 14th International Conference on Artificial Intelligence and Statistics*, volume 5, pages 315–323, 2011.

[22] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Pearson Prentice Hall, Upper Saddle River, New Jersey, third edition, 2008.

[23] K. Group. OpenCL 2.0 Reference. https://www.khronos.org/registry/OpenCL/sdk/2.0/docs/man/xhtml/. [Online; accessed 2016].

[24] M. T. Hagan and M. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.

[25] K. He et al. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[26] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv:1502.01852 [cs]*, 2015.

[27] G. E. Hinton et al. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580v1 [cs.NE]*, July 2012.

[28] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[29] K. Hornik. Approximation capabilities of multilayer feed forward networks. *Neural Networks*, 4(2):251–257, 1991.

[30] Intel. Intel Math Kernel Library Reference - C. https://software.intel.com/en-us/mkl-developer-reference-c, 2017. [Online; accessed 2015-2017].

[31] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167v3*, 2015.

[32] ISO/IEC14882:2011. Programming Language C++. Standard, International Organization for Standardization, Sept. 2011.

[33] A. Karpathy. Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/optimization-1/. [Online; accessed 2016].

[34] Y. T. Kim. Contrast Enhancement Using Brightness Preserving Bi-Histogram Equalization. *IEEE Transactions on Consumer miscs*, 43(1):1–8, 1997.

[35] J. F. Kolen and S. C. Kremer. Gradient flow in recurrent nets: The difficulty of learning long term dependencies. In *A Field Guide to Dynamical Recurrent Networks*, pages 237–243. Wiley-IEEE Press, 2001.

[36] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[37] K. Kuresson. Sparse matrix algorithms using gpu, 2012.

[38] H. Lari-Najafi, M. Nasiruddin, and T. Samad. Effect of initial weights on back-propagation and its variations. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, pages 218–219, 1989.

[39] Y. LeCun. A learning procedure for an asymmetric threshold network. In *Cognitiva 85*, pages 559–604, Paris, France, 1985.

[40] Y. LeCun. A learning procedure for an asymmetric threshold network. *Cognitiva 85*, 323:533–536, 1986.

[41] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Muller. *Neural Networks: tricks of the trade*. Springer, New York City, NY, 1998.

[42] Y. LeCun et al. Gradient-based learning applied to document recognition. In *Proc. of IEEE*, pages 1–46, Nov. 1998.

[43] Y. LeCun, I. Kanter, and S. A. Solla. Second order properties of error surfaces: Learning time and generalization. In R. Lippmann, J. Moody, and D. S. Touretzky, editors, *Neural Information Processing Systems*, pages 918–924. Morgan Kaufmann, San Mateo, CA, 1991.

[44] Y. Lee, S. H. Oh, and M. W. Kim. The effect of initial weights on premature saturation in back-propagation learning. In *Procs. of the International Joint Conference on Neural Networks IEEE*, volume 1, pages 765–770, Seatle, WA, 1991.

[45] M. Leshno et al. Multilayer feedforward networks with a non-polynomial activation function can approximate any function. *Neural Networks*, 6:861–867, 1993.

[46] A. L. Maas, A. Y. Hannun, , and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. 30th International Conference on Machine Learning*, 2013.

[47] R. Maitra. Discrimination and Classification - Introduction. http://www.public.iastate.edu/ maitra/stat501/lectures/Classification-I.pdf, 2012. [Online; accessed 2015].

[48] D. W. Marquardt. An algorithm for least-squares estimation of non-linear parameters. *Journal of the Society of Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[49] M. K. Mazumder et al. Solar Panel Obscuration by Dust and its Mitigation in the Martian Atmosphere. *Particles on Surfaces 9: Detection, Adhesion and Removal*, pages 1–29, 2006.

[50] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[51] A. J. McEvoy, T. Markvart, and L. Castaer. *Principles of Solar Cell Operation*, pages 7–30. Academic Press, Elsevier, 2 edition, 2012.

[52] G. J. Mclachlan. Mahalanobis distance. *Resonance*, 4(6):20–26, 1999.

[53] D. Montana. Neural network weight selection using genetic algorithms. In S. Goonatilake and S. Khebbal, editors, *Intelligent Hybrid Systems*. Wiley, 1995.

[54] A. Ng et al. Unsupervised feature learning and deep learning tutorial. http://ufldl.stanford.edu, 2015. [Online; accessed 2016].

[55] D. Nguyen and B. Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *1990 IJCNN International Joint Conference on Neural Networks*, June 1990.

[56] M. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com, 2016. [Online; accessed 2016].

[57] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3(2):246–257, 1991.

[58] D. B. Parker. Optimal algorithms for adaptive networks: second order back propagation, second order direct propagation, and second order hebbian learning. In *Proceedings of the IEEE Conference on Neural Networks*, page 593600, 1987.

[59] R. Reed and R. Marks. *Neural Smithing*. MIT Press, Cambridge, MA, 1999.

[60] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[61] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel Distributed Processing*, 1(8), 1986.

[62] C. P. Ryan, F. Vignola, and D. K. McDaniels. Solar cell arrays: Degradation due to dirt. *Proceedings of 1989 Annual Conference of The American Solar Energy Society*, pages 234–237, 1989.

[63] N. Sapkota. Real time digital night vision using nonlinear contrast enhancement. Master's thesis, University of Nevada, Las Vegas. Las Vegas, NV, 2003.

[64] F. Schule, R. Schweiger, and K. Dietmayer. Augmenting Night Vision Video Images with Longer Distance Road Course Information. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 1233–1238, 2013.

[65] M. Serfling, O. Loehlein, R. Schweiger, and K. Dietmayer. Camera and Imaging Radar Feature Level Sensorfusion for Night Vision Pedestrian Recognition. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 597–603, 2009.

[66] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[67] S. S. Sodhi, P. Chandra, and S. Tanwar. A new weight initialization method for sigmoidal feedforward artificial neural networks. In *International Joint Conference on Neural Networks (IJCNN)*, pages 291–298, Beijing, China, July 2014.

[68] C. Szegedy et al. Going deeper with convolutions. *arXiv:1409.4842v1*, 2014.

[69] C. K. Teo. Digital Enhancement of Night Vision and Thermal Images. Master's thesis, Naval Postgraduate School. Monterey, CA, 2003.

[70] L. M. Waghmare, N. N. Bidwai, and P. P. Bhogle. Neural network weight initialization. In *Proc. IEEE International Conference on Mechatronics and Automation*, pages 679–681, Harbin, China, Aug. 2007.

[71] R. E. Walpole et al. *Probability & Statistics for Engineers & Scientists*. Pearson Prentice Hall, 8th edition, 2007.

[72] A. Waxman et al. Sensor Fused Night Vision: Assessing Image Quality in the Lab and in the Field. In *9th International Conference on Information Fusion*, pages 1–8, 2006.

[73] E. W. Weisstein. Central Limit Theorem. http://mathworld.wolfram.com/CentralLimitTheorem.html. [Online; accessed 2015].

[74] E. W. Weisstein. Convolution. http://mathworld.wolfram.com/Convolution.html. [Online; accessed 2016].

[75] E. A. Yfantis. Dynamic Redundancy Bit Allocation and Packet Size to Increase Throughput in Noisy Real Time Video Wireless Transmission. *The Journal of Combinatorial Mathematics and Combinatorial Computing JCMCC*, 86:163–169, 2013.

[76] E. A. Yfantis. Telemedicine: The present and its future. In *Plenary Lecture: 14th International Conference in Applied Computer Science*, Cambridge, MA, Jan. 2014.

[77] E. A. Yfantis et al. Pollution Detection in Urban Areas Using the Existing Camera Networks. *International Journal of Multimedia Technology*, 3(3):98–102, 2013.

[78] E. A. Yfantis and A. Fayed. A Camera System for Detecting Dust and Other Deposits on Solar Panels. *Advances in Image and Video Processing*, 2(5), 2014.

[79] E. A. Yfantis and A. Fayed. Authentication and Secure Robot Communication. *International Journal of Advanced Robotic Systems*, 11:1–6, 2014.

[80] E. A. Yfantis and A. Fayed. Robot Vision and Distance Estimation. In *14th International Conference in Applied Computer Science*, Cambridge, MA, Jan. 2014.

[81] E. A. Yfantis, A. Fayed, E. Zamora Ramos, and A. Amriphale. Pollution Detection in Urban Areas Using the Existing Camera Networks. *International Journal of Multimedia Technology*, 3(3):98102, 2013.

[82] E. A. Yfantis, M. Nakakuni, and E. Zamora Ramos. Low-Bandwidth Transmission Algorithm for Reliable Wireless Communication. In *IEEE 7th Annual Computing and Communication Workshop and Conference*, Las Vegas, Nevada, Jan. 2017.

[83] Z. Yu, W. Xiqin, and P. Yingning. New Image Enhancement Algorithm for Night Vision. In *International Conference on Image Processing. ICIP '99*, volume 1, pages 201–203, 1999.

[84] S. Yue and M. Hashino. Probability distribution of annual, seasonal and monthly precipitation in Japan. *Hydrological Sciences Journal*, 52(5):863–877, 2010.

[85] E. Zamora Ramos. Using Image Processing Techniques to Estimate the Air Quality. *Journal of McNair Scholars Institute*, pages 189–194, 2011.

[86] E. Zamora Ramos. Intensity Weighted Histogram Equalization Method for Night Vision. *Advances in Image and Video Processing*, 3(4), 2015.

[87] E. Zamora Ramos, S. Ho, and E. A. Yfantis. Using spectral decomposition to detect dirty solar panels and minimize impact on energy production. *Advances in Image and Video Processing*, 3(6), 2015.

[88] E. Zamora Ramos, M. Nakakuni, and E. A. Yfantis. Quantitative Measures to Evaluate Neural Network Weight Initialization Strategies. In *IEEE 7th Annual Computing and Communication Workshop and Conference*, Las Vegas, Nevada, Jan. 2017.

[89] E. Zamora Ramos, M. Ramos, K. Moutafis, and E. A. Yfantis. A Robot Architecture for Detecting Dust and Cleaning Solar Panels. In *31st International Conference on Computers and Their Applications CATA'16*, Las Vegas, NV, 2016.

[90] E. Zamora Ramos, M. Ramos, K. Moutafis, and E. A. Yfantis. Robot-Server Architecture for Optimizing Solar Panel Power Output. *Transactions on Machine Learning and Artificial Intelligence*, 4(4), 2016.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Ernesto Zamora Ramos
zamorara@unlv.nevada.edu

Degrees:

Bachelor of Science in Computer Science 2013
University of Nevada Las Vegas

Dissertation Title: Improving Pattern Recognition and Neural Network Algorithms With Applications to Solar Panel Optimization

Dissertation Examination Committee:

Chairperson, Dr. Evangelos A. Yfantis, Ph.D.
Committee Member, Dr. Hal Berghel, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Committee Member, Dr. Andreas Stefik, Ph.D.
Graduate Faculty Representative, Dr. Sarah Harris, Ph.D.

Professional Affiliations:

Institute of Electrical and Electronics Engineers (IEEE) – Member
Association for Computing Machinery (ACM) – Member

Publications:

Zamora Ramos, E., Nakakuni, M. and Yfantis E. A., "Quantitative Measures to Evaluate Neural Network Weight Initialization Strategies." IEEE 7th Annual Computing and Communication Workshop and Conference, Las Vegas, Nevada, Jan. 2017.

Yfantis E. A., Nakakuni, M. and Zamora Ramos, E., "Low-Bandwidth Transmission Algorithm for Reliable Wireless Communication." IEEE 7th Annual Computing and Communication Workshop and Conference, Las Vegas, Nevada, Jan. 2017.

Zamora Ramos, E. et al. "Robot-Server Architecture for Optimizing Solar Panel Power Output." Transactions on Machine Learning and Artificial Intelligence, Vol. 4 No. 4, 2016, pp. 9-17.

Zamora Ramos, E. et al. "A Robot Architecture for Detecting Dust and Cleaning Solar Panels." 31st International Conference on Computers and Their Applications, CATA 16, Las Vegas, Nevada, 2016.

Zamora Ramos, E., Ho, S., Yfantis E. A., "Using Spectral Decomposition to Detect Dirty Solar Panels and Minimize Impact on Energy Production," Journal Of Advances in Image and Video Processing, Vol. 3 No. 6, 2015, pp. 1-12.

Zamora Ramos, E., "Intensity Weighted Histogram Equalization Method for Night Vision," Journal Of Advances in Image and Video Processing, Vol. 3 No. 3, 2015, pp. 18-27.

Zamora Ramos, E. and Yfantis E. A., "Multilevel Encryption with Steganography and Lossless Wavelets," Midwest Conference on Combinatorics and Combinatorial Computing, 2014.

Yfantis, E. A., Fayed, A., Zamora, E. and Amriphale A., "Pollution detection in urban areas using the existing camera networks," International Journal of Multimedia Technology, Vol. 3 No. 3, 2013, pp. 98-102.

Zamora Ramos, E., "Using Image Processing Techniques to Estimate the Air Quality," McNair Scholars Research Journal, UNLV chapter, 6th ed., 2012, pp 189-194.