University of South Carolina
Scholar Commons

Theses and Dissertations

2017

Deep Learning: An Exposition

Ryan Kingery
University of South Carolina

Follow this and additional works at: https://scholarcommons.sc.edu/etd

 Part of the Mathematics Commons

Recommended Citation
Kingery, R.(2017). *Deep Learning: An Exposition.* (Master's thesis). Retrieved from https://scholarcommons.sc.edu/etd/4348

This Open Access Thesis is brought to you by Scholar Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact dillarda@mailbox.sc.edu.

www.manaraa.com

Deep Learning: An Exposition

by

Ryan Kingery

Bachelor of Science
Clemson University 2013

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science in

Mathematics

College of Arts and Sciences

University of South Carolina

2017

Accepted by:

Edsel Peña, Director of Thesis

Stephen Dilworth, Reader

Cheryl L. Addy, Vice Provost and Dean of the Graduate School

المنارة للاستشارات

www.manaraa.com

© Copyright by Ryan Kingery, 2017
All Rights Reserved.

www.manaraa.com

Abstract

In this paper we describe and survey the field of deep learning, a type of machine learning that has seen tremendous growth and popularity over the past decade for its ability to substantially outperform other learning methods at important tasks. We focus on the problem of supervised learning with feedforward neural networks. After describing what these are we give an overview of the essential algorithms of deep learning, backpropagation and stochastic gradient descent. We then survey some of the issues that occur when applying deep learning in practice. Last, we conclude with an important application of deep learning to the problem of handwriting recognition.

www.manaraa.com

Table of Contents

www.manaraa.com

List of Figures

www.manaraa.com

Chapter 1

Preliminaries

1.1 Introduction

Machine learning is often defined as the field of study that gives computers the ability to learn without being explicitly programmed [7]. While this informal definition suggests that machine learning is a subfield of artificial intelligence, it has found applications in many other fields over the past few decades, e.g. finance, medicine, particle physics, linguistics, and neuroscience. In fact, machine learning is one of the main drivers of the so called big data revolution currently ongoing.

The basic premise of machine learning is the use of a set of observations to uncover an underlying process [1]. One can abstractly view this process as a form of function (or more generally probability distribution) estimation: Given some initial data, find a function that best describes the data while still predicting the values of new, unseen data really well. In this setting, one assumes there is a set of input *features* that can be used to predict the desired output with reasonable accuracy. Depending on how the initial data is specified there are several machine learning paradigms in current use, the most important of which are the following:

- **Supervised Learning:** Both inputs and outputs are given.
- **Unsupervised Learning:** Only inputs are given.
- **Reinforcement Learning:** Inputs are given with only a few graded outputs.

In this paper we will be focused on supervised learning, which is the most commonly

www.manaraa.com

used form of learning in practice. Supervised learning can be further subdivided into regression and classification problems depending on whether the outputs allowed are discrete (classification) or continuous (regression).

There are many supervised learning methods available. Examples include linear regression, ANOVA, logistic regression, nearest neighbors, naive Bayes, support vector machines (SVMs), and neural networks. Neural networks turn out to be the foundation of deep learning. Neural networks are a learning method based loosely on how the human brain is believed to operate. They were first developed in the early 1950s by Frank Rosenblatt, the founder of artificial intelligence, and as such were one of the earliest learning methods used in machine learning [7].

As the name suggests, a neural network is a network of objects that are roughly modeled on the biological neuron (Figure 1.1). A neuron functions by accepting a series of electrical impulses and deciding whether or not to fire those impulses along to other neurons. This process can roughly be modeled as follows: accept a sequence of inputs, weigh and sum those inputs and add a threshold value, and pass this sum through an activation function that determines whether the neuron will fire. It is this model that forms the basis of the artificial neuron used to define a neural network.

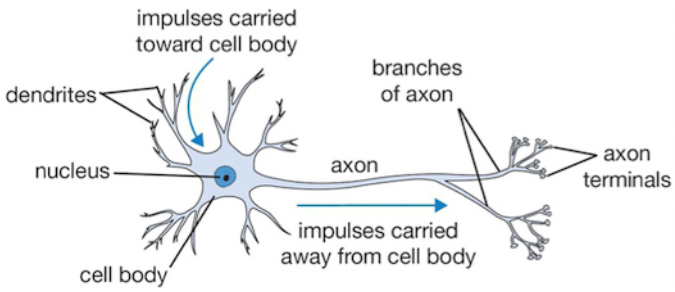

Figure 1.1 Illustration of a biological neuron [9].

Deep learning is a type of neural network learning that has gained substantial popularity over the past decade for its ability to learn difficult tasks in computer

www.manaraa.com

vision, speech, natural language processing, video games, particle physics, and other fields [11]. Deep learning is based loosely on the idea that representations of data should be hierarchical. Deep learning algorithms should learn multiple levels of representations corresponding to different levels of abstraction, forming a hierarchy of concepts. Moreover, such algorithms should consist of cascades of many layers of nonlinear processing, both for feature extraction and transformation. In the framework of neural networks deep learning typically refers to networks with many hidden layers, where each successive layer operates on higher level features [7].

Though deep learning can be used for many types of learning and can involve various types of neural networks, we will focus on the most widely used version that involves supervised learning with feedforward neural networks (or multilayer perceptrons). After introducing these ideas we will examine the workhorse algorithms of deep supervised learning, backpropagation and stochastic gradient descent. Once these have been examined we will look at some, but by no means all, of the issues that can arise from applying deep learning in real life.

1.2 Supervised Learning

The problem of supervised learning can be formulated abstractly in the following way: Let $f : X \to Y$ be a function from a *feature space* X to a *target space* Y, let H be a *hypothesis class* of functions $h : X \to Y$, and let

$$D = \{(x_1, y_1), \ldots, (x_n, y_n)\} \subset X \times Y$$

be the *training data*. The goal is to use a *learning algorithm* A to find a function $g \in H$ that best approximates f on X in some pre-defined sense, given D.

Suppose the hypothesis class H is parametrized by θ, i.e. $H = \{h_\theta : \theta \in \Theta\}$. The most common way to find $g \in H$ is by first defining a *loss function* $L = L(y, h_\theta(x))$. The goal is then to find the value $\theta^* \in \Theta$ that minimizes the *risk* $R(\theta) \equiv E(L)$. As

www.manaraa.com

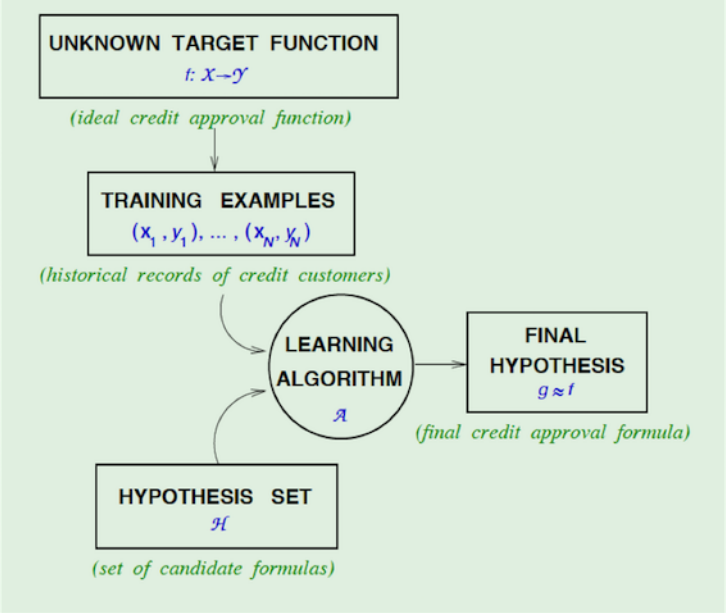

Figure 1.2 A schematic of the supervised learning process [1].

only D is given, however, we instead must minimize an estimator of the risk functional and establish conditions under which minimizing this estimator will asymptotically result in minimizing the risk as well. The estimator used is the *empirical risk*

$$R_{emp}(\theta) \equiv \frac{1}{n}\sum_{i=1}^{n} L(y_i, h_\theta(x_i)),$$

The process described is known as the empirical risk minimization (ERM) principle [14].

That one can find θ^*, and hence g, by minimizing the empirical risk follows from the following theorem from statistical learning theory. First, we say that ERM is *consistent* provided $R_{emp}(\theta_n)$ and $R(\theta_n)$ each converge in probability to the minimum risk $R(\theta^*)$ for some sequence of parameters $\theta_n \in \Theta$.

Theorem 1.1. *Suppose D is iid with joint distribution $P(x, y)$. Let $\{L(y, h_\theta(x))\}_{\theta\in\Theta}$ be a collection of loss functions whose expectations with respect to $P(x, y)$ are uniformly bounded. Then the ERM principle is consistent if and only if the empirical*

www.manaraa.com

risk $R_{emp}(\theta)$ converges uniformly to the actual risk $R(\theta)$ in the following sense: For any $\varepsilon > 0$,

$$\lim_{n\to\infty} P\{\sup_\theta(R(\theta) - R_{emp}(\theta)) > \varepsilon\} = 0.$$

That is, ERM is consistent if and only if R_{emp} converges uniformly in probability to R in the one-sided sense. For a proof of this result see [15]. Under stronger conditions we can also say something about the rate of this convergence. Define the *growth function* G by $G(n) \equiv \log \sup_D N_D$, where N_D is the number of different ways to separate D using a set of indicator functions [14].

Theorem 1.2. *If $\lim_{n\to\infty} \frac{1}{n}G(n) = 0$, then the ERM is consistent and convergence is fast in the following sense: For any $\varepsilon > 0$, there is some constant $c > 0$ such that*

$$P\{R(\theta_n) - R(\theta^*) > \varepsilon\} < e^{-cn\varepsilon^2}.$$

When convergence takes place the supervised learning problem is said to *generalize*, and R is referred to as the *generalization error* while R_{emp} is referred to as the *training error*.

Once the loss function and hypothesis class have been specified and ERM has been applied, the learning algorithm simply becomes the optimization problem of finding θ^*. Thus, learning algorithms become optimization algorithms. A popular class of optimization algorithms used in machine learning are based on gradient descent.

Gradient descent is a simple algorithm used to find a (local) minimum for L. First, a value θ_0 is initialized, often randomly, and a *learning rate* $\eta > 0$ is specified. Assuming the gradient $\frac{\partial L}{\partial \theta}$ has been found, the algorithm then updates θ via

$$\theta_{n+1} = \theta_n - \eta \frac{\partial L}{\partial \theta_n}.$$

Provided η is sufficiently small, θ_n will converge to a local minimum θ^* [8]. Note which minimum the algorithm converges to depends on the initialization. If L is convex this won't be a problem, but convexity rarely occurs in deep learning.

www.manaraa.com

Since the ultimate choice of g depends on the loss function, one must specify which loss function is being used before applying the learning algorithm. Such a choice in practice depends mainly on the hypothesis class chosen. Below are some examples of common learning models. Note that neural networks will be addressed in the next chapter.

- **Linear Regression:** The hypothesis class consists of functions of the form $h_\theta(x) = \theta \cdot x = \sum_{j=0}^{k} \theta_k x_k$ where $\theta, x \in \mathbb{R}^{k+1}$ and $x_0 \equiv 1$. The typical loss function is the *mean square loss* $L = \frac{1}{2}(y - \theta \cdot x)^2$. Note that in this case θ^* can be found exactly via the normal equations.

- **Logistic Regression:** Here the hypothesis class consists of functions of the form $h_\theta(x) = \frac{1}{1+e^{\theta \cdot x}}$ where again $\theta, x \in \mathbb{R}^k$, and $y \in \{0, 1\}$. A typical loss function is the *log loss* $L = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$.

- **Nearest Neighbors:** The hypothesis class in this case consists of functions of the form $h_\theta(x) = y_m$, where $(x_m, y_m) \in D$ and x_m is the point closest to x with respect to the metric defined on X. The loss function is the *0-1 loss* $L = I(h_\theta(x) \neq y)$. Note in this case $R(\theta) = P\{h_\theta(x) \neq y\}$, i.e. the expected loss is just the probability of classification error.

We conclude this chapter by noting that the size of the target space Y determines what type of problem is being addressed. Specifically, if Y is discrete the supervised learning problem is a *classification* problem. If Y is continuous it is a *regression* problem. If Y is neither discrete nor continuous the problem is said to be a *mixture* problem. The type of deep learning we will study focuses primarily on the classification problem, which is the most commonly occurring situation in applied machine learning settings.

www.manaraa.com

Chapter 2

Deep Learning

2.1 Feedforward Neural Networks

We now give a formal introduction to the feedforward neural network, the simplest and most widely used form of deep learning. Here are a few essential definitions to get started.

An *activation function* is any function $\sigma : \mathbb{R} \to \mathbb{R}$ that is non-constant, bounded, increasing, and continuous everywhere (except possibly a finite number of points). Abusing notation slightly, we will also use σ to denote the vectorized activation function, i.e. $\sigma(x)_i \equiv \sigma(x_i)$ for all $x_i \in \mathbb{R}$.

A *neuron* is a collection (X, Y, s, σ) where $X \subset \mathbb{R}^m$ is the input space, $Y \subset \mathbb{R}$ is the output space, $s : X \to \mathbb{R}$ is an input map, and $\sigma : \mathbb{R} \to Y$ is an activation function.

Most commonly, we take $Y = \{0, 1\}$ and $s(x) \equiv w \cdot x + b$ where $w \in \mathbb{R}^m$ is a vector of input *weights*, and $b \in R$ is the *bias*. In this situation, a neuron is represented by a function $\sigma(w \cdot x + b)$, where σ is an activation function. We will assume from here on that our neurons are of this form.

Some examples of common activation functions are the following:

- Perceptron: $\sigma(z) = \text{sgn}(z)$
- Sigmoid: $\sigma(z) = \frac{1}{1+e^z}$
- Hyperbolic tangent: $\sigma(z) = \tanh(z)$

www.manaraa.com

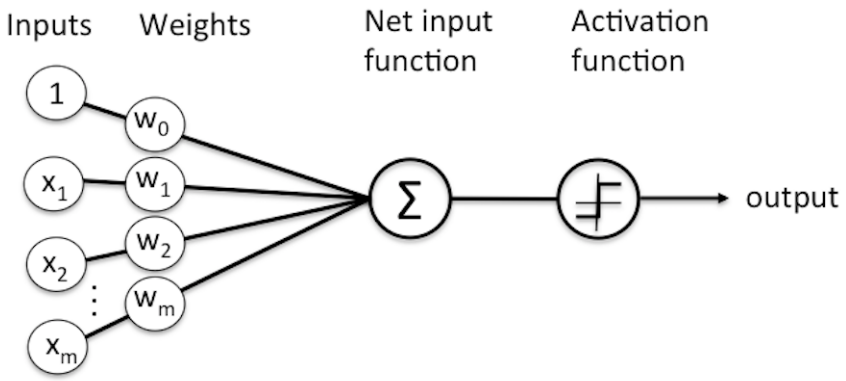

Figure 2.1 A neuron receiving inputs $x_1, \cdots, x_m$, which are weighted and summed with weights $w_1, \cdots, w_m$ and added to a bias w_0 before being passed through a nonlinear activation function to produce an output [4].

- Rectified linear unit (ReLU): $\sigma(z) = \max\{0, z\}$

A *feedforward neural network* is a directed, acyclic graph where each node is a neuron and each edge is labeled the neuron's inputs and weights. For simplicity we will assume each neuron has the same activation function, though this need not be the case in practice. The initial layer is called the *input layer*, the terminal layer is called the *output layer*, and the remaining internal layers are called *hidden layers* (Figure 2.2). Note there are other types of neural networks, e.g. recurrent neural networks, that have many important applications as well, but those won't be discussed here.

Let us first consider the simplest case of a feedforward neural network with no hidden layers. When $\sigma(z) = \text{sgn}(z)$ we get the original perceptron model. Such a model is only able to classify data with a linear decision boundary, which considerably weakens the versatility of the model. In particular, it is well known that the perceptron model cannot correctly classify the XOR function [7].

The other common activation functions mentioned above lead to the same problem. Even though these are each valid neural networks, they are quite weak since

www.manaraa.com

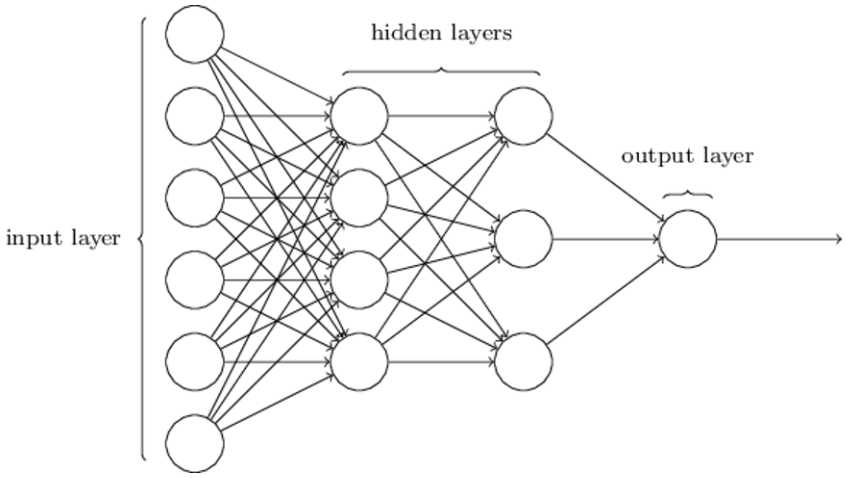

Figure 2.2 A feedforward neural network with two hidden layers. Each node represents a neuron receiving inputs from the previous layer [13].

they are unable to take advantage of the hierarchical structure of nonlinear transformations afforded by a neural network model. Note that we can allow these networks to learn nonlinear decision boundaries by using kernels, but in general such models still won't be as powerful as neural networks with hidden layers.

Next, consider the case of a feedforward neural network with exactly one hidden layer. Such neural networks are incredibly versatile, and as such are the most commonly used in practice. In fact, these networks can learn pretty much *any* decision boundary if given enough neurons. This result, known as the Universal Approximation Theorem, is stated formally below. For a proof, see [6].

Theorem 2.1. *Let σ be a vectorized activation function, $w^{(1)} \in \mathbb{R}^{m\times n}$, $w^{(2)} \in \mathbb{R}^{1\times m}$, $b^{(1)} \in \mathbb{R}^m$, $b^{(2)} \in \mathbb{R}$, and $x \in \mathbb{R}^n$. Let H be a hypothesis class of functions of the form $h(x) = w^{(2)}\sigma(w^{(1)}x + b^{(1)}) + b^{(2)}$. Let $f : [a, b]^n \to \mathbb{R}$ be continuous. Then for any $\varepsilon > 0$, there exists $g \in H$ such that*

$$\max_{a\le x\le b} |f(x) - g(x)| < \varepsilon.$$

www.manaraa.com

That is, H is dense in $C[a,b]^n$, the space of continuous functions $h : [a,b]^n \to R$.

Corollary 2.2. *Under the conditions above, H is dense in $L^p(\mathbb{R}^n)$ for any $1 \leq p < \infty$.*

Proof. Observe the above theorem trivially implies that H is dense in the space $C_c(\mathbb{R}^n)$ of compactly supported functions on $\mathbb{R}^n$. Since $C_c(\mathbb{R}^n)$ is known to be dense in $L^p(\mathbb{R}^n)$ the result easily follows. □

Note that the Universal Approximation Theorem does not specify how many neurons are needed to ensure $\max |f(x) - g(x)| < \varepsilon$ for each given ε. Also note that if neural networks with one hidden layer can learn any reasonable function, so can neural networks with more hidden layers since H only becomes larger.

2.2 Neural Network Learning

Neural network learning is performed much the same way as other supervised learning methods. A loss function is first specified for the network, which is then minimized with respect to *both* the weights and biases. The principle difference in between neural networks and simpler supervised learning methods is that the structure and size of neural networks makes it necessary to have special algorithms to compute the gradients of the loss function and then to minimize the loss function. The standard algorithms for doing these two things are, respectively, backpropagation and stochastic gradient descent.

2.2.1 Backpropagation

Backpropagation is an algorithm designed to efficiently compute the gradient of the loss function on a neural network. It is conceptually simple, computationally efficient, and almost always works [10]. Conceptually, backpropagation is based on the chain rule from multivariate calculus.

www.manaraa.com

Suppose an N layer feed-forward neural network has weight matrices $w^{(1)}, \ldots, w^{(m)}$ and bias vectors $b^{(1)}, \ldots, b^{(N)}$. Suppose the l^{th} layer has activations given by $a^{(l)} = \sigma(z^{(l)})$, where $z^{(l)} = w^{(l-1)}a^{(l-1)}$ for $l \geq 2$ and $a^{(1)} = x$, where $x \in \mathbb{R}^d$ is a single input of d features. Suppose a loss function $L = L(w, b)$ has been specified for the network. Define the *error* associated with the j^{th} neuron in layer l by

$$\delta_j^{(l)} \equiv \frac{\partial L}{\partial z_j^{(l)}}.$$

Define the *Hadamard product* operation on two vectors componentwise by

$$(v \circ w)_i \equiv v_i w_i.$$

Also, define $\frac{\partial L}{\partial a}$ and $\frac{\partial L}{\partial w}$ componentwise, i.e.

$$\left(\frac{\partial L}{\partial a}\right)_j \equiv \frac{\partial L}{\partial a_j} \text{ and } \left(\frac{\partial L}{\partial w}\right)_{jk} \equiv \frac{\partial L}{\partial w_{jk}}.$$

This allows us to state the algorithm in vectorized form, which along with the Hadamard product yields a more computationally efficient implementation [13].

Algorithm 2.3. Backpropagation on a single training example with a feedforward neural network.

- Input: Initialize $w^{(l)}$ and $b^{(l)}$ for all $l = 1, \cdots, N-1$ and set $a^{(1)} = x$.
- Feedforward to compute activations: $z^{(l)} = w^{(l-1)}a^{(l-1)} + b^{(l-1)}$ and $a^l = \sigma(z^l)$, for $l = 2, \ldots, N$.
- Compute output error: $\delta^N = \frac{\partial L}{\partial a^{(N)}} \circ \sigma'(z^{(N)})$.
- Backpropagate to compute errors: $\delta^{(l)} = (w^{(l)})^T \delta^{(l+1)} \circ \sigma'(z^{(l)})$, for $l = N-1, \ldots, 2$.
- Output: Set $\frac{\partial L}{\partial w^{(l-1)}} = \delta^{(l)}(a^{(l-1)})^T$ and $\frac{\partial L}{\partial b^{(l-1)}} = \delta^{(l)}$, for $l = 2, \cdots, N$.

www.manaraa.com

The correctness of the algorithm follows from the chain rule. We have

$$\delta^{(l)} = \frac{\partial L}{\partial z^{(l+1)}}\left(\frac{\partial z^{(l+1)}}{\partial z^{(l)}}\right)^T = (w^{(l)})^T\delta^{(l+1)} \circ \sigma'(z^{(l)}),$$

$$\frac{\partial L}{\partial w^{(l-1)}} = \frac{\partial L}{\partial z^{(l)}}\left(\frac{\partial z^{(l)}}{\partial w^{(l-1)}}\right)^T = \delta^{(l)}(a^{(l-1)})^T,$$

$$\frac{\partial L}{\partial b^{(l-1)}} = \frac{\partial L}{\partial z^{(l)}}\left(\frac{\partial z^{(l)}}{\partial b^{(l-1)}}\right)^T = \delta^{(l)}.$$

Notice that as specified, backpropagation only computes the gradients of the loss with respect to a single input x. In order to train the neural network, however, we require the gradients of the empirical risk,

$$\frac{\partial R_{emp}}{\partial w} = \frac{1}{n}\sum_x \frac{\partial L}{\partial w} \qquad \text{and} \qquad \frac{\partial R_{emp}}{\partial b} = \frac{1}{n}\sum_x \frac{\partial L}{\partial b}.$$

This naively suggests that in order to apply gradient descent to update the weights and biases, we must first compute the loss gradients for each training example and sum them up. Unfortunately, such an approach, called *batch gradient descent*, would cause the network to train very slowly. We remedy this by changing how we apply gradient descent.

2.2.2 Stochastic Gradient Descent

The most widely used learning algorithm for neural networks is *stochastic gradient descent* (SGD) [11]. Rather than perform backpropagation on the entire data set before updating the weights via gradient descent, i.e. batch gradient descent, one instead performs backpropagation on a smaller *mini-batch* of data chosen from the set at random and updates the weights only using that batch. This introduces uncertainty into the gradient calculation since we are estimating the gradient for the entire data set using the gradient for the mini-batch. Because this estimate is noisy the weights and biases may not move precisely down the gradient at each iteration of gradient descent. This noise can be advantageous, however [10].

www.manaraa.com

Suppose we randomly draw a mini-batch M of size m from the training set D. Then unbiased estimators for the gradients of R_{emp} are

$$\frac{\partial R_{emp}}{\partial w} \approx \frac{1}{m} \sum_{x \in M} \frac{\partial L}{\partial w} \qquad \text{and} \qquad \frac{\partial R_{emp}}{\partial b} \approx \frac{1}{m} \sum_{x \in M} \frac{\partial L}{\partial b}.$$

This suggests that we can apply gradient descent with these estimators instead to get the update rules

$$w_{n+1} = w_n - \eta \frac{1}{m} \sum_{x \in M} \frac{\partial L}{\partial w},$$

$$b_{n+1} = b_n - \eta \frac{1}{m} \sum_{x \in M} \frac{\partial L}{\partial b}.$$

This process is the gist of the SGD algorithm given below, based on [7].

Algorithm 2.4. Stochastic gradient descent update for a single epoch.

- Input the training set D, minibatch size m, and learning rate η.
- Randomly initialize the weights w and biases b.
- While elements in D have not been sampled do the following:
- Randomly sample, without replacement, a minibatch of m examples from the training set D.
- Use backpropagation to compute the weight and bias gradients of the empirical risk over the entire mini-batch.
- Update weights and biases using gradient descent with learning rate η.

When using SGD in practice it is useful to run SGD several times. Each run of SGD is called an *epoch*. The goal is to run SGD over enough epochs that the loss function starts to stabilize around a minimum. In practice, the number of epochs needed to get convergence can range from just a handful to several hundred. Such convergence, however, depends on specifying the learning rate to be just small enough

www.manaraa.com

to move down the gradient on each step. Since the loss functions used with neural networks are usually highly non-convex we can rarely guarantee that such a minimum is in fact the global minimum sought after. Interestingly enough, however, this is rarely a problem in practice [10].

SGD is the preferred learning algorithm for neural networks because it is usually much faster than batch gradient descent, especially when dealing with very large data sets. Also, SGD often results in better solutions due to the random noise of the algorithm, which prevent the algorithm from getting stuck in the closest local minimum to the initialized values of weights and biases [10].

Before concluding this section, we mention that in practice the loss function chosen for a neural network is usually either the mean square loss

$$L = \frac{1}{2}(y - a^{(N)})^2,$$

or the log loss (or cross entropy)

$$L = -y \log(a^{(N)}) - (1 - y) \log(1 - a^{(N)}).$$

While the mean square loss is perhaps the simplest of the two, the log loss prevents saturation of activation functions in the output layer. We will discuss saturation in the next section.

2.3 Practical Considerations

When implementing a deep learning algorithm in practice there are other factors that can substantially affect how well the algorithm performs. We survey a few of those factors here along with some of the current thinking on how to deal with them.

2.3.1 Model Selection

Model selection is the process of choosing which model setup to use in the learning process. Examples of model selection include choosing between neural networks or

www.manaraa.com

linear regression, which features to use, which loss function to use, the learning rate or mini-batch size in SGD, how many hidden layers a neural network should have, and which activation functions to use in a network.

The specific parameters chosen in the model selection and learning process are called *hyperparameters*. Examples of hyperparameters include the regularization parameter in a loss function, the learning rate in gradient descent, the batch size and number of epochs in SGD, and any other parameter that affects the performance of the learning algorithm. Several heuristics have been developed to choose the hyperparameters that give the best overall performance.

To perform model selection it is recommended to analyze the collection of models on a separate data set from the training set. Doing so makes it less likely that we will overfit the data with too complex a model that will fail to generalize well to unseen data. This separate data set is called the *validation set*.

A common, though laborious, technique is to select the hyperparameters through trial and error [13]. In using this technique it is recommended to start by deciding first which network architecture to use. Once an architecture has been specified one modifies the other hyperparameters in the problem one by one while holding the others fixed. Suppose those hyperparameters are the regularization parameter, learning rate, mini-batch size, and the number of epochs to run SGD. One approach then might be to first modify the learning rate while holding the others fixed, then modify the regularization parameter, then the mini-batch size, and finally the number of epochs. The goal is to find the combination that minimizes the number of misclassifications on the validation set.

A slightly modified approach to trial and error is to use a grid search to find the optimal combination of hyperparameters. In this method one specifies a grid of possible values for each hyperparameter and then goes linearly through each combination of values to select the optimal choice of hyperparameters. This method is usually

www.manaraa.com

quicker than trial and error, but requires specifying a grid of appropriate values for each hyperparameter.

In recent years more elaborate, but generally more efficient, techniques have been proposed for finding hyperparameters. One such technique is to use a random search in place of grid search [3]. Another technique uses Bayesian optimization techniques that model a learning algorithm's generalization error performance as a Gaussian process. The search for automated techniques for choosing hyperparameters is an active area of machine learning research [13].

2.3.2 Overfitting

Overfitting occurs when a learning algorithm performs very well on the training data but poorly on unseen data. This often occurs when the hypothesis class chosen is in some sense too large, in which case the hypothesis class is said to have *high variance*. Because the Universal Approximation Theorem states that neural networks can learn pretty much any decision boundary, they are highly susceptible to overfitting.

To deal with overfitting several techniques are used. The first and perhaps most obvious thing to try is to just collect more data. By collecting more training data, Theorem 1.2 guarantees that, with probability one, the empirical risk will better estimate the actual risk provided the ERM principle is consistent, i.e. that the neural network will generalize better. While this is a perfectly valid thing to try, collecting new data is often very labor intensive and expensive to do, and hence not usually ideal.

Another technique to try is to use a simpler hypothesis class. In the case of neural networks this means choosing a simpler network with fewer hidden layers or fewer neurons. Doing so would just amount to performing a model selection where the models are neural networks with varying numbers of neurons or hidden layers.

Generally, the preferred technique to deal with overfitting is *regularization*. Reg-

www.manaraa.com

ularization typically involves modifying the loss function by adding a penalty term, which acts to penalize weights that are too large or too skewed. Usually the penalty term chosen has the form $\frac{\lambda}{n}\|w\|^2$, where the norm is usually (but not always) the L^2 norm, w is a column vector containing all weights in the network (but not the biases), and $\lambda > 0$ is some *regularization parameter*. In the log loss case, the regularized loss function then becomes (in vectorized form)

$$L = -y \log(a^{(N)}) - (1-y) \log(1-a^{(N)}) + \frac{\lambda}{n}\|w\|^2.$$

The strength of regularization depends on the value λ. Taking $\lambda \approx 0$ gives a loss function that is more prone to overfitting but less biased, while taking λ to be large gives a loss function less prone to overfitting but with high bias. Note that λ is an example of a hyperparameter, so choosing lambda is done using the usual methods of model selection.

Note that in principle the opposite of overfitting can occur as well. This is called underfitting, and occurs when a learning algorithm performs poorly on *both* the training data and on unseen data. Due to the Universal Approximation Theorem this is rarely a problem with neural networks. In any case, this occurs when the hypothesis class chosen is in some sense too small. Its then said to have *high bias*. Underfitting can be corrected simply by extending the hypothesis class (e.g. by adding more hidden layers), or by adding extra features.

2.3.3 Saturation

Saturation occurs when a neuron with a sigmoid-like activation function receives an input that is large in absolute value. Consider the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$. When $|z|$ is large, $|\sigma(z)| \approx 1$, hence its derivative $\sigma'(z) \approx 0$. This means when backpropagation computes the error associated with this neuron we get $\delta \approx 0$, hence $\frac{\partial L}{\partial w} \approx 0$ and $\frac{\partial L}{\partial b} \approx 0$. This implies that SGD will learn the weights and bias of this neuron very slowly.

www.manaraa.com

Perhaps the most obvious remedy for saturation is to simply choose a different, non-sigmoid activation function. One popular example of such a function is the ReLU $\sigma(z) = \max\{0, z\}$. Such a function has the property that its derivative is a step function, which means positive inputs can never cause saturation no matter how large the inputs are. Other advantageous properties of the ReLU include sparse activation (only about 50% of neurons in a network are activated at any one time), scale invariance, and efficient computation [10]. Another popular choice is the softplus function $\sigma(z) = \log(1 + e^z)$, a smooth approximation to the ReLU.

Another way to mitigate the learning slowdown caused by saturation is to choose a nice loss function whose gradient doesn't depend on $\sigma'(z)$. One can show that the log loss has this property. This then prevents neurons in the output layer from saturating since the gradient will only near zero when $y \approx \sigma(z)$, which is ideal.

2.3.4 Initialization

The first step of backpropagation requires initializing the weights and biases. It turns out that how this initialization is performed can strongly affect how the network learns. To see an example of this, suppose we initialize all the weights and biases to zero. Then each neuron will pass on the same constant value, which implies that the weights will repeatedly update to the same values and thus learning will not take place.

It is generally preferred in practice to initialize the weights and biases randomly from some small interval $[-\varepsilon, \varepsilon]$. The interval should be small especially if the neurons in the first layer are sigmoids, since initializing with weights and biases too large in absolute value will cause those neurons to saturate. One practical scheme for assuring this is to sample randomly from a uniform $(0, 1)$ distribution and rescale the values to be in the interval. Another scheme is to sample values randomly from a Gaussian distribution with mean 0 and variance $\frac{1}{m_{in}}$, where m_{in} is the number of input weights

www.manaraa.com

into each neuron in the first hidden layer [2]. This assures that the chosen values are localized around 0 just enough to prevent saturation.

2.3.5 Network Architecture

Since the complexity of a neural network is governed in part by how many hidden layers and neurons it has it makes sense that the architecture of a neural network can affect how well it learns. An example of this was shown with the Universal Approximation Theorem. Neural networks with no hidden layers are rather weak, while those with at least one hidden layer are capable of learning any function (given enough neurons). It is thus usually preferable to choose a neural network with at least one hidden layer. Moreover, it is often the case that an architecture with more hidden layers and fewer neurons per layer can outperform an architecture with few hidden layers with many neurons per layer [7]. These *deep neural networks* naturally reflect what it means to perform deep learning since they are composed of large cascades of nonlinear transformations.

www.manaraa.com

Chapter 3

Application: Handwriting Recognition

We conclude with an application of deep learning to the historically significant problem of handwriting recognition. As the name suggests, handwriting recognition is the problem of training a computer to recognize handwritten input from paper documents or other devices. This has important applications in document imaging, signature verification, bank check processing, and other areas. As it happens, attempting to perform handwriting recognition *without* machine learning turns out to be a nigh on impossible task due to all the variations present in human handwriting. With machine learning, however, the problem can be solved easily with just a few lines of code.

The first step in handwriting recognition is preprocessing, which uses algorithms to binarize, normalize, sample, smooth, and denoise the input sample. The next step is segmentation, which involves separating words in the sample into individual characters [13]. Once the sample has been broken into simple characters, the last step is to classify what those characters are. It is this type of classification problem for which deep learning is well-suited.

The training set we will use for this problem is the one from the MNIST , or modified NIST, database. This set is a preprocessed and segmented set of 70,000 labeled training examples of numerical characters that were collected at the National Institute of Science and Technology (NIST) in the 1990s [12]. Following standard practice, we take 60,000 samples for the training set and the remaining 10,000 samples as a *test set* used to evaluate how well the learning algorithm is generalizing.

www.manaraa.com

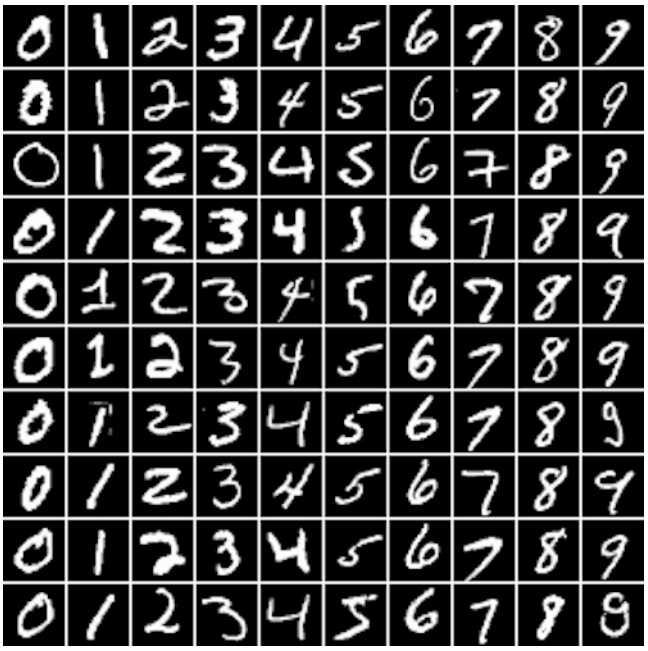

Figure 3.1 Examples from the MNIST dataset. Each block represents a distinct example. [7].

Now, each training example is a 28×28 pixel grayscale image, where each of these pixels is represented numerically by a value between 0, representing white, and 1, representing black, with the values in between representing various shades of gray. Each such pixel value will be used to represent a feature, so in total the training set will have $28 \times 28 = 784$ distinct features. This will correspond to an input layer of 784 neurons. Each training example is pre-labeled with its correct digit classification, i.e. with values from 0 to 9. Since there are 10 digits for classification, the output layer will consist of 10 neurons.

In performing digit classification we first note that a completely random classifier would have a 1 in 10 chance of classifying any inputted digit correctly, yielding a classification accuracy of 10%. We also note that a human would be expected to classify about 99% or more digits correctly (albeit a whole lot slower than a computer).

www.manaraa.com

All computation will be performed on a standard 1.3 GHz Intel Core i5 processor using the Python *scikit-learn* library. For each scenario we use the log loss with no regularization unless specified, and we use sigmoid activations for each neuron.

To begin, we attempt a neural network with no hidden layers, in this case a neural network with a 784-10 architecture. We use SGD with a learning rate of 0.1, a mini-batch size of 10, and train for 30 epochs. Running SGD with these inputs yields a test classification accuracy of 92.13%. Note that the accuracy on the training set is similar, so overfitting is not a problem here. This is certainly much better than random, but still nowhere near as well as a human would be expected to do.

Next, we attempt a neural network with one hidden layer containing 30 neurons, i.e. a 784-30-10 architecture. We again run SGD using the same inputs over 30 epochs. Running SGD then yields an accuracy of 96.31%. The accuracy on the training set suggests some overfitting may be happening. We can get slightly better performance out of this algorithm by shrinking the learning rate to 0.01 (it may be stuck oscillating about a minimum), and setting the regularization parameter to 5.0. Doing this yields 96.84%, accuracy, roughly a half-percent improvement.

Keeping in mind the regularization problem above, we now attempt a slightly larger neural network with a 784-100-10 architecture. We run SGD this time with a learning rate of 0.5, regularization parameter of 5.0, mini-batch size of 10, and train for 60 epochs. Doing so yields an accuracy of 97.46%. We can improve this result further by shrinking the learning rate to 0.1 and running SGD for another 30 epochs to get an accuracy of 98.27%.

At this point it is tempting to ask whether we can significantly improve accuracy further by simply adding more neurons, or by adding more hidden layers. This problem has, in fact, been extensively studied, and it turns out that accuracy can indeed be improved quite a bit. The current record accuracy for MNIST digit recognition is 99.77%, and was obtained in 2012 using a convolutional neural network, a special

www.manaraa.com

type of deep neural network that is particularly well-suited to image classification problems [5]. For a more detailed list of historical records set with MNIST see [12].

www.manaraa.com

Bibliography

[1] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from Data: A Short Course*. AML Book, 2012.

[2] Yoshua Bengio. "Practical Recommendations for Gradient-Based Training of Deep Architectures". In: *Lecture Notes in Computer Science Neural Networks: Tricks of the Trade* (2012). DOI: 10.1007/978-3-642-35289-8_26. URL: https://arxiv.org/pdf/1206.5533v2.pdf.

[3] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* (2012).

[4] Amitabha Chakravarty. *Deep Neural Networks - solution to many learning problems*. URL: https://www.linkedin.com/pulse/deep-neural-networks-solution-many-learning-amitabha-chakravarty (visited on 05/20/2017).

[5] D. Ciresan, U. Meier, and J. Schmidhuber. "Multi-Column Deep Neural Networks for Image Classification". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition* (2012). DOI: 10.1109/cvpr.2012.6248110. URL: https://arxiv.org/pdf/1202.2745.pdf.

[6] G. Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In: *Mathematics of Control, Signals, and Systems* 2.4 (Oct. 1988). DOI: 10.1007/bf02551274.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2017.

[8] O. Guler. *Foundations of Optimization in Finite Dimensions*. Springer, 2009.

[9] Andrej Karpathy. URL: https://computing.ece.vt.edu/~f15ece6504/slides/L3_backprop.pptx.pdf (visited on 05/20/2017).

[10] Yann A. Lecun et al. "Efficient BackProp". In: *Lecture Notes in Computer Science Neural Networks: Tricks of the Trade* (2012). DOI: 10.1007/978-3-642-35289-8_3. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

www.manaraa.com

[11] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521.7553 (2015). DOI: 10.1038/nature14539. URL: https://www.nature.com/nature/journal/v521/n7553/pdf/nature14539.pdf.

[12] Yann Lecun, Corinna Cortes, and Chris Burges. *THE MNIST DATABASE*. URL: http://yann.lecun.com/exdb/mnist/ (visited on 05/20/2017).

[13] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/.

[14] V.N. Vapnik. "An Overview of Statistical Learning Theory". In: *IEEE Transactions on Neural Networks* 10.5 (Sept. 1999). DOI: 10.1109/72.788640. URL: http://www.mit.edu/~6.454/www_spring_2001/emin/slt.pdf.

[15] V.N. Vapnik. *Statistical Learning Theory*. Wiley, 1998.

www.manaraa.com