Iowa Research Online
The University of Iowa's Institutional Repository

University of Iowa
Iowa Research Online

Theses and Dissertations

Summer 2019

The effectiveness of numerical approximation for dynamic programming problems

Wyatt Jones
University of Iowa

Follow this and additional works at: https://ir.uiowa.edu/etd

Part of the Economics Commons

Copyright © 2019 Wyatt Jones

This dissertation is available at Iowa Research Online: https://ir.uiowa.edu/etd/6968

Recommended Citation
Jones, Wyatt. "The effectiveness of numerical approximation for dynamic programming problems." PhD (Doctor of Philosophy) thesis, University of Iowa, 2019.
https://doi.org/10.17077/etd.r7aw-1ipf

Follow this and additional works at: https://ir.uiowa.edu/etd
Part of the Economics Commons

www.manaraa.com

THE EFFECTIVENESS OF NUMERICAL APPROXIMATION FOR DYNAMIC PROGRAMMING PROBLEMS

by

Wyatt Jones

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Economics
in the Graduate College of
The University of Iowa

August 2019

Thesis Supervisor: Professor Rabah Amir

المنارة للاستشارات

www.manaraa.com

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Wyatt Jones

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Economics at the August 2019 graduation.

Thesis Committee: ______________________
Rabah Amir, Thesis Supervisor

____
Barrett Thomas

____
Anne Villamil

____
Suyong Song

____
Nicholas C. Yannelis

المنارة للاستشارات
www.manaraa.com

ABSTRACT

The motivation of this thesis is the study of numerical methods for solving dynamic programming problems. In the first chapter, I present methods for reducing the computational cost imposed by the curse of dimensionality in the action and state space that occurs when solving for a Markov perfect equilibrium (MPE). These methods circumvent the computational problems for Markov perfect models in which the size of the state space or action space is large. This is because these methods are able to overcome the issue with approximating a highly non-linear or even discontinuous value function and thus allow the algorithm to use a small subset of the state space to approximate the rest of the value function. I use a model from the dynamic IO literature which is a dynamic oligopoly model with heterogeneous firms to evaluate the difference between using multidimensional Chebyshev polynomials and using artificial neural nets (ANN) for approximation, and present issues that arise when trying to use the gradient of the value function for approximations with Hermite interpolation. I also discuss how value function approximation with continuous functions can be used to find the optimal action quickly through the use of gradient based optimization techniques.

In the second chapter I examine the use of reinforcement learning to solve a high dimensional operations research problem. This methodology expands upon the value function approximation used in the first chapter through the use of an algorithm that uses both a value function approximation and a policy function approximation. I focus on the traveling salesman problem (TSP) and train recurrent neural networks (RNN) that, given a set of city

المنارة للاستشارات
www.manaraa.com

coordinates, output a probability distribution over the next city to visit in a route. Using route labels provided by Google's OR-Tools TSP solver I train multiple network architectures using supervised learning. I compare the performance of these architectures to the performance when using the route length as a reward signal to train the networks using reinforcement learning. I show that supervised learning is a useful tool for the optimization of hyperparameters that will be used in reinforcement learning, and to evaluate the performance improvement of an architecture change. I also provide evidence that while reinforcement learning is a more general optimization framework than using handcrafted heuristics, in practice it is necessary to build a neural network architecture specific to the problem of interest and that a network architectures ability to be trained using supervised learning does not guarantee the ability to be trained using reinforcement learning.

PUBLIC ABSTRACT

The motivation of this thesis is to research how to use recent advancements in applied mathematics, and computer science to allow for economic and operations research models to study more complex environments. In the first chapter I use a model from the economic field of industrial organization that has many firms competing in a market. Each firm has a product that consumers think has a different level of quality. Since there are so many possible combinations of each firm having a specific quality level the model can only be solved when there are only a few firms. I use the recent advancements in computer science with the use of artificial neural networks (ANN) to expand the number of firms that can be in the model. I also discuss how to combine this method with other advancements in applied mathematics and why ANN are an improvement over the current methodology.

In the second chapter I use the recent advancements in recurrent neural networks (RNN), and reinforcement learning to solve the traveling salesman problem (TSP). In this problem there are many cities and a salesman must decide the route through all the cities and back home in the shortest possible distance. This is one of the oldest problems in operations research and through many years several methods have emerged that perform very well on this problem. These methods took many years to develop and they do not perform well when the TSP's problem statement is changed slightly. Therefore there is interest in using RL to automatically discover these methods for the TSP and other similar problems. In the second chapter I detail how the implementation of the RNN must be built specifically for the problem of interest which limits how much more general this methodology is over

المنارة للاستشارات
www.manaraa.com

existing methods, how to use supervised learning (SL) to get the correct settings so that RL will work, and discuss the issue of reproducibility in RL.

المنارة للاستشارات
www.manaraa.com

TABLE OF CONTENTS

LIST OF TABLES . . . viii

LIST OF FIGURES . . . ix

CHAPTER

1 VALUE FUNCTION APPROXIMATION FOR DYNAMIC GAME MODELS 1

- 1.1 Introduction . . . 1
- 1.2 Model . . . 2
  - 1.2.1 Numerical example . . . 4
  - 1.2.2 Pakes-McGuire algorithm . . . 7
- 1.3 Avoiding the curse of dimensionality in state space . . . 10
  - 1.3.1 Approximating the value function with Lagrange interpolation . 18
  - 1.3.2 Approximating the value function with Hermite interpolation . . 21
- 1.4 Conclusion . . . 26

2 CHOOSING THE RIGHT NEURAL NETWORK ARCHITECTURE FOR THE TRAVELING SALESMAN PROBLEM . . . 28

- 2.1 Introduction . . . 28
- 2.2 Previous work . . . 31
- 2.3 Experimental design . . . 34
  - 2.3.1 Unidirectional encoder/decoder . . . 36
  - 2.3.2 Bidirectional encoder/decoder . . . 37
  - 2.3.3 Pointer network . . . 37
  - 2.3.4 Attention layer . . . 38
- 2.4 Results . . . 39
  - 2.4.1 Supervised learning . . . 39
    - 2.4.1.1 Hyperparameter and architecture choice . . . 40
    - 2.4.1.2 In-sample variance . . . 41
  - 2.4.2 Reinforcement learning . . . 42
    - 2.4.2.1 Hyperparameter and architecture choice . . . 42
    - 2.4.2.2 In-sample variance . . . 43
- 2.5 Conclusion . . . 43

APPENDIX

A APPENDIX TO CHAPTER 1 . . . 54

المنارة للاستشارات
www.manaraa.com

A.1 Avoiding the curse of dimensionality in the action space 54
A.2 Plots 58
A.3 Multidimensional complete Chebyshev polynomial approximation 65

B APPENDIX TO CHAPTER 2 66

B.1 Policy Gradient Method 66
B.2 Recurrent Neural Networks 68
B.3 Additional Architecture and Hyperparameter Choices 69

REFERENCES 71

للاستشارات المنارة
www.manaraa.com

LIST OF TABLES

Table

2.1 Table of supervised learning experimental results 46

2.2 Table of reinforcement learning experimental results 49

المنارة للاستشارات

www.manaraa.com

LIST OF FIGURES

Figure

1.1 Plot of both versions of $g(\omega)$ 5

1.2 Plot showing run time for n firms 11

1.3 The first four Chebyshev polynomials 13

1.4 Lagrange and Hermite Chebyshev approximation of Sin(x) 15

1.5 Lagrange and Hermite Chebyshev approximation of f(x) 16

1.6 ANN approximation of f(x) 17

2.1 Multiple runs of the unidirectional encoder/decoder using supervised learning Attention=Luong, LR=1e-3, Number of cells=128 Each color represents a run of this method with a different random initialization 45

2.2 Multiple runs of the unidirectional encoder/decoder using supervised learning Attention=Luong, LR=1e-3, Number of cells=128 Each color represents a run of this method with a different random initialization 47

2.3 Multiple runs of the unidirectional encoder/decoder using supervised learning Attention=Bah, LR=1e-3, Number of cells=128 Each color represents a run of this method with a different random initialization 48

2.4 Multiple runs of the unidirectional encoder/decoder using reinforcement learning Attention=Luong, LR=1e-3, Number of cells=128 Each color represents a run of this method with a different random initialization 50

2.5 Multiple runs of the unidirectional encoder/decoder using reinforcement learning Attention=Luong, LR=1e-3, Number of cells=128 Each color represents a run of this method with a different random initialization 51

2.6 Multiple runs of the unidirectional encoder/decoder using reinforcement learning Attention=Bah, LR=1e-3, Number of cells=128 Each color represents a run of this method with a different random initialization 52

المنارة للاستشارات
www.manaraa.com

2.7 Multiple runs of the unidirectional encoder/decoder using reinforcement learning Attention=Luong, LR=1e-4, Number of cells=128 Each color represents a run of this method with a different random initialization 53

A.1 The equilibrium value, investment policy and pricing policy fit on an 18x18 grid with 12 degree multidimensional Chebyshev polynomials 59

A.2 The equilibrium value, investment policy and pricing policy fit on an 12x12 grid with 12 degree multidimensional Chebyshev polynomials 60

A.3 The equilibrium value, investment policy and pricing policy fit on an 6x6 grid with 6 degree multidimensional Chebyshev polynomials 61

A.4 The equilibrium value, investment policy and pricing policy fit on an 18x18 grid using a neural net with a single fully connected layer with 324 nodes and tanh activation function 62

A.5 The equilibrium value, investment policy and pricing policy fit on an 12x12 grid using a neural net with a single fully connected layer with 144 nodes and tanh activation function 63

A.6 The equilibrium value, investment policy and pricing policy fit on an 6x6 grid using a neural net with a single fully connected layer with 144 nodes and tanh activation function 64

المنارة للاستشارات
www.manaraa.com

CHAPTER 1
VALUE FUNCTION APPROXIMATION FOR DYNAMIC GAME MODELS

1.1 Introduction

The analysis of dynamic industrial organization (I.O.) models is limited by the computational burden of computing equilibrium policies. These policies are typically computed by solving for the fixed point of a function whose domain is a subset of $\mathbb{R}^n$, where n is the dimension of the state space (Pakes & McGuire, 1992). In dynamic I.O. models, the dimension of the state space is typically the number of potentially active firms. In the Ericson and Pakes (1995) framework, each firm's state is an element of a discrete grid with m possible values. If we impose that the policy functions are symmetric, so that each firm has the same policy function, then the number of elements in the state space grows exponentially in the number of states per firm. Additionally, if there are multiple decision variables, the number of objective function evaluations required to find the optimal policy grows exponentially in the number of decision variables. These curses of dimensionality in the state and action spaces limit the applications that the Ericson and Pakes (1995) framework can be applied to (Aguirregabiria and Nevo (2013); Doraszelski and Judd (2007); Pakes and McGuire (2001); Weintraub, Benkard, and Van Roy (2008)).

This paper presents adaptations to the existing methodology that circumvent these computational problems for Markov Perfect models in which the size of the state space or action space is large. This is because these adaptations are able to overcome the issue with approximating a highly non-linear or even discontinuous value function and thus allow the

المنارة للاستشارات
www.manaraa.com

algorithm to use a small subset of the state space to approximate the rest of the value function.

In this paper, I present methods for reducing the computational cost imposed by the curse of dimensionality in the action and state space that occurs when solving for a Markov Perfect Equilibrium (MPE).

The rest of the paper is organized as follows. In Section 1.2, I outline the dynamic industry model and describe the relationship between my work and previous literature. In Section 1.3, I present methods for avoiding the curse of dimensionality in the state space, the difference between using multidimensional Chebyshev polynomials and using Artificial Neural Nets (ANN) for approximation, and issues that arise when trying to use the gradient of the value function for approximations. Section 1.4 presents conclusions and directions for future research. Finally, in Section A.1, I present methods for avoiding the curse of dimensionality in the action space and how it relates to value function approximation.

1.2 Model

In order for their model to be tractable, Ericson and Pakes (1995) make assumptions about firm's behavior. In this paper I will define a model as tractable if there are methods that use a reasonable amount of computational resources to solve for an equilibrium or analytical tools can be used to solve for an equilibrium. In an industry with n firms, each firm is assumed to have a single dimensional state variable ω_j, where $j \in \{1, \ldots, n\}$, that determines the relative position of a firm in an industry compared to the strength of competition outside of the industry. In this paper ω_j represents the quality to consumers of

www.manaraa.com

firm j's product. States are assumed to be discrete and typically $\omega_j \in \mathbb{Z}_+$. This assumption reduces the multidimensional heterogeneity that exists between firms in the real world to a single dimension. Additionally, the discretization of the state space reduces the accuracy with which industry dynamics can be represented.

Let $S = \{\omega_1, \ldots, \omega_n\} \in \mathbb{Z}_+^n$ denote the industry state. It is assumed that current profits for firm j are weakly decreasing in S_{-j} and weakly increasing in ω_j. Therefore an increase in firm j's own state improves its position in the market and if other firms increase their own state, firm j's current period profits will decrease.

The state S changes stochastically conditional on the current state, and actions of every firm in the industry. Firms' choose their actions to maximize the expected present discounted value of profits as a function of the state S. To further reduce the complexity of the framework Ericson and Pakes (1995) make the necessary assumptions to assure that the state follows a Markov process. Thus firms are restricted to Markovian strategies given the current state and do not directly observe the actions of the other firms.

While these assumptions limit the framework's ability to generate the variation observed in the data they are necessary to construct a more tractable model however; even after these assumptions it is not computationally tractable to solve industry model instances with more than a few firms (Weintraub et al., 2008). To further reduce the complexity of the framework so that it is computationally tractable, Doraszelski and Satterthwaite (2010) impose additional symmetry restrictions. By adding additional structure to the framework they prove the existence of a symmetric equilibrium in pure strategies.

In this paper firm j's value function when in industry state S will be expressed by

المنارة للاستشارات
www.manaraa.com

$$V_j(S) = \max_{\mu_j \in \Gamma(S)} \{\pi(S, \mu_j, \mu_{-j}) + \beta \mathbb{E}[V_j(S')|S, \mu_j, \mu_{-j})]\}$$

where μ_j is the policy of firm j which will could include the pricing, investment, and entry or exit decision, $\Gamma(S)$ is the set of feasible actions in state S, μ_{-j} is a vector containing the policies of all firms other than firm j, and S' is the state transitioned to, conditional on S, μ_j and μ_{-j}.

1.2.1 Numerical example

Throughout this paper I will refer to a numerical example that is identical to the example used in Doraszelski and Judd (2012). The example is a simplified version of the Pakes McGuire quality ladder model (Doraszelski & Judd, 2012). I will focus on the case where there are two firms choose the price of their product and quantity of investment to improve their state. The state variable $S = \{\omega_1, \omega_2\}$ represents the vertically differentiated quality of their product. Let $\omega_j \in \{1, \ldots, 18\}$ and the approximation nodes $S_i = \{\omega_{1,i}, \omega_{2,i}\} = \{\lfloor \frac{i}{18} \rfloor, i \mod 18\}$ for $i = 1, \ldots, 324$. Consumer l's utility function for product j is given by $u_{lj} = g(\omega_j) - p_j + \epsilon_{lj}$ where ϵ_{lj} is an extreme type 1 error. The function $g(\omega)$ maps the discrete states, ω_j, to the quality consumers perceive for product j. Both Pakes and McGuire (1992) and Doraszelski and Judd (2012) used the function

$$g(\omega) = \begin{cases} 3\omega - 4 & \text{if } \omega \leq 5 \\ 12 + \ln(2 - \exp(16 - 3w)) & \text{if } \omega > 5 \end{cases}$$

however, as Figure 1.1 shows this function is not defined for a range between 5 and 6.

www.manaraa.com

This causes a problem when attempting to approximate the model with a continuous state space. Therefore, I will use the function

$$g(\omega) = \frac{14.7}{1 + \exp(4 - 1.2\omega)}$$

which Figure 1.1 shows is continuous and a close approximation of the original function.

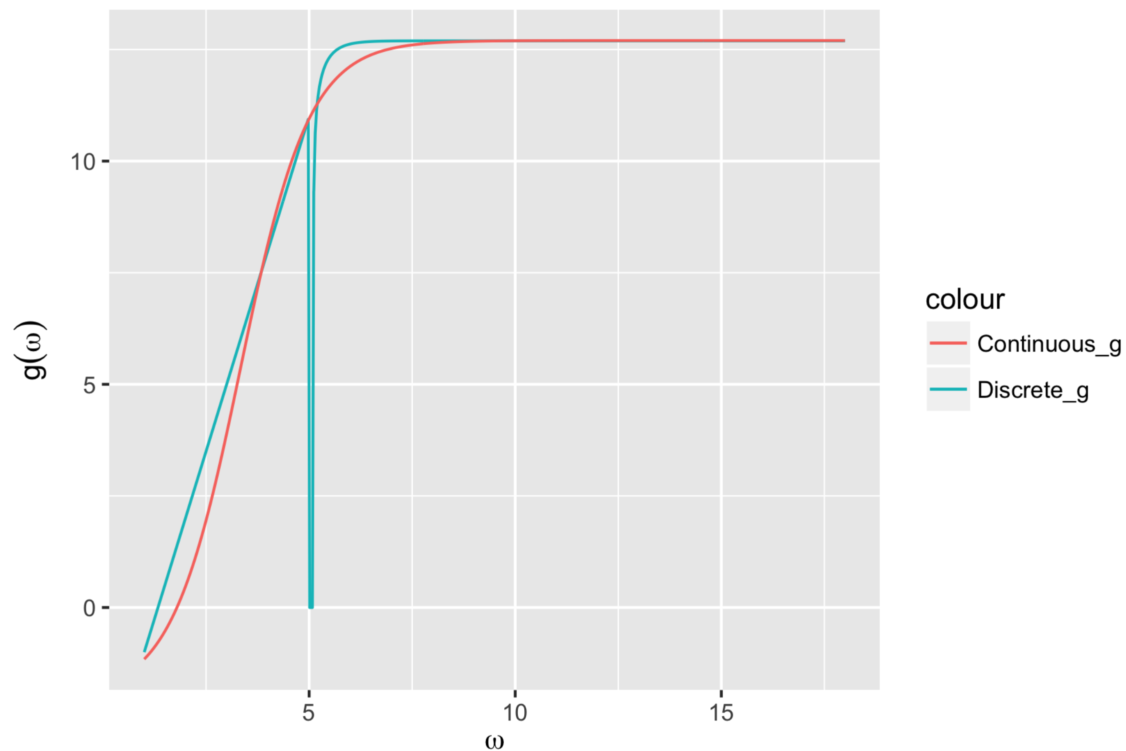

Figure 1.1: Plot of both versions of $g(\omega)$

The single period objective function for firm 1 as a function of the industry state, S, the actions of firm 1, μ_1, and the other firm's actions, μ_2, can be expressed with the

www.manaraa.com

following equation. Let $\mu_j = \{p_j, x_j\}$, where p_j represents the price charged to consumers for product j and x_j represents the level that firm j invests into the quality of their product. M represents the market size, and c represents the marginal cost of production. The single period payoff function is

$$\pi_1(S, \mu_1, \mu_2) = M \frac{\exp(g(S_1) - p_1)}{1 + \exp(g(S_1) - p_1) + \exp(g(S_2) - p_2)}(p_1 - c) - x_1.$$

In order to calculate the expected future value for a policy μ_1 let

$$\mathbb{E}[\hat{V}(S')|S, \mu_1, \mu_2] = \sum_{\omega_1'=1}^{m} \sum_{\omega_2'=1}^{m} \hat{V}(\{\omega_1', \omega_2'\})\Pr(\omega_1'|\omega_1, x_1)\Pr(\omega_2'|\omega_2, x_2),$$

where the probability of transitioning to the state ω' given investment x and current state ω is given by

$$\Pr(\omega'|\omega, x) = \begin{cases} \frac{(1-\delta)\alpha x}{1+\alpha x} & \text{if } \omega' = \omega + 1 \\ \frac{1-\delta+\delta\alpha x}{1+\alpha x} & \text{if } \omega' = \omega \\ \frac{\delta}{1+\alpha x} & \text{if } \omega' = \omega - 1 \end{cases},$$

$$\Pr(\omega'|1, x) = \begin{cases} \frac{(1-\delta)\alpha x}{1+\alpha x} & \text{if } \omega' = 2 \\ \frac{1+\delta\alpha x}{1+\alpha x} & \text{if } \omega' = 1 \end{cases},$$

$$\Pr(\omega'|m, x) = \begin{cases} \frac{1-\delta+\alpha x}{1+\alpha x} & \text{if } \omega' = m \\ \frac{\delta}{1+\alpha x} & \text{if } \omega' = m - 1 \end{cases},$$

www.manaraa.com

and δ represents the probability that the outside alternative increases in quality, and α represents the efficacy of investment in generating increases in efficiency [1]. Through out the paper I will reference the MPE of this model found when using the parameters in Pakes and McGuire (1992) which are, $\beta = 0.925$, $M = 5$, $c = 5$, $\alpha = 3$, and $\delta = 0.7$.

1.2.2 Pakes-McGuire algorithm

The Pakes and McGuire (1992) algorithm is an iterative backward solution method which uses numbers stored in memory from the previous iteration to calculate an update to those numbers and runs until the updated numbers satisfy a convergence condition. At the start of the algorithm N approximation nodes, $\{S_i\}_{i=1}^{N}$, must be chosen. Each node represents a possible industry state that can be visited in the model. It is also necessary to initialize an $\epsilon > 0$ for the convergence criterion and to initialize the value function.

The algorithm uses the assumption that firms use symmetric strategies so that only one value function needs to be calculated. For notational purposes I will denote the firm for which the value function is calculated as firm 1. Since we are interested in finding only symmetric MPE, the Pakes and McGuire (1992) algorithm uses the calculation of firm 1's policies in iteration k-1 to determine the policies of all other firms in iteration k. For example, in the numerical example the pricing and investment policy for firm 2 in state S_i, and iteration k is represented as $\mu_{i,-1}^{k} = \{p_{i,2}^{k}, x_{i,2}^{k}\} = \{p_{i',1}^{k-1}, x_{i',1}^{k-1}\}$ where i' is such that if $S_i = \{\omega_1, \omega_2\}$ then $S_{i'} = \{\omega_2, \omega_1\}$.

[1] A key property of this choice of transition probabilities, convexity of the cdf (or its integral) in the investment variable, plays a key role in establishing existence and properties of Markov equilibrium/policies in related contexts in economic dynamics see (Amir, 1996a, 1996b, 1997).

المنارة للاستشارات
www.manaraa.com

More generally, for each iteration k the algorithm uses firm 1's policy from last iteration, μ_1^{k-1} to determine the policy that other firms will choose, μ_{-1}^k, and then calculates μ_1^k based on μ_{-1}^k and v_i^{k-1}. Then the algorithm checks a convergence condition by evaluating the distance between the previous iteration's value function and the current value function.

Algorithm 1 Pakes McGuire Algo

1: **procedure** Choose N approximation nodes $\{S_i\}_{i=1}^N$, error tolerance for value function updating ϵ.

2: Let $v_i^0 = 0$ for $i = 1, \ldots, N$

3: **while** $\left\|v_1^k - v_1^{k-1}\right\| > \frac{\epsilon(1-\beta)}{2\beta}$ **do**

4: **for** $i = 1, \ldots, N$ **do**

5: Solve the Bellmann equation and store in memory $v_{i,1}^k$ and $\mu_{i,1}^k$

$$v_{i,1}^k = \max_{\mu_{i,1}^k \in \Gamma(S_i)} \pi(S_i, \mu_{i,1}^k, \mu_{i,-1}^{k-1}) + \beta \mathbb{E}[\hat{V}(S')|S_i, \mu_{i,1}^k, \mu_{i,-1}^{k-1})]$$

6: **end for**

7: Let $k = k + 1$

8: **end while**

9: **end procedure**

The CPU run time of this algorithm can be approximated with the following formula. Let K represent the number of value function iterations required for convergence, N the number of states in the state space, and f_{evals} be the number of function evaluations required

المنارة للاستشارات
www.manaraa.com

to find the optimal policy

$$\text{CPU run time} = (K)(N)(f_{evals})(\text{Computation time of f})$$

$$H(S_i) = \max_{\mu_{i,1}^k \in \Gamma(S_i)} f(S_i, \mu_{i,1}^k) \quad . \tag{1.1}$$

$$f(S_i, \mu_{i,1}^k) = \pi(S_i, \mu_{i,1}^k, \mu_{i,-1}^{k-1}) + \beta \mathbb{E}[\hat{V}(S')|S_i, \mu_{i,1}^k, \mu_{i,-1}^{k-1})]$$

This algorithm suffers from several computational issues. There are curses of dimensionality in both the state space and action space that can cause the algorithm to take too long to run. For a fixed number of iterations required for convergence, which I will denote as K, if the industry of interest has just 10 firms then N can be too large. In the numerical example in this paper, each firm has 18 possible states and so for 10 firms, the size of the state space would be $18^{10} = 3,570,467,226,624 = N$. If the time it takes to compute $f(S, \mu)$ takes just 0.01 seconds then it would take 413,248 days to just evaluate the function once in each state. There are many different ways to decrease the computational cost imposed by increasing the number of firms, and in Section 1.3 I will show several methods of value function approximation to overcome this issue.

In applications of the Ericson and Pakes (1995) framework firms can have many decision variables. Typically a firm chooses the price of its product, how much to invest to improve their state, and whether to enter or exit. However many firms can have a much more complex decision making process. In the numerical example used in this paper firms just choose the price of their product and level of investment to improve the quality of their product. In order to choose the optimal action using a simple grid search procedure with grid points separated by a 1e-5 space there would need to be a 400,000 by 500,000 grid

المنارة للاستشارات
www.manaraa.com

representing each possible choice that the firm could make (since in equilibrium the optimal price varies from 6 to 11 and the optimal level of investment varies from 0 to 4). Thus, the number of function evaluations per state per iteration, which I will denote as f_{evals}, would be 200,000,000,000. An algorithm implemented this way would take 23,148 days to find the optimal action in a single state in a single iteration. There are many different methods that allow the optimal action to be found with a high degree of accuracy very quickly and Section A.1 discusses how to use these methods to decrease f_{evals}.

In many applications the run time required to compute an equilibrium determines the way in which the theoretical model was written (Benkard, Jeziorski, & Weintraub, 2015; Doraszelski & Pakes, 2007). Often, this leads to including only 2-4 firms, or using a profit function that can be computed outside the algorithm's main loop to reduce the number of decision variables.

Thus in order to develop richer models of industry dynamics that are a closer representation of reality, it is important to understand the factors that determine a model's run time and the methods that can be used to make a model computationally tractable.

1.3 Avoiding the curse of dimensionality in state space

The curse of dimensionality in the state space is often a more difficult issue to overcome. Typically, it is easier to solve a dynamic programming model computationally when the state space is discrete rather than continuous and the dimension of the state space is small. This is true in the Ericson and Pakes (1995) framework. However, when the state space is large it is computationally more efficient to approximate the discrete state space

المنارة للاستشارات
www.manaraa.com

with continuous functions. This can be seen in the following graph which compares the run time for this model when it had a discrete grid state space and when the state space is continuous with a Artificial Neural Network Approximation.

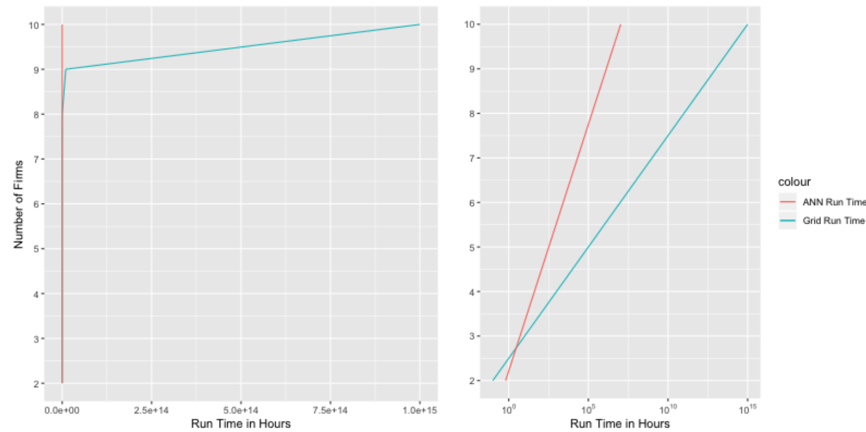

Figure 1.2: Plot showing run time for n firms

This plot shows that as the dimension of the state space increases that there is an exponential increase in the difference in run time between the method that used a discrete grid and the method that used a continuous approximation. The method with a continuous state space is able to obtain the reduction in computational cost by reducing the number of states that need to be visited for each dimension by approximating the value function between observed states.

In order to approximate the value function there are many different methods that can be used. A simple example is approximating the function $v(S) = \sin(S)$ where $S \in [0, 4\pi]$. Using the Lagrange data, $\{S_i, v_i\}_{i=1}^{N}$, where $S_i = \frac{4\pi i}{N}$, $v_i = \sin(S_i)$, and $N = 6$ then there

المنارة للاستشارات
www.manaraa.com

are many appropriate interpolation methods. Pakes and McGuire (1992) discussed the use of polynomial interpolation. Polynomial interpolation with Chebyshev polynomials uses a sum of orthogonal polynomials to approximate the function v.

Chebyshev polynomials can be defined using a recursive definition or using a trigonometric definitive. Using the trigonometric definition, a n degree Chebyshev polynomial of the first kind is defined as

$$T_n(S) = \cos(n \ \arccos(\frac{2S - S_{min} - S_{max}}{S_{min} - S_{max}})).$$

If $S_{min} = -1$ and $S_{max} = 1$ then the first four polynomials will be

$$T_0(S) = 1$$

$$T_1(S) = S$$

$$T_2(S) = 2S^2 - 1 \qquad ,$$

$$T_3(S) = 4S^3 - 3S$$

$$T_4(S) = 8S^4 - 8S^2 + 1$$

المنارة للاستشارات

www.manaraa.com

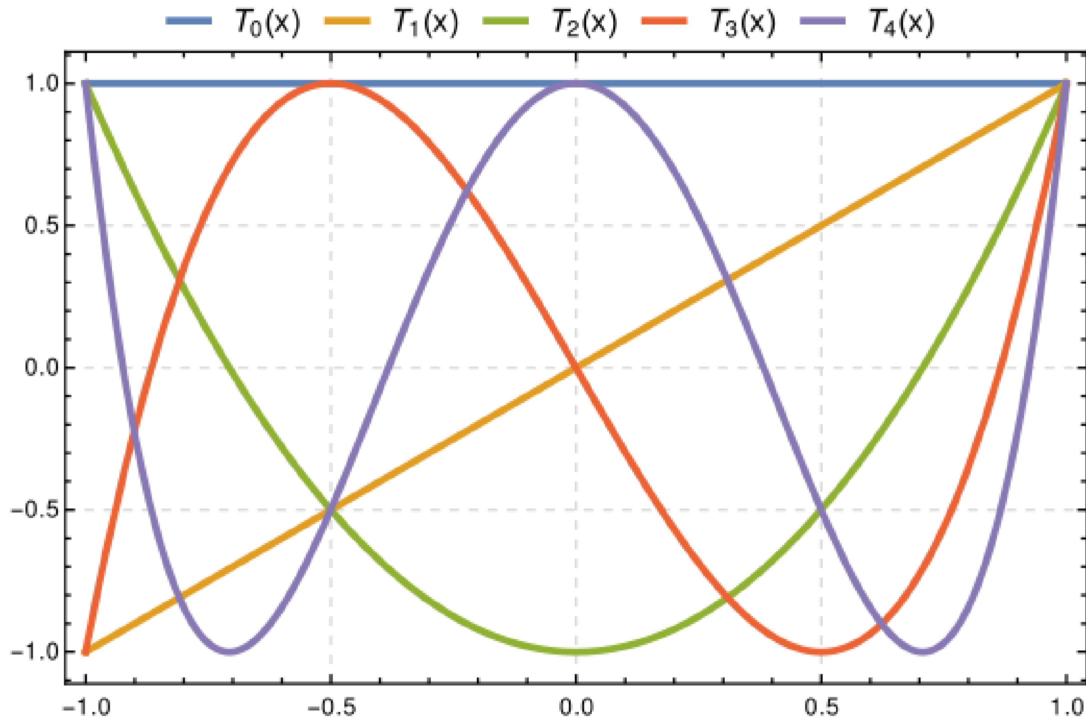

Figure 1.3: The first four Chebyshev polynomials

and then approximation can be written as $\hat{V}(S;\boldsymbol{b}) = \sum_{k=0}^{n} b_k T_k(S)$. When interpolating a function with Chebyshev polynomials it is important to choose the approximation nodes based off of the roots of the polynomials. The Chebyshev nodes for one dimension are given by $S_i = (z_i + 1)(S_{max} - S_{min})/2 + S_{min}$ and $z_i = -\cos((2i-1)\pi/(2m))$ where m is the number of approximation nodes. In order to approximate the function $v(S)$ using the Lagrange data $\{S_i, v_i\}_{i=1}^{N}$, where $v_i = v(S_i)$, it is necessary to solve for the coefficients, b, using

$$\min_{\boldsymbol{b}} \sum_{i=1}^{N} (\hat{V}(S_i;\boldsymbol{b}) - v_i)^2.$$

المنارة للاستشارات
www.manaraa.com

This problem can be solved easily with OLS or with the closed form Chebyshev Approximation Formula. This approximation method has a few cases where it performs poorly. One case where the approximation is poor is when N is too small however, for a fixed N the approximation can be improved by fitting the polynomial approximation using Hermite data. Hermite data consists of the approximation node, the function evaluated at the approximation node and also the gradient of the function evaluated at the approximation node. In order to fit the Chebyshev polynomial using Hermite data, $\{S_i, v_i, d_i\}_{i=1}^{N}$, where $d_i = \frac{dv(S_i)}{dS}$, the coefficients are found by solving the following OLS problem.

$$\min_{\boldsymbol{b}} \sum_{i=1}^{N} \Big((\hat{V}(S_i; \boldsymbol{b}) - v_i)^2 + (\frac{\partial \hat{V}(S_i; \boldsymbol{b})}{\partial S} - d_i)^2 \Big).$$

Chebyshev polynomials are an attractive choice for fitting a function using Hermite data since the derivative of type 1 Chebyshev polynomials are a function of type 2 Chebyshev polynomials. Since $\frac{\partial T_n(S)}{\partial S} = nU_{n-1}(S)$ and type 2 Chebyshev polynomials can be defined by the following equation.

$$U_n(S) = \frac{\sin((n+1)\arccos(\frac{2S - S_{min} - S_{max}}{S_{min} - S_{max}}))}{\sin(\arccos(\frac{2S - S_{min} - S_{max}}{S_{min} - S_{max}}))}.$$

By incorporating the gradient information a function can be approximated to a given degree of accuracy with fewer approximation nodes. Figure 1.4 shows how Hermite approximation is able to approximate the function $\sin(x)$ with four approximation nodes to a much better degree of accuracy than the Lagrange approximation.

المنارة للاستشارات
www.manaraa.com

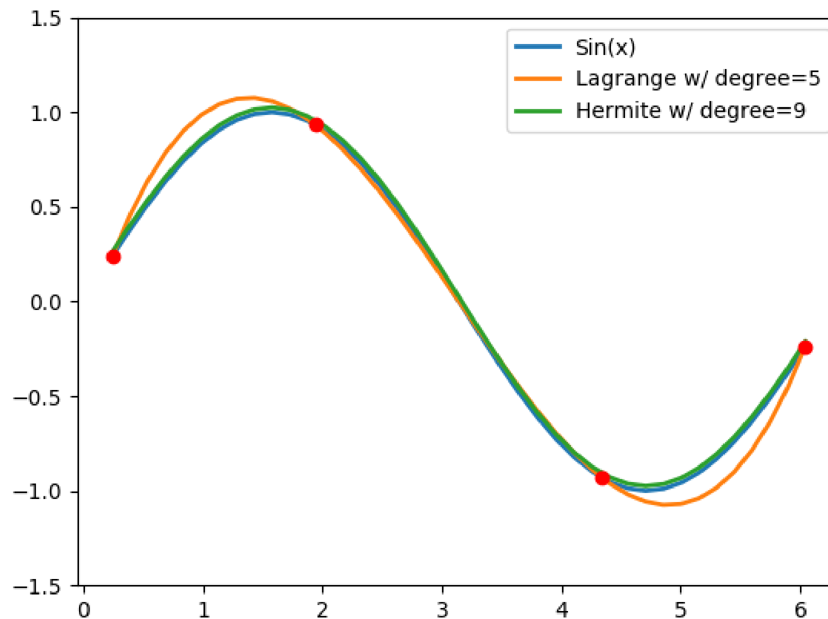

Figure 1.4: Lagrange and Hermite Chebyshev approximation of Sin(x)

Another case where polynomial approximations perform poorly is when there are large discontinuities or non-linearities in the function that is being approximated. When trying to approximate a discontinuous function polynomial approximations can often show the Gibbs phenomenon which is when the approximating polynomial overshoots or undershoots the function around a jump discontinuity. This overshooting/undershooting persists even when the number of approximation nodes increases. This can be seen in Figure 1.5 when Lagrange and Hermite Chebyshev polynomials are used to approximate the function $f(x)$ with 10 approximation nodes.

www.manaraa.com

$$f(x) = \begin{cases} 1 & \text{if } x \leq 1/2 \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

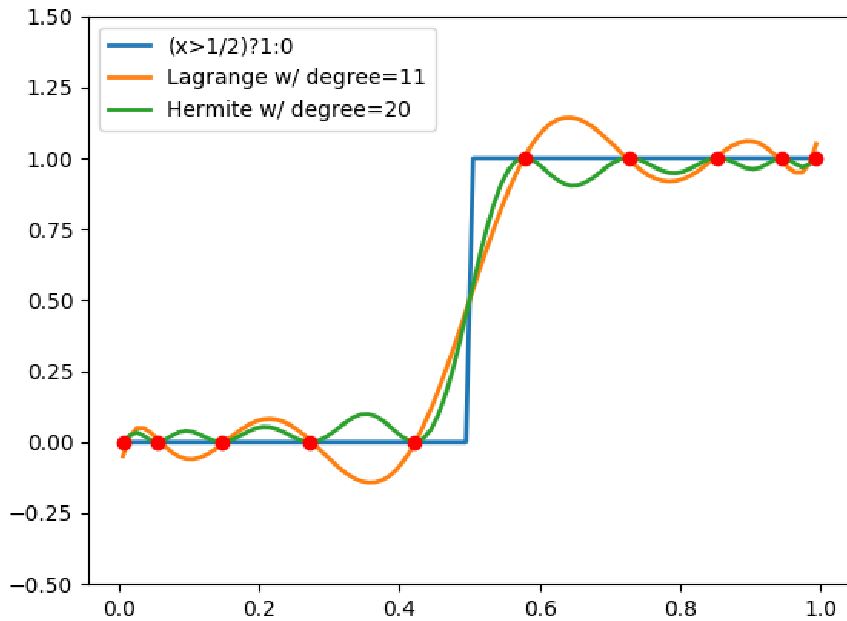

Figure 1.5: Lagrange and Hermite Chebyshev approximation of f(x)

In order to overcome this issue it is necessary to use an approximation techinque that is able to approximate the function locally. Local approximation can be done by fitting a piece-wise linear function, but these functions are not smooth and the kink that occurs at each approximation node can cause problems when used for value function approximation. Cubic splines are smooth and can approximate the function locally however this method

المنارة للاستشارات
www.manaraa.com

does not generalize to n dimensions.

Polynomials are an attractive choice for approximating the value function since they are universal approximators and so they can fit all functions however, in practice polynomials often require a computationally prohibitive amount of data to fit some functions to a desired degree of accuracy. Park and Sandberg (1991) show that Artificial Neural Nets (ANN) are also universal approximators. Artificial Neural Nets are able to approximate functions locally to avoid the Gibbs phenomenon, and also scale very easily to approximating multidimensional functions. There has also been a large influx of computationally efficient software packages for training and evaluating neural nets. For an introduction to ANN please refer to Jain, Mao, and Mohiuddin (1996).

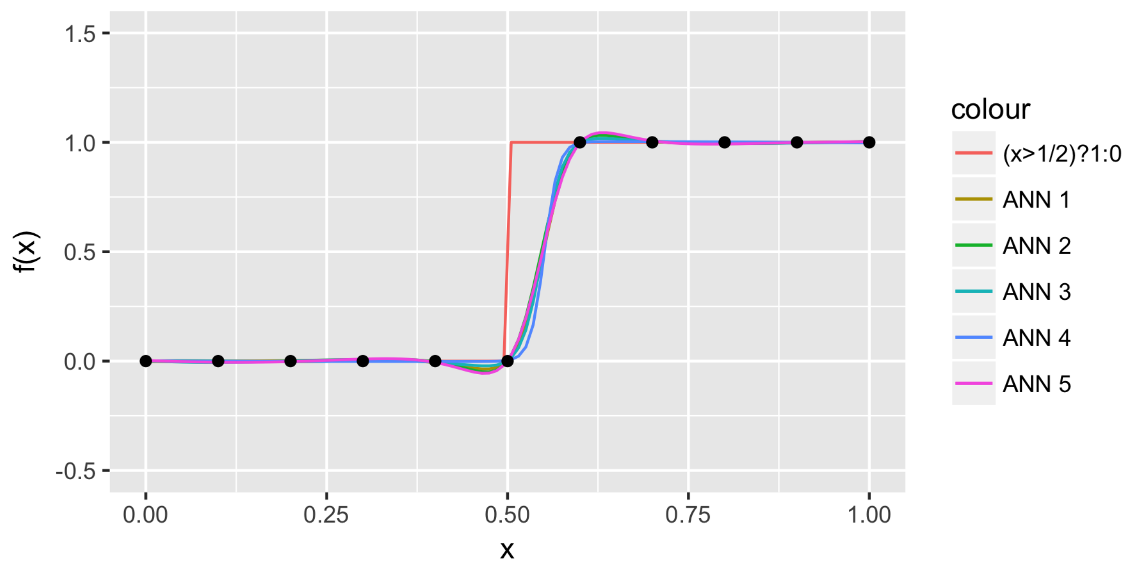

Figure 1.6: ANN approximation of f(x)

www.manaraa.com

Figure 1.6 shows how a ANN with a single hidden layer with 10 nodes was able to approximate the function without the same issues that Chebyshev polynomials had. The graph was created using the same ANN architecture, trained for the same number of iterations, but each of the 5 ANN started out with a different set of random initial weights. This figure also shows a drawback of using ANN, which is that the quality of the approximation can depend on the initialization of the parameters. This can be seen with the fifth approximation. They also sometimes get stuck in a local minimum and provide very bad approximations. Zainuddin and Pauline (2008) showed that using different network structures such as a Radial Basis Function Network or a Wavelet Neural Network can improve the accuracy of ANN by reducing the frequency that the ANN gets stuck in a local minimum. Llanas, Lantarón, and Sáinz (2008) show that ANN can approximate any discontinuous function with a single hidden layer and Llanas and Lantarón (2007) also show that the Hermite approximation problem can be solved using ANN.

1.3.1 Approximating the value function with Lagrange interpolation

Using either Chebyshev polynomials or ANN to approximate a function allows for an accurate representation of a continuous function using a relatively small set of points. In the following algorithm these techniques will be used to approximate the value function of firm 1. By approximating the value function, it is possible to reduce the number of approximation nodes required to solve for an equilibrium.

المنارة للاستشارات
www.manaraa.com

Algorithm 2 Pakes McGuire Algo with L-VFA

1: **procedure** Choose N approximation nodes $\{S_i\}_{i=1}^{N}$, error tolerance for value function updating ϵ.

2: Initialize $\boldsymbol{b}$, and $\{\mu_{i,-1}^{0}\}_{i=1}^{N}$.

3: **while** $\left\|\hat{V}_1^k - \hat{V}_1^{k-1}\right\| > \frac{\epsilon(1-\beta)}{2\beta}$ **do**

4: **for** $i = 1, \ldots, N$ **do**

5: Solve the Bellmann equation and store in memory $\hat{V}_{i,1}^k$ and $\mu_{i,1}^k$

$$v_{i,1}^k = \max_{\mu_{i,1}^k \in \Gamma(S_i)} \pi(S_i, \mu_{i,1}^k, \mu_{i,-1}^{k-1}) + \beta \mathbb{E}[\hat{V}(S'; \boldsymbol{b}) | S_i, \mu_{i,1}^k, \mu_{i,-1}^{k-1})]$$

6: Minimize the mean squared error and store in memory $\boldsymbol{b}$

$$\min_{\boldsymbol{b} \in \Omega} \sum_{i=1}^{N} \left(v_{i,1}^k - \hat{V}(S_i; \boldsymbol{b})\right)^2$$

7: Let $k = k + 1$

8: **end for**

9: **end while**

10: **end procedure**

The Pakes McGuire algorithm with Lagrange value function approximation (L-VFA) replaces the discretized value function with a continuous approximation. Since the value function can now be approximated by a continuous and differentiable function it is now possible to solve for an equilibrium of a model that uses a continuous state space. This is an attractive feature since empirically the return from investment is continuously distributed,

المنارة للاستشارات
www.manaraa.com

and since firms do not have a set of predetermined discrete levels of quality and so VFA allows for a more realistic theoretical model in addition to a more computationally efficient method for solving for an equilibrium.

Using the numerical example in this paper I compared the equilibrium found using two methods on six different grids of approximation nodes. The first method was using multidimensional Chebyshev polynomials to approximate the value function and the second method was a ANN with a single hidden layer. When using the Chebyshev polynomials I used a grid based on the location Chebyshev nodes which are defined by $x_i = (z_i + 1)(x_{max} - x_{min})/2 + x_{min}$ and $z_i = -\cos((2i-1)\pi/(2m))$ where m is the number of approximation nodes per firm. When using the ANN I used an equidistant grid of m points per firm. I examined the equilibrium value function and policy functions found when $m = 6, 12,$ and 18.

The results show that the ANN was able to approximate the value function to a high enough degree of accuracy that the equilibrium policy functions maintained their qualitative structure when $m = 6, 12,$ and 18. The multidimensional Chebyshev approximation was able to preserve the policy function's qualitative structure when $m = 12,$ and 18 however, when $m = 6$ the value function failed to converge. The approximation failed to converge because when the value function increases very quickly when firm 2's state is at a low value of about 3 and firm 1's state is between 3 and 6. This large rate of increase leads the approximation to exhibit the Gibbs phenomenon which can be seen in Figure A.2 and more prominently in Figure A.3 [2]. This is especially an issue because during the value function iterations the

[2]https://github.com/wmjones/VFA-for-dynamic-gamesMy Github Repository has the animated gifs of each of the 6 examples where each frame of the gif is a value function iteration, source code, and an easy to use makefile.

المنارة للاستشارات
www.manaraa.com

region that was overshot is reinforced and the approximation error propagates through the subsequent value function iterations.

These results suggest that ANN are able to approximate the value function to a higher degree of accuracy than Chebyshev polynomials when the number of approximation nodes is reduced. This is useful since if Chebyshev polynomials can approximate it with a sufficient degree of accuracy when $m = 12$ and ANN can approximate the value function with a sufficient degree of accuracy when $m = 6$, then with 10 firms VFA with Chebyshev polynomials is approximately 60 times faster than a method that uses $m = 18$ and VFA with ANN is approximately 60,000 times faster than a method that uses $m = 18$.

L-VFA can also be easily combined with other methods for reducing the size of the state space. For example, it is easy to use parallelization to reduce the computational cost of the algorithm by finding the value of each state in parallel. This approximately divides the run time of the algorithm by the number of threads that evaluation of each state is parallelized over. In the numerical example I parallelized over 8 threads to reduce computation time.

1.3.2 Approximating the value function with Hermite interpolation

Similar to how the approximation accuracy of one dimensional functions can be improved by fitting Hermite data rather than Lagrange data, the value function can be approximated using Hermite data. Cai and Judd (2015) discuss a method for easily obtaining the gradient of the value function in order to use Hermite data to approximate the value function. Cai and Judd (2015) showed that Hermite value function approximation (H-VFA) was able to either produce a higher accuracy in a given amount of time or attain the same

المنارة للاستشارات
www.manaraa.com

accuracy with much less computation time when compared to L-VFA.

If the right hand side of the Bellman equation is written as

$$\begin{aligned} & H(S) = \max_{\mu} f(S,\mu) \\ \text{s.t.} \quad & g(S,\mu) = 0, \\ & h(S,\mu) \geq 0 \end{aligned} \tag{1.3}$$

,

then the gradient of $H(S)$ is provided by

$$\frac{\partial H(S)}{\partial S_j} = \frac{\partial f}{\partial S_j}(S,\mu^*(x)) + \lambda_1^*(S)^T \frac{\partial g}{\partial S_j}(S,\mu^*(S)) + \lambda_2^*(S)^T \frac{\partial h}{\partial S_j}(S,\mu^*(S)).$$

For $j = 1,\ldots,n$ and n is the dimension of S. The computation of the gradient requires finding several derivatives and is difficult to compute. However, the problem can be rewritten by adding in the variable y such that

$$\begin{aligned} & H(S) = \max_{\mu,y} f(y,\mu) \\ \text{s.t.} \quad & g(y,\mu) = 0, \\ & h(y,\mu) \geq 0, \\ & S_j - y_j = 0, \quad j = 1,\ldots,n. \end{aligned} \tag{1.4}$$

.

Then the gradient of $H(S)$ is provided by

$$\frac{\partial H(S)}{\partial S_j} = \tau_j^*(S)$$

www.manaraa.com

where $\tau_j^*(S)$ is the Lagrange multiplier for the constraints $S_j - y_j = 0$.

There are several derivative-free constrained optimization software libraries that provide along with the solution to the maximization problem the Lagrange multiplier for each constraint such as MATLAB's fmincon function. This method allows for an easy method to obtain the gradient of the value function and thus able to use the Hermite data, $\{S_i, v_i, d_i\}_{i=1}^N$, where d_i is the gradient vector evaluated at S_i, to approximate the value function.

The coefficients to approximate the value function using multidimensional Chebyshev polynomials are obtained by solving the following equation and the notation for multidimensional Chebyshev polynomials is presented in Appendix A.3.

$$\min_{\boldsymbol{b}} \left\{ \sum_{i=1}^{N} \Big(v_i - \sum_{0 \le |\alpha| \le r} b_\alpha T_\alpha(S_i) \Big)^2 + \sum_{i=1}^{N} \sum_{j=1}^{n} \Big(d_{i,j} - \sum_{0 \le |\alpha| \le r} b_\alpha \frac{\partial}{\partial S_j} T_\alpha(S_i) \Big)^2 \right\}.$$

While the L-VFA algorithm presented in Cai and Judd (2015) is easily added to the Pakes McGuire Algorithm, a direct adaptation of the H-VFA algorithm has additional complications.

www.manaraa.com

Algorithm 3 Pakes McGuire Algo with H-VFA using Chebyshev polynomials

1: **procedure** Choose N approximation nodes $\{S_i\}_{i=1}^N$, error tolerance for value function updating ϵ.

2: Initialize $\boldsymbol{b}$, and $\{\mu_{i,-1}^0\}_{i=1}^N$.

3: **while** $\left\|\hat{V}_1^k - \hat{V}_1^{k-1}\right\| > \frac{\epsilon(1-\beta)}{2\beta}$ **do**

4: **for** $i = 1, \ldots, N$ **do**

5: Obtain the value function at S_i, and gradient by solving Equation (1.3) or Equation (1.4). Then store in memory $v_{i,1}^k$, $\mu_{i,1}^k$, and d_i^k.

6: Minimize the mean squared error and store in memory $\boldsymbol{b}$ using Hermite data $\{S_i, v_{i,1}, d_{i,1}\}_{i=1}^N$ by solving

$$\min_{\boldsymbol{b}} \left\{ \sum_{i=1}^{N} \left(v_i - \sum_{0 \le |\alpha| \le r} b_\alpha T_\alpha(S_i) \right)^2 + \sum_{i=1}^{N} \sum_{j=1}^{n} \left(d_{i,j} - \sum_{0 \le |\alpha| \le r} b_\alpha \frac{\partial}{\partial S_j} T_\alpha(S_i) \right)^2 \right\}$$

7: Let $k = k + 1$

8: **end for**

9: **end while**

10: **end procedure**

Step 5: in this algorithm causes complications. Since the reaction of other firms is taken into account, a firm's decision in the gradient calculation using Equation 1.3 would require the calculation of the gradient of the policy function. This is because the policy that the other firms choose depends on what the industry's state is. Thus in order complete Step 5: using Equation 1.3 it is necessary to determine the gradient of the policy function. Since

www.manaraa.com

the analytical gradient of the policy function cannot be obtained due to it being the argmax of the right hand side of the bellman equation, it is necessary to numerically approximate the gradient. This can be done easily with the central difference method. However the accuracy of the central difference method is limited by the step size which in this context is the distance between two approximation nodes. Thus in order to obtain an accurate approximation of the policy function there would need to be more approximation nodes which prevents H-VFA from decreasing the number of approximation nodes. This also prevents the use of more efficient optimization methods such as the L-BFGS algorithm.

If Step 5: is calculated using (1.4) then there can be an issue with the numerical optimization method converging. This is due to the fact that a derivative free algorithm is used to calculate the solution and that by adding in an additional n decision variables $\{y\}_{j=1}^{n}$ the number of function evaluations required to find the optimal action will grow significantly. Therefore the computational cost of finding the optimal action may cause the Pakes Mcguire Algorithm with H-VFA to be slower than Pakes Mcguire Algorithm with L-VFA.

It is also easy to incorporate other methods to reduce the size of the state space. Pakes and McGuire (1992) show that

$$V(\omega_1, \omega_2, \ldots, \omega_N) = V(\omega_1, \omega_{\pi(2)}, \ldots, \omega_{\pi(n)})$$

for any $n-1$ dimensional vector $\pi = [\pi(2), \ldots, \pi(n)]$ which is a permutation of $(2, \ldots, n)$. Therefore, the value function does not need to be evaluated at each distinct approximation node but instead only needs to be evaluated at the nodes where $\omega_2 \geq \omega_3, \ldots, \geq \omega_n$. This causes the number of approximation nodes in the state space to be calculated as $\frac{(m+n-1)!}{(m-1)!n!}$ instead of m^n. For 10 firms and $m = 18$ this leads to $N = 8,436,285$, when $m = 12$

www.manaraa.com

this leads to $N = 352,716$ and when $m = 6$ this leads to $N = 3,003$. Thus if it takes approximately 0.01 seconds to evaluate the function f, it will now only take 23.4 hours to evaluate the function $f(S, \mu)$ once in each state when $m = 18$, 0.98 hours to it when $m = 12$ and only 0.008 hours to do it when $m = 6$. This reduction in computation time allows for the exploration of richer models with larger state spaces which were previously unable to be analyzed due to computational constraints.

1.4 Conclusion

I have shown that by adding value function approximation to the Pakes McGuire algorithm it is possible to reduce the number of states needed per firm and in A I show how to incorporate numerical optimization methods with value function approximation to reduce the curse of dimensionality in the action space. I have also discussed the benefits that ANN have over multidimensional Chebyshev polynomials for value function approximation and the complications that arise from attempting to fit the value function using Hermite data in dynamic game models. These conclusions are supported by a numerical example of the Pakes McGuire quality ladder model.

The L-VFA numerical example when fit with ANN used a simple single hidden layer ANN. Future research could examine the benefits to using more complex network structure such as deep neural networks. Liang and Srikant (2016) show that the number of neurons needed to approximate a function decreases exponentially in the number of layers in the network. Future research could also incorporate other methods from approximate dynamic programming such as using post-decision state sampling to reduce the computational cost

للاستشارات المنارة
www.manaraa.com

of computing the expected value, or using ANN to construct a continuous implementation of the dynamic lookup table presented in Ulmer, Mattfeld, and Köster (2017).

المنارة للاستشارات
www.manaraa.com

CHAPTER 2
CHOOSING THE RIGHT NEURAL NETWORK ARCHITECTURE FOR THE TRAVELING SALESMAN PROBLEM

2.1 Introduction

The traveling salesman problem (TSP) is the oldest and one of the most well researched combinatorial optimization problems. The TSP is the problem of finding the shortest path through a set of geographically disbursed customers. In order to solve the TSP one needs to search the space of possible routes through a graph to find the route with the minimum total edge weights. Finding the optimal route in a TSP is NP-hard when the nodes are located in $\mathbb{R}^2$ and the edge weights are the euclidean distance between two points. In the TSP with 20 customers, which this paper focuses on, the problem's state is drawn from $\mathbb{R}^{2\times 20}$ and the action space includes 6.08×10^{16} possible routes.

In practice, modern TSP solvers use heuristic methods chosen specifically for the version of the TSP that is to be solved. Through decades of research and by evaluating many different potential heuristics, the operations research literature has been able to develop heuristics that perform well enough to solve the TSP with thousands of nodes. Unfortunately, these heuristics often do not perform well when the TSP's problem statement is altered slightly or on other similar combinatorial optimization problems.

Machine learning offers an alternative to existing approaches that would allow for the automatic discovery of heuristics which would remove the need for handcrafted heuristics and methods that can be applied to multiple problem specifications. Most machine learning methods rely on supervised learning, which trains a parameterized function by minimizing

المنارة للاستشارات
www.manaraa.com

the loss between a predicted output and an optimal label. Labels are correct output for a training data set that can be used to determine if the method's predictions are correct. However this method of training relies on access to optimal labels which are not available for combinatorial optimization problems. This is because the objective is to train a neural network so that it performs better to the current best approximation of the optimal policy and not to simply replicate the existing methods performance.

Reinforcement learning (RL) provides a solution to this issue because it does not require optimal labels to train but instead relies on reward feedback from the problem specification. Recent advancements in the RL literature have drawn widespread attention for their ability to out perform human professionals. Silver et al. (2016) applied reinforcement learning to approximate the optimal policy for the board game GO which is a dynamic game with a state space containing 2.082×10^{170} elements. While this framework did use some labels obtained from previous games played by professionals Silver et al. (2017) was able to train a policy that performs better than the previous version, beat every professional player that it has faced, and does not use any labels obtained from human play. (Hessel et al., 2017) applied reinforcement learning methods to approximate the optimal policy for 57 different games played on the Atari 2600 console. These 57 problems are dynamic in nature with a state space of $3,932,160$ elements and no clear method for specifying an analytical reward function. Hessel et al. (2017) was able to show that their application of RL methods are able to train neural networks that perform better than humans on all 57 games.

The advancements in the RL literature and research showing the exceptional performance on dynamic problems with large state spaces has motivated research into the

المنارة للاستشارات
www.manaraa.com

application of these methods for many problems in the operations research field. Recently multiple neural network architectures have been proposed to solve the TSP (Bello, Pham, Le, Norouzi, & Bengio, 2016; Deudon, Cournut, Lacoste, Adulyasak, & Rousseau, 2018; Nazari, Oroojlooy, Snyder, & Takáč, 2018; Vinyals, Fortunato, & Jaitly, 2015). However, the degree to which the neural network architecture needs to be specialized to the problem of interest to be trained using reinforcement learning has not been evaluated. Additionally, the relationship between an architecture's performance when trained using supervised and reinforcement learning is not well understood. This paper examines how performance changes as an architecture is specialized to the problem of interest and how performance changes based on the training methodology.

In order to determine the appropriate neural network architecture for a given problem researchers evaluate many different possible architectures and make iterative changes to the architecture in order to maximize performance. This papers shows that there is a major complication when trying to select the appropriate neural network architecture that will be trained using reinforcement learning. In Section 2.4 I present evidence that there is often not the necessary feedback to iteratively improve the architecture until the network is specialized enough to a given problem that it is able to improve performance through training with reinforcement learning. This is due to the fact that the subset of architectures that will effectively train on a given problem is very small compared to the number of potential architectures and that when the architecture is not specialized enough the feedback that the researcher observes is that the network simply does not improve while training is taking place. This makes it difficult to evaluate the value of different changes to the neural

المنارة للاستشارات
www.manaraa.com

network's architecture when the neural network is not learning. Section 2.4 also presents evidence that supervised learning can be used to provide feedback for hyperparameter and network architecture choices so that the network can then be more efficiently trained using reinforcement learning.

In Section 2.2 I review the background that is closely related to my work. In Section 2.3 I discuss the methodology I use to study the impact that architecture choice and training methodology have on model performance. In Section 2.4 I discuss the results from the experiments and in In Section 2.5 I present the conclusions and directions for future research.

2.2 Previous work

The Traveling Salesman Problem (TSP) is a well studied combinatorial optimization problem and many exact and approximate methods have been developed to solve it. The problem was first formulated in Menger (1932) and it is often used as a benchmark for many optimization methods. The current best exact dynamic programming algorithm for TSP has a complexity of $\mathcal{O}(2^n n^2)$ which makes it infeasible on problems larger than 40 points. While calculating the optimal route exactly is infeasible for most problems many approximate methods that rely on heuristic approaches have been able to perform well on problem instances with much more than 40 points. Modern TSP solvers, which rely on handcrafted heuristics that determine how to navigate the space of feasible policies efficiently are able to solve TSP instances with thousands of points.

Google's vehicle routing problem solver (OR-Tools) is a state of the art TSP solver which relies on a combination of local search algorithms and metaheuristics specifically

المنارة للاستشارات
www.manaraa.com

crafted for the TSP. OR-Tools applies hand-engineered heuristics such as 2-opt to move from one potential solution to the next and then uses a metaheuristic such as guided local search, which escapes local minima by putting a penalty on solution features that are known to decrease performance (Bello et al., 2016).

While these methods are very successful in solving the TSP the search heuristics are often not successful for newly encountered problems. Because search algorithms have the same performance when averaged over all problems, the appropriate heuristics for each problem must be carefully selected. The difficulty of finding the best heuristics for each problem has lead to research into finding a more general approach.

Motivated by recent success in using recurrent neural networks (RNN) with long-short term memory cells (LSTM) to solve problems with a sequence for input and output researchers have studied the application of RNN's to the TSP (Sutskever, Vinyals, & Le, 2014). Please see Appendix B.2 for a brief description of RNNs, LSTM cells and links for more details.

Training an artificial neural network is most commonly done using label of the correct output however, for the TSP labels are not available. Labels could be generated using an existing method but this would only allow for a replication in the existing method's performance. Reinforcement learning offers an alternative method to train the network which does not require labels, or an analytical reward function. Reinforcement learning instead trains the artificial neural network by interacting with an environment and attempting to update parameters so that the reward obtained from the network's policy is maximized. Recent improvements in this field include increasing performance by combining multiple

المنارة للاستشارات
www.manaraa.com

methods (Hessel et al., 2017), increasing the ability to use available computational resources (Babaeizadeh, Frosio, Tyree, Clemons, & Kautz, 2016), improving sample efficiency (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017), and has lead to researchers attempting to use reinforcement learning on new problems.

Recent research on the use of artificial neural networks to solve the TSP began with Vinyals et al. (2015) which introduced a network architecture called a Pointer Network. Vinyals et al. (2015) used a RNN with non-parametric softmaxes that was trained using supervised learning and showed that this network architecture was capable of approximating the optimal policy for the TSP 20. (Bello et al., 2016) showed that this approach was able to be trained using reinforcement learning and further improved the performance of the architecture presented by (Vinyals et al., 2015) through several different small architecture changes and various post processing steps. Nazari et al. (2018) have made alterations to the network architecture so that it can solve more general Vehicle Routing Problems. Deudon et al. (2018) show that the combination of artificial neural networks trained using reinforcement learning with handcrafted heuristics lead to improvements in model performance.

While (Vinyals et al., 2015) was able to show that Pointer Networks are able to be trained using supervised learning to solve the TSP and then (Bello et al., 2016) showed that the same architecture was able to be trained using reinforcement learning, it is unclear how much an artificial neural network's architecture needs to be specialized for a given problem in order to be able to approximate the optimal policy. The relationship between an architectures ability to be trained using supervised learning and its ability to be trained using reinforcement learning is also a topic which requires future research. These issues are

المنارة للاستشارات
www.manaraa.com

fundamental to determining if using artificial neural networks are indeed a more general approach than using handcrafted heuristic approaches or if the artificial neural networks need handcrafted architectures specialized to the specific problem of interest.

Motivated by these issues, this paper studies multiple neural network architectures of varying degree of specialization to the TSP and examines how they perform when trained using both supervised learning, and reinforcement learning. I also study the effect that small variations in the choice of hyperparameters will have on model performance. Specifically, the hyperparameters I will examine are the learning rate and the number of LSTM cells in each layer. The learning rate determines the step size for which the parameters are updated, and the number of LSTM cells in each layer affects the number of parameters that are to be trained. I will also study the impact that small network architecture changes have on model performance. Specifically, I will examine how changes to the way that the method determines weights for the input sequence have on performance.

2.3 Experimental design

In this section I propose an experimental design in order to study the affect that neural net architecture and hyperparameter choice has on both supervised and reinforcement learning. Please see Appendix B.3 for a brief description of hyperparameter and network architectures that were studied but not included in this experimental design. In order to parameterize the policy function I will use a RNN with LSTM cells. This is done by letting the probability of a given route be determined by the following

المنارة للاستشارات

www.manaraa.com

$$p_\theta(\pi|s) = p_\theta(y_1, \dots, y_T|x_1, \dots, x_T) = \prod_{t=1}^{T} p_\theta(y_t|y_1, \dots, y_{t-1}, c)$$

.

Let $s = (x_1, \dots, x_T)$ be the problem's state which is a sequence where x_i is the x-y coordinates for point i. Let y_t be the ID of the point traveled to at time t, θ is a vector of trainable parameters, and c is a context vector which the RNN uses to maintain information about the state. In order to prevent the policy from choosing a previously chosen location I store in memory a record of each location chosen and then penalize the probability of moving to that location so that it cannot be chosen again.

For each neural net architecture and each hyperparameter choice the method will be trained using both supervised and reinforcement learning. The loss for supervised learning will be the cross entropy loss between the parameterized policy and a near optimal policy provided by using the TABU search from Google's OR-Tools. Reinforcement learning will be done using the A3C algorithm (Babaeizadeh et al., 2016). For each architecture trained using reinforcement learning I use a critic network composed of 5 convolutional layers with a fully connected layer that outputs the final prediction for a given problem state.

In order to evaluate the network architectures sensitivity to hyperparameter choice I will evaluate each method with three different learning rates and three different numbers of LSTM cells. The learning rates will vary between 1e-2, 1e-3, and 1e-4 and the number of LSTM cells will vary between 32, 64, and 128 LSTM cells.

Each method will be optimized using the ADAM optimizer as is standard in the literature with the learning rate varying between each setting, and I will hold fixed the other

www.manaraa.com

hyperparameters of the ADAM optimizer by setting $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1\text{e-}8$.

The methods will be evaluated by determining how many times longer the parameterized policies route length is compared to the length of a route found using TABU search. I will use the average relative length for the last 10,000 problem instances that the method evaluated after training for 36 hours using 16 cores from Intel's Haswell CPU architecture. Each method is also run three times using different initial random seeds.

2.3.1 Unidirectional encoder/decoder

The first major Neural Network architecture that I use is directly from the Neural Machine Translation (NMT) literature (Sutskever et al., 2014). This is a general architecture which is designed to be trained using supervised learning and accept a sequence as input and predict a sequence as output. The parameterization consists of two RNN. The first RNN is called the encoder and it maps the input sequence $x_{t\,t=1}^{T}$ into a context vector c. The second RNN is called the decoder and it maps the context vector to the final output. To construct the context vector c the following formula is used

$$h_t = f(x_t, h_{t-1})$$

where h_t is the cell's hidden state, f is an LSTM and $c = h_T$. Once the encoder has constructed the context vector c the decoder then determines the output probability at each step using the formulas

$$p_\theta(y_t | y_1, \ldots, y_{t-1}, c) = g(y_{t-1}, s_t, c)$$

المنارة للاستشارات
www.manaraa.com

$$s_t = f(s_{t-1}, y_{t-1}, c_t)$$

where $g(y_{t-1}, s_t, c)$ is the softmax of the output of the decoder's LSTM cells and $f(s_{t-1}, y_{t-1}, c_t)$ is how the LSTM cells in the decoder updates its hidden state s_t. In this paper $p_\theta(y_t|y_1, \ldots, y_{t-1}, c)$ is the probability that the route moves to the location y in step t.

2.3.2 Bidirectional encoder/decoder

The second major Neural Network architecture that I use the Bidirectional Encoder/Decoder framework presented in Bahdanau, Cho, and Bengio (2014). This architecture is identical to the Unidirectional Encoder/Decoder except that the Bidirectional encoder consists of both a forward and a backward RNN. The forward RNN reads the input sequence in order from $t = 1$ to T and the backward RNN reads the input sequence from T to $t = 1$. The hidden state h_t is then constructed by concatenating the two vectors $h_t = [\overrightarrow{h}_t^\top; \overleftarrow{h}_t^\top]$. This architecture is less dependent on the order for which the sequence is encoded which is beneficial for the TSP because the order that the cities are encoded does not matter.

2.3.3 Pointer network

The final architecture is from Vinyals et al. (2015) which was constructed specifically to solve the TSP. In this architecture there is no longer an encoder and the probability of moving to the next point is simplified to the following

$$p_\theta(y_t|y_1, \ldots, y_{t-1}, c) = \text{softmax}(c)$$

للاستشارات المنارة
www.manaraa.com

where h_t is the LSTM's hidden state at step t and c is the context vector. The context vector is calculated using an attention layer.

2.3.4 Attention layer

The NMT literature finds that introducing a mechanism that will focus on part of the input sequence while decoding would increase performance and Vinyals et al. (2015) found that due to the spatial nature of the TSP that the attention layer would allow the RNN to be invariant to the input sequence's order.

In this paper I will study how the two most common attention mechanisms impact the performance of each method. The output of the attention mechanism is a new context vector c that is then input into the decoder. Each attention mechanism is constructed with the following formulas

$$c_t = \sum_{j=1}^{T} \alpha_{tj} h_j$$

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T} \exp(e_{tk})}$$

$$e_{tj} = a(s_{t-1}, h_j)$$

where t represents the current step during decoding, k represents the location in the input sequence, and a is a specific attention mechanism. The first specific attention mechanism that I use is the Bahdanau Attention Mechanism from Bahdanau et al. (2014) where e_{tj} is computed by

المنارة للاستشارات
www.manaraa.com

$$e_{tj} = v_\alpha^\top \tanh(W_\alpha s_{t-1} + U_\alpha h_j)$$

where $v_\alpha \in \mathbb{R}^n, W_\alpha \in \mathbb{R}^{n\times n}, U_\alpha \in \mathbb{R}^{n\times 2n}$ are weight matricies.

The second attention mechanism is the Luong Attention Mechanism from Luong, Pham, and Manning (2015) where e_{tj} is computed by

$$e_{tj} = s_{t-1}^\top W_\alpha h_j$$

where $W_\alpha \in \mathbb{R}^{n\times n}$ is a weight matrix. I will refer to these attention mechanisms as the Luong and Bahdanau attention respectively.

2.4 Results

I this section I analyze the experimental results. First I show that when training using supervised learning that as the Neural Network Architecture becomes more specialized to the problem of interest there is an increase in performance per training step holding all else constant. Then I show that that many methods that could be trained using supervised learning were not able to be trained using reinforcement learning.

2.4.1 Supervised learning

In this section I discuss the results from training each method with supervised learning where the labels used to train are obtained through the use of OR-Tool's TSP solver.

المنارة للاستشارات
www.manaraa.com

2.4.1.1 Hyperparameter and architecture choice

In Table 2.1 the results from training the 35 different methods to solve the TSP 20 are presented. It shows that while the methods that used a Unidirectional and Bidirectional encoder were able to achieve near optimal results on the TSP 20 that they had very inconsistent performance when hyperparameters were slightly changed.

Table 2.1 also shows that Unidirectional Encoder/Decoder was only able to train using the Luong Attention Mechanism, the Bidirectional encoder/decoder was able to be trained with both attention mechanisms and the Pointer Network had better performance when using the Bahdanadu Attention Mechanism. While the experimental evidence shows that the best run for the Unidirectional and Bidirectional Encoder/Decoder out performed the best run for the Pointer Network, the Pointer Network was more robust to changes in the hyper parameters and was more consistently able to achieve a performance within 10% of the optimal route length and was able to achieve high performance very quickly compared to the other architectures.

By comparing Figure 2.1, Figure 2.2, and Figure 2.3 we can see that while the best run that used a Unidirectional Encoder/Decoder architecture was able to achieve optimal results the average performance improvement per training step is less than the average for methods that used a Pointer Network architecture. This suggests that the Pointer Network is more consistently able to achieve performance improvements during the course of training than the Encoder/Decoder frameworks are.

المنارة للاستشارات
www.manaraa.com

2.4.1.2 In-sample variance

The experiment also provides evidence of significant in-sample variance. Identical runs of a method often yield very different results when only the initial random seed was changed. This can be seen in Figure 2.1 where the relative length of multiple different runs of the best performing Unidirectional Encoder/Decoder architecture are plotted over the time that the method was trained. This plot shows that a method that can yield optimal performance can have difficulty reproducing the same performance when rerun. This in-sample variance is also present in the best performing Bidirectional Encoder/Decoder architecture which can be seen in Figure 2.2. While the Unidirectional and Bidirectional Encoder/Decoder architectures exhibited a high degree of in-sample variance the Pointer Network does not. Figure 2.3 shows that the best performing Pointer Network architecture is able to train consistently to a low relative length but did not have a run that performed as well as the best Unidirectional or Bidirectional Encoder/Decoder architectures.

The in-sample variance that can occur with these methods can be a major problem for reproducibility because even if the architecture and hyperparameters are the same different initial random seeds can cause significant differences in performance for the method. The in-sample variance is also a problem for further research in this field because the evaluation of a newly proposed architecture can be noisy which makes tuning hyperparameters difficult. These problems could be avoided by performing many sample runs and reporting both the best case and average case for the method however, this can be extremely computationally intensive and many researchers choose to not report the frequency for which their architecture trained successfully.

المنارة للاستشارات
www.manaraa.com

2.4.2 Reinforcement learning

In this section I discuss the results from training each method with reinforcement learning.

2.4.2.1 Hyperparameter and architecture choice

Table 2.2 presents the average relative length of the best run for each method when trained using reinforcement learning. This table shows that the Unidirectional and Bidirectional Encoder/Decoder networks failed completely to train using reinforcement learning for any hyperparameter setting. Figure 2.4 and Figure 2.5 show that while these methods were able to be trained using supervised learning, their performance when first initialized is that of a random policy and that there is no improvement throughout the course of training. This behavior is consistent for all the different hyperparameter choices for the Unidirectional and Bidirectional Encoder/Decoder architectures. These results show that evaluating the effect that different hyperparameters have on an architecture's performance is difficult to determine when only using feedback from the networks performance when trained using reinforcement learning. This suggests that hyperparameter tuning is not possible when the neural network architecture is not specialized enough for the problem of interest so that reinforcement learning is possible. This can be a significant barrier for research in this area because when evaluating the performance of a newly proposed network architecture it is not possible to determine if poor performance is due to poor hyperparameter choices or if the issue is with the network architecture. Additionally, the high computational cost of each sample run can prohibit using the average of multiple runs to evaluate the effect that hyperparameters have

المنارة للاستشارات
www.manaraa.com

on performance.

2.4.2.2 In-sample variance

Figure 2.6 presents the relative length of multiple different runs for the best performing Pointer Network architecture. This figure shows that once the neural network architecture is specialized enough for a given problem and the hyperparameters are tuned reinforcement learning can be used to learn the optimal policy function in combinatorial optimization problems. These results replicate the findings of Bello et al. (2016). This figure also shows that once hyperparameters are tuned there is not a high degree of in-sample variance.

Figure 2.7 shows that training is not always monotonic and that the sample variance observed in the supervised learning can be present in reinforcement learning for some hyperparameter choices. This shows that a single training run of a neural network may not be representative of the potential performance and so multiple runs for each architecture are needed to evaluate performance.

2.5 Conclusion

This paper has shown that the Unidirectional and Bidirectional neural network architectures which were not specialized for the TSP were able to be trained using supervised learning but they were not sufficiently specialized to be trained using reinforcement learning. The Pointer Network whose architecture was chosen specifically to solve the TSP was able to be trained using both supervised and reinforcement learning. This evidence shows that while reinforcement learning is a general methodology that can be applied to a wide variety

المنارة للاستشارات
www.manaraa.com

of problems it is necessary to choose a neural network architecture that is specialized to the specific problem of interest.

The necessity of choosing neural network architectures specific to a problem of interest is problematic due to a high degree of in-sample variance, sensitivity to hyperparameter choice, and that small architecture variations can lead to large performance differences. This paper showed the effect that changes in the learning rate, number of cells in a network, and functional form for the attention function has on the network's performance. Due to the computational requirements of each run, the number of hyperparameters to be optimized, and potential variations in network architecture discovering the settings that yield high performance can be very costly. I showed that when using reinforcement learning there is less feedback than when using supervised learning for the effect that hyperparameters have on performance because the majority of options remained stuck at the performance of a random policy. This suggests that supervised learning may serve as a useful method for the optimization of hyperparameters before reinforcement learning is attempted.

The comparison between an architectures performance when trained using supervised and reinforcement learning also showed that a networks ability to approximate the optimal policy does not guarantee that it will be able to learn the optimal policy using reinforcement learning. There does however appear to be a relationship between the speed and consistency with which a neural network architecture is able to be trained using supervised learning and its ability to be trained using reinforcement learning. An important area of future research will be to study how consistent this relationship is for different problem statements and what the limitations are for using supervised learning as a benchmark for an architecture's

المنارة للاستشارات
www.manaraa.com

performance when trained using reinforcement learning.

Tables

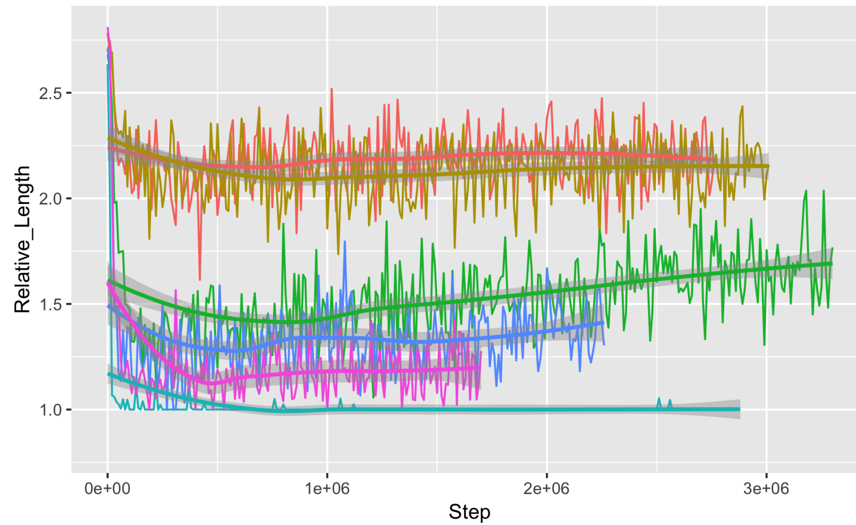

Figure 2.1: Multiple runs of the unidirectional encoder/decoder using supervised learning

Attention=Luong, LR=1e-3, Number of cells=128

Each color represents a run of this method with a different random initialization

www.manaraa.com

Architecture	Attention	# of cells	LR	Relative Length
Unidirectional	Luong	128	1e-2	2.77
Unidirectional	Luong	128	1e-3	1.00
Unidirectional	Luong	128	1e-4	1.99
Unidirectional	Luong	32	1e-3	2.13
Unidirectional	Luong	64	1e-3	1.01
Unidirectional	Bah	128	1e-2	2.76
Unidirectional	Bah	128	1e-3	2.43
Unidirectional	Bah	128	1e-4	2.45
Unidirectional	Bah	32	1e-3	2.66
Unidirectional	Bah	64	1e-3	2.61
Bidirectional	Luong	128	1e-2	2.28
Bidirectional	Luong	128	1e-3	1.00
Bidirectional	Luong	128	1e-4	2.08
Bidirectional	Luong	32	1e-3	1.50
Bidirectional	Luong	64	1e-3	1.01
Bidirectional	Bah	128	1e-2	2.77
Bidirectional	Bah	128	1e-3	1.50
Bidirectional	Bah	128	1e-4	2.17
Bidirectional	Bah	32	1e-3	1.00
Bidirectional	Bah	64	1e-3	2.38
Pointer	Luong	128	1e-2	2.22
Pointer	Luong	128	1e-3	1.12
Pointer	Luong	128	1e-4	1.54
Pointer	Luong	32	1e-3	1.51
Pointer	Luong	64	1e-3	1.57
Pointer	Bah	128	1e-2	1.21
Pointer	Bah	128	1e-3	1.06
Pointer	Bah	128	1e-4	1.07
Pointer	Bah	32	1e-3	1.11
Pointer	Bah	64	1e-3	1.07

Table 2.1: Table of supervised learning experimental results

المنارة للاستشارات
www.manaraa.com

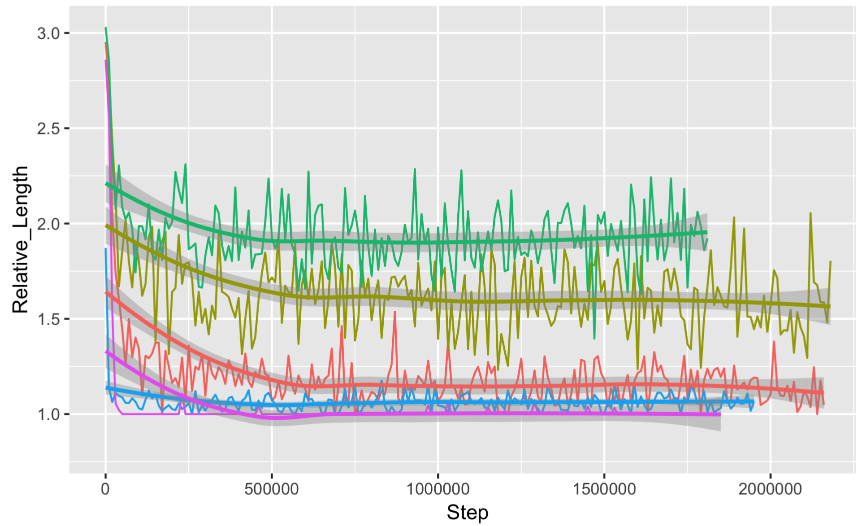

Figure 2.2: Multiple runs of the unidirectional encoder/decoder using supervised learning

Attention=Luong, LR=1e-3, Number of cells=128

Each color represents a run of this method with a different random initialization

www.manaraa.com

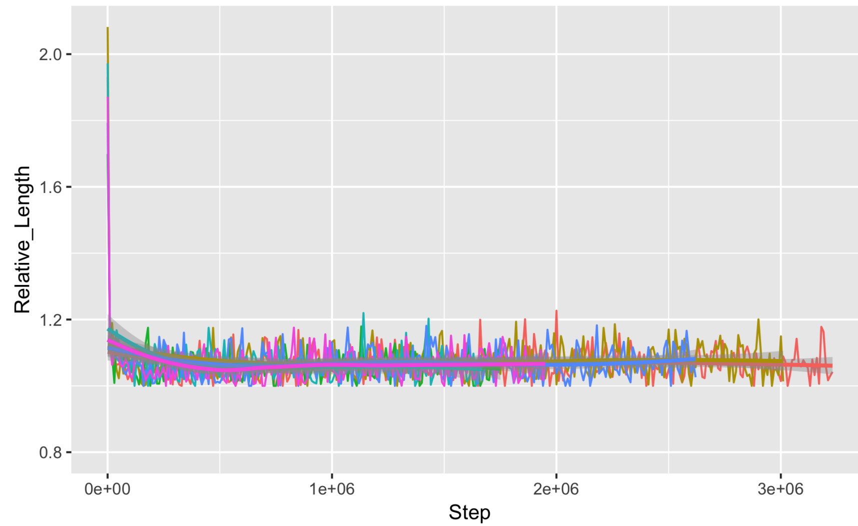

Figure 2.3: Multiple runs of the unidirectional encoder/decoder using supervised learning

Attention=Bah, LR=1e-3, Number of cells=128

Each color represents a run of this method with a different random initialization

المنارة للاستشارات
www.manaraa.com

Architecture	Attention	# of cells	LR	Relative Length
Unidirectional	Luong	128	1e-2	2.75
Unidirectional	Luong	128	1e-3	2.74
Unidirectional	Luong	128	1e-4	2.84
Unidirectional	Luong	32	1e-3	2.64
Unidirectional	Luong	64	1e-3	2.74
Unidirectional	Bah	128	1e-2	2.76
Unidirectional	Bah	128	1e-3	2.73
Unidirectional	Bah	128	1e-4	2.74
Unidirectional	Bah	32	1e-3	2.71
Unidirectional	Bah	64	1e-3	2.74
Bidirectional	Luong	128	1e-2	2.77
Bidirectional	Luong	128	1e-3	2.75
Bidirectional	Luong	128	1e-4	2.71
Bidirectional	Luong	32	1e-3	2.73
Bidirectional	Luong	64	1e-3	2.74
Bidirectional	Bah	128	1e-2	2.69
Bidirectional	Bah	128	1e-3	2.74
Bidirectional	Bah	128	1e-4	2.75
Bidirectional	Bah	32	1e-3	2.74
Bidirectional	Bah	64	1e-3	2.75
Pointer	Luong	128	1e-2	2.64
Pointer	Luong	128	1e-3	1.22
Pointer	Luong	128	1e-4	1.31
Pointer	Luong	32	1e-3	1.27
Pointer	Luong	64	1e-3	1.21
Pointer	Bah	128	1e-2	2.65
Pointer	Bah	128	1e-3	1.03
Pointer	Bah	128	1e-4	1.12
Pointer	Bah	32	1e-3	1.15
Pointer	Bah	64	1e-3	1.04

Table 2.2: Table of reinforcement learning experimental results

المنارة للاستشارات
www.manaraa.com

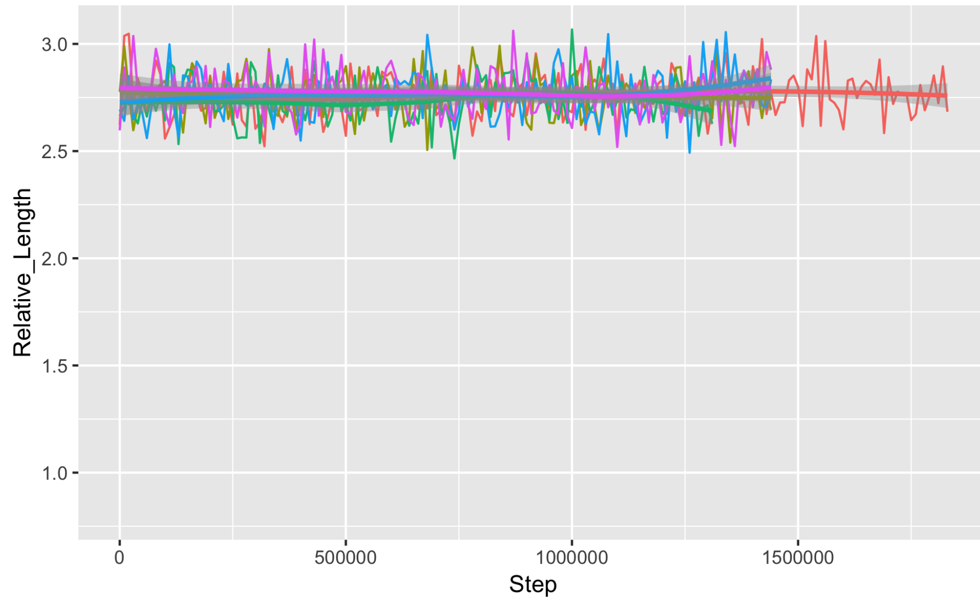

Figure 2.4: Multiple runs of the unidirectional encoder/decoder using reinforcement learning

Attention=Luong, LR=1e-3, Number of cells=128

Each color represents a run of this method with a different random initialization

www.manaraa.com

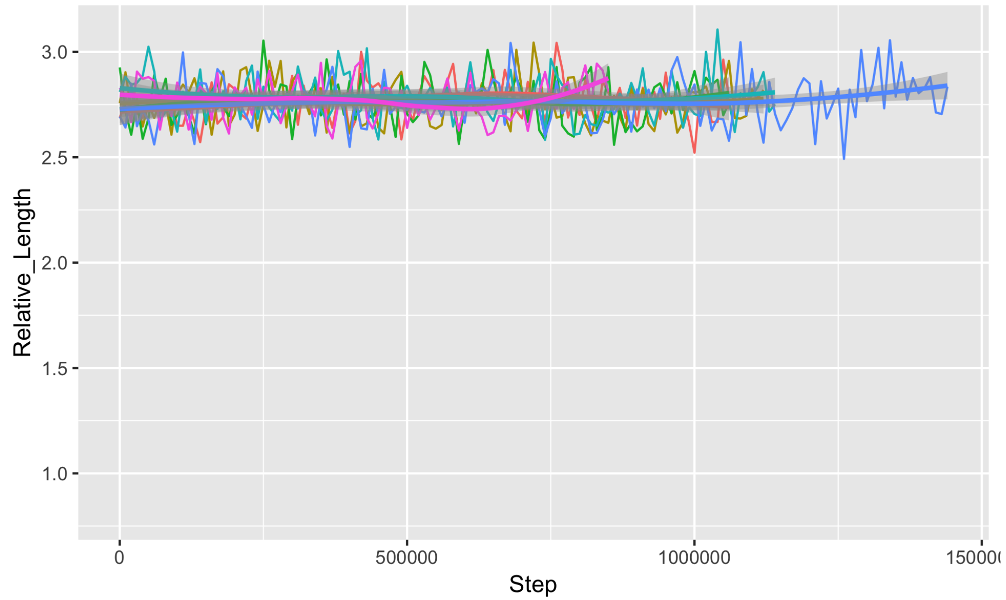

Figure 2.5: Multiple runs of the unidirectional encoder/decoder using reinforcement learning

Attention=Luong, LR=1e-3, Number of cells=128

Each color represents a run of this method with a different random initialization

www.manaraa.com

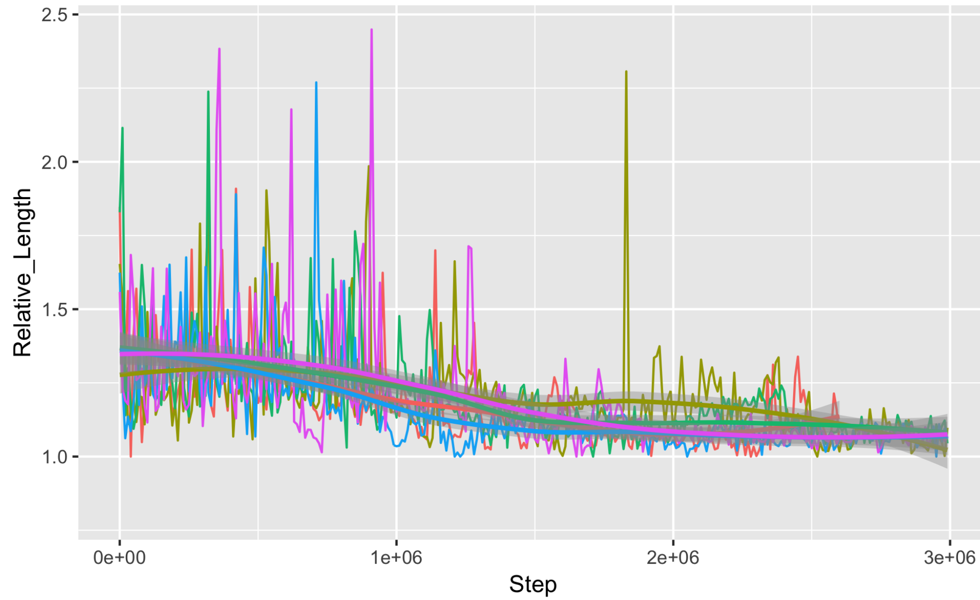

Figure 2.6: Multiple runs of the unidirectional encoder/decoder using reinforcement learning

Attention=Bah, LR=1e-3, Number of cells=128

Each color represents a run of this method with a different random initialization

للاستشارات المنارة

www.manaraa.com

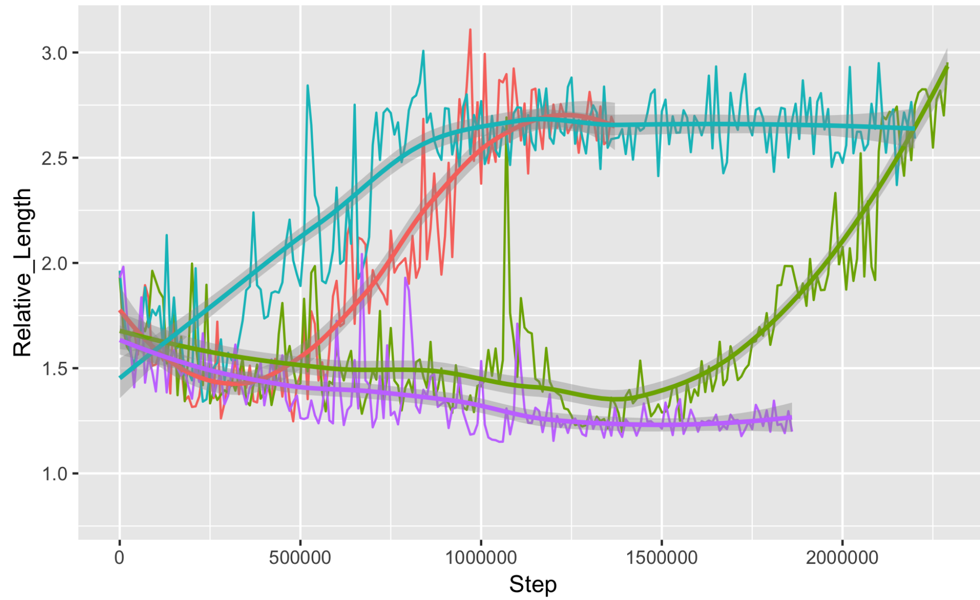

Figure 2.7: Multiple runs of the unidirectional encoder/decoder using reinforcement learning

Attention=Luong, LR=1e-4, Number of cells=128

Each color represents a run of this method with a different random initialization

www.manaraa.com

APPENDIX A
APPENDIX TO CHAPTER 1

A.1 Avoiding the curse of dimensionality in the action space

In order to reduce the computational cost caused by a curse of dimensionality in the action space it is necessary to either reduce the number of decision variables, or reduce the number of function evaluations required to find the policy that maximizes the objective. In the Ericson and Pakes (1995) framework it is common to use a static-dynamic breakdown to reduce the number of decision variables. A static-dynamic breakdown means that the model is constructed in such a way that in the dynamic industry firms to choose the same prices that they would choose in a single period model in equilibrium. If this is the case then the profitability for firm 1 in each state only has to be computed once and then stored in memory. In some settings it is not desirable to use a static-dynamic breakdown since the pricing decision does influence the dynamics of the industry. This is the case in dynamic network industries where the price of a product in period t affects the market share for each firm in t+1 (Chen & Doraszelski, 2006; Chen, Doraszelski, & Harrington Jr, 2009; Mitchell & Skrzypacz, 2006).

If there several decision variables, a high degree of accuracy is required, or if function evaluations of $f(S_i, \mu_{i,1}^k$ are costly then it is advantageous to use more sophisticated methods to find the optimal policy in a state. The first issue that arises is if the decision variable is continuous. Exit and entry decisions are naturally modeled as a discrete 0 for exit and 1 for remaining in the industry, but many other policy decisions, such as how much to invest, or

www.manaraa.com

the price charged for a product are most naturally modeled as a continuous variable. Since in the Ericson and Pakes (1995) framework the state space is discretized in order for $H(S)$ from Equation 1.1 to be well defined it is necessary to map continuous actions to discrete future states. This is because the objective function $f(S,\mu)$ contains the evaluation of the expected value of future payoffs conditional on the current state and actions. In this expectation the value function $V : S' \in \{S_i^N\} \to \mathbb{R}$ can only be evaluated if $S' \in \{S_i^N\}$ where $S' = h(S,\mu)$. The transition function $h(S,\mu)$ maps the policy μ and discrete states S to discrete states S'. In the numerical example used in this paper the transition function represents the change in quality a firm receives as a function of their current quality and level of investment. The transition function is restricted to increasing quality by one level, remaining at the current level, or decreasing quality by one level and the probability of each event is conditional on the level of investment, x.

In order to discuss numerical optimization methods for it is helpful to rewrite the right hand side of the Bellman equation as

$$\begin{aligned} & H(S) = \max_{\mu} f(S,\mu) \\ \text{s.t.} \quad & g(S,\mu) = 0, \\ & h(S,\mu) \geq 0 \end{aligned} \tag{A.1}$$

The most straight forward approach to solving this problem is to use a simple grid search. To do the grid search you choose a grid for each element in the vector μ and then evaluate f at each point in the grid and then choose the action that has the largest value. While this method will find the global maximum of f up to the accuracy limit imposed by

المنارة للاستشارات
www.manaraa.com

how fine the grid is, this method requires far too many function evaluations to be used in most contexts.

The numerical analysis literature has produced many algorithms that have far better convergence rates to the optimal action and there are many efficiently implemented software packages that makes using these algorithms very easy. However, the run time required to find the optimal action varies significantly based on which method was chosen, the problem's structure, and the specific way that the algorithm was implemented in a software package.

The first numerical optimization algorithm I used for solving this problem was the method presented in Wachter and Biegler (2006) and implemented in the C++ library IPOPT. This method is a derivative free algorithm for solving constrained optimization problems and is also one of the most standard computational methods for solving non-linear optimization problems with constraints. It is also the default method for MATLAB's function fmincon. When I used this algorithm for the numerical example, it had convergence issues and the software implementation was not suited to dynamic programming applications. This is because dynamic programming applications require the numerical method to be applied inside a for loop for the value function iterations and inside a for loop that is iterating over each approximation node. This means that the algorithm and software implementation should have limited overhead and not require access to a solver outside of the source code which the IPOPT library did.

The C++ library NLOPT introduced in Johnson (2017) does not have the implementation issues that IPOPT has. This library provides access to many different optimization methods. The numerical methods to solve constrained non-linear optimization problems

المنارة للاستشارات
www.manaraa.com

can be categorized into several categories. The first category, global optimization focuses on finding the global maximum. For example, the DIRECT algorithm presented in Jones, Perttunen, and Stuckman (1993) relies on the systematic division of the search domain into smaller and smaller rectangles. Methods in this category are typically very slow and thus are not tractable for the applications of interest.

The second category is local derivative-free optimization algorithms. I compared NLOPT's implementation of the Constrained Optimization by Linear Approximations (COBYLA) presented in (M. Powell, 1998; M. J. Powell, 1994) to IPOPT's method. While methods in this category are not guaranteed to find the global optimum they are much faster and perform well on convex problems. I found that COBYLA was able to converge however the method required approximately 50 function evaluations per state per value function iteration to find the optimal action within a tolerance of 1e-5.

The third category is local gradient-based optimization algorithms. These methods require the analytical gradient of the optimization problem with respect to the decision variables to be supplied. This gradient is often easy to analytically derive and supply to the algorithm in the applications of interest. These methods have been shown to converge at a much faster rate than derivative-free algorithms (Bradie, 2006). I compared the L-BFGS algorithm presented in (Liu & Nocedal, 1989; Nocedal, 1980) with the COBYLA method and found that L-BFGS required approximately 5.6 function evaluations per state per value function iteration to find the optimal action within a tolerance of 1e-5. Local gradient-based optimization algorithms also do not suffer from the curse of dimensionality in the action space which makes them attractive for applications where there are hundreds of decision

www.manaraa.com

variables and many constraints. This advancement in numerical analysis allows economic researchers to model firms with more realistic and complicated action spaces.

These methods all improve upon the simple grid search algorithm and depending on the context dramatically reduce f_{evals}. The derivative free methods however will still suffer from the curse of dimensionality in the action space that occurs when there are many decision variables. In order to overcome this issue it is recommended to analytically supply the gradient of f with respect to μ to the algorithm. If you supply the analytical gradient then it is possible to use algorithms in the local gradient-based optimization category which have been shown to further reduce f_{evals} and also overcome the curse of dimensionality in the action space.

A.2 Plots

In each plot the top row is the value function, the middle row is the policy function for investment, and the bottom row is the policy function for product price. Each column shows a different view of the 3D graph where the first column shows firm 2's quality increasing on the left and firm 1's quality increasing on the right. The middle column shows a rotation of the first column so that firm 1's quality is increasing to the right and the right column the rotation such that firm 1's quality is decreasing to the right. The top row was generated by evaluating the value function approximation on a grid of 150x150 points, and coloring the points based on their value. The middle and bottom rows cannot be evaluated in this way since the algorithm presented do not interpolate the policy functions. Since it is hard to see the structure of the policy functions when only examining discrete points I interpolated

المنارة للاستشارات
www.manaraa.com

the policy functions using a Delaunay triangulation method from the package deldir in R (Turner, 2017).

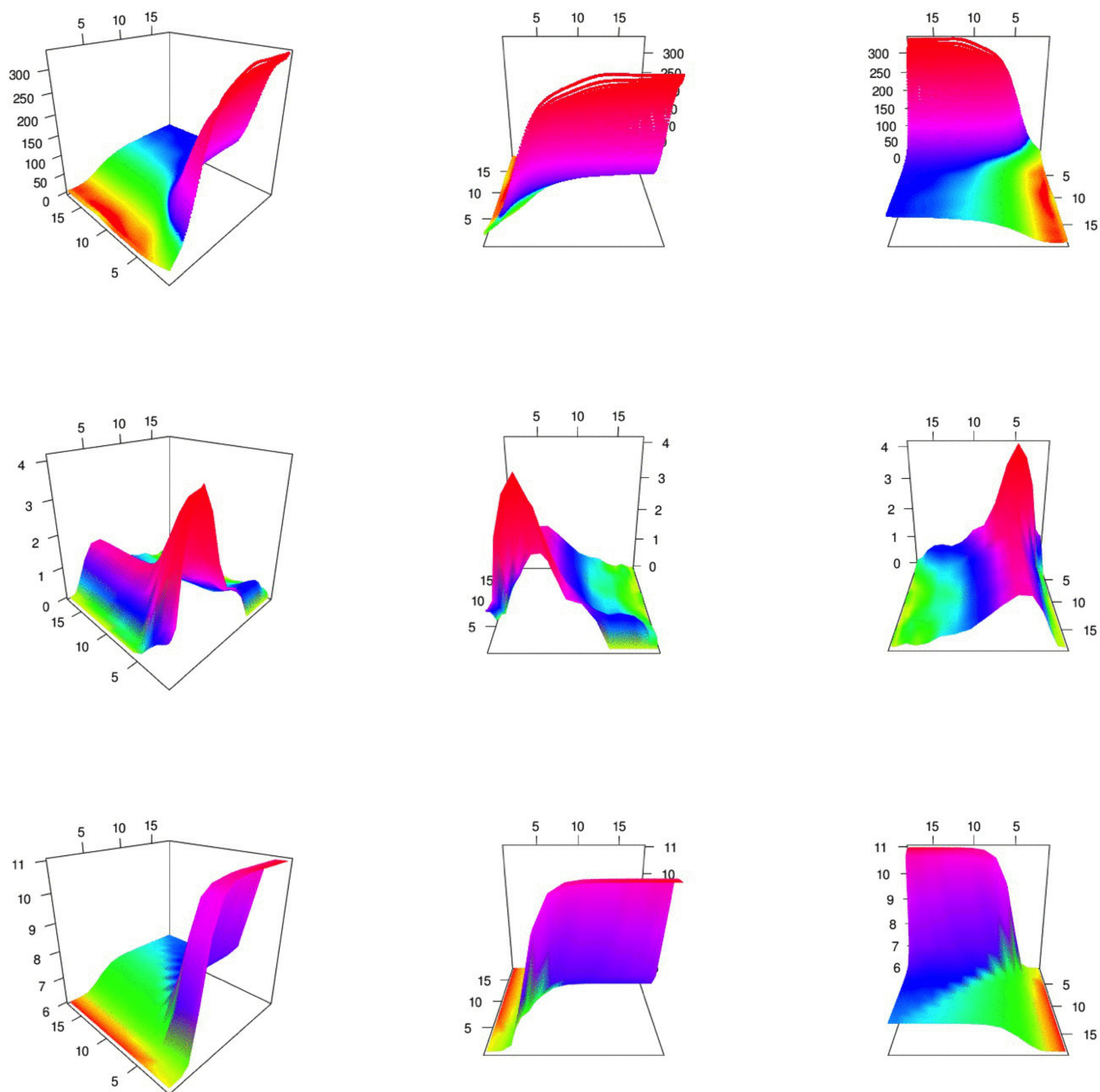

Figure A.1: The equilibrium value, investment policy and pricing policy fit on an 18x18 grid with 12 degree multidimensional Chebyshev polynomials

المنارة للاستشارات
www.manaraa.com

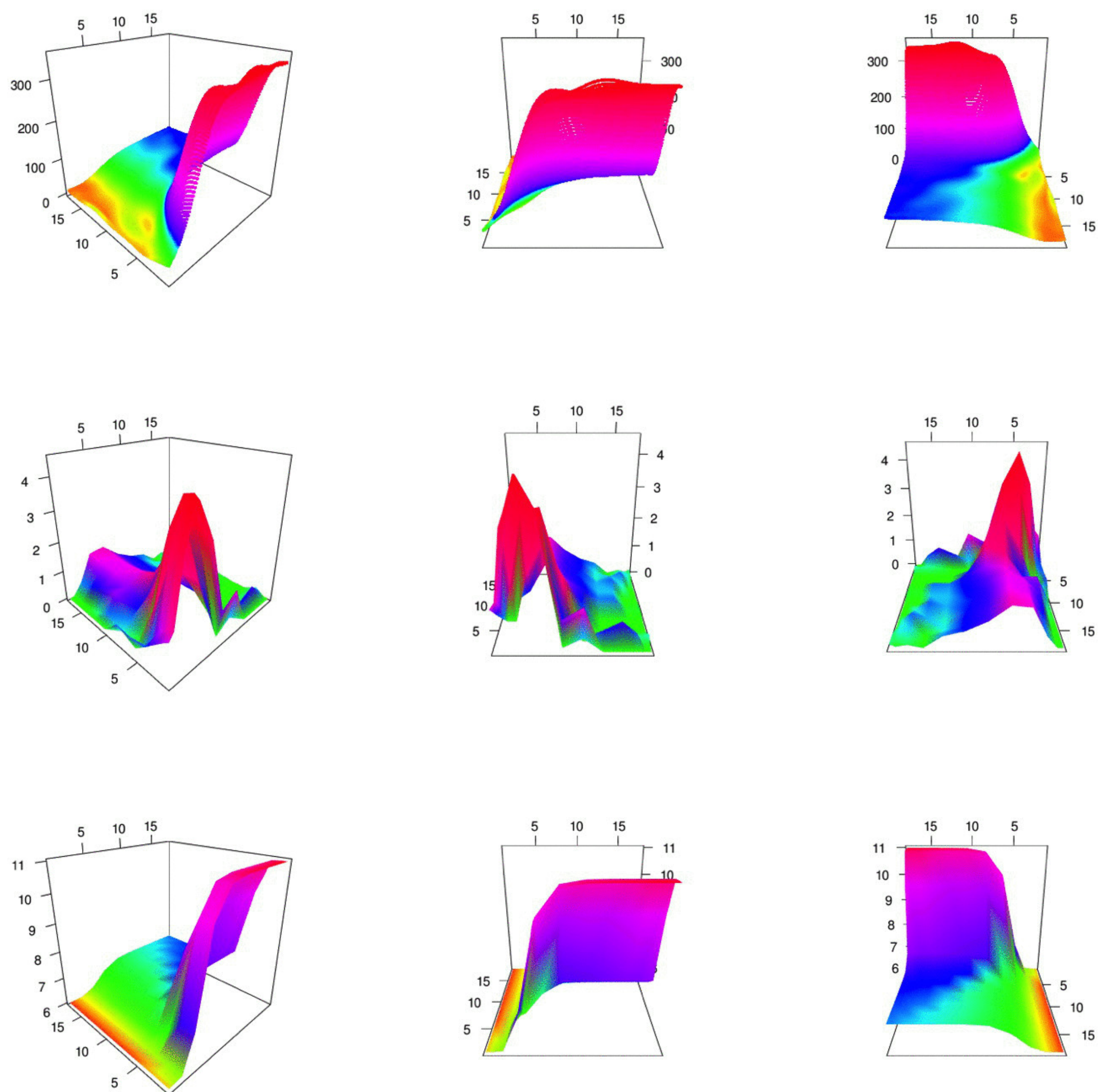

Figure A.2: The equilibrium value, investment policy and pricing policy fit on an 12x12 grid with 12 degree multidimensional Chebyshev polynomials

المنارة للاستشارات
www.manaraa.com

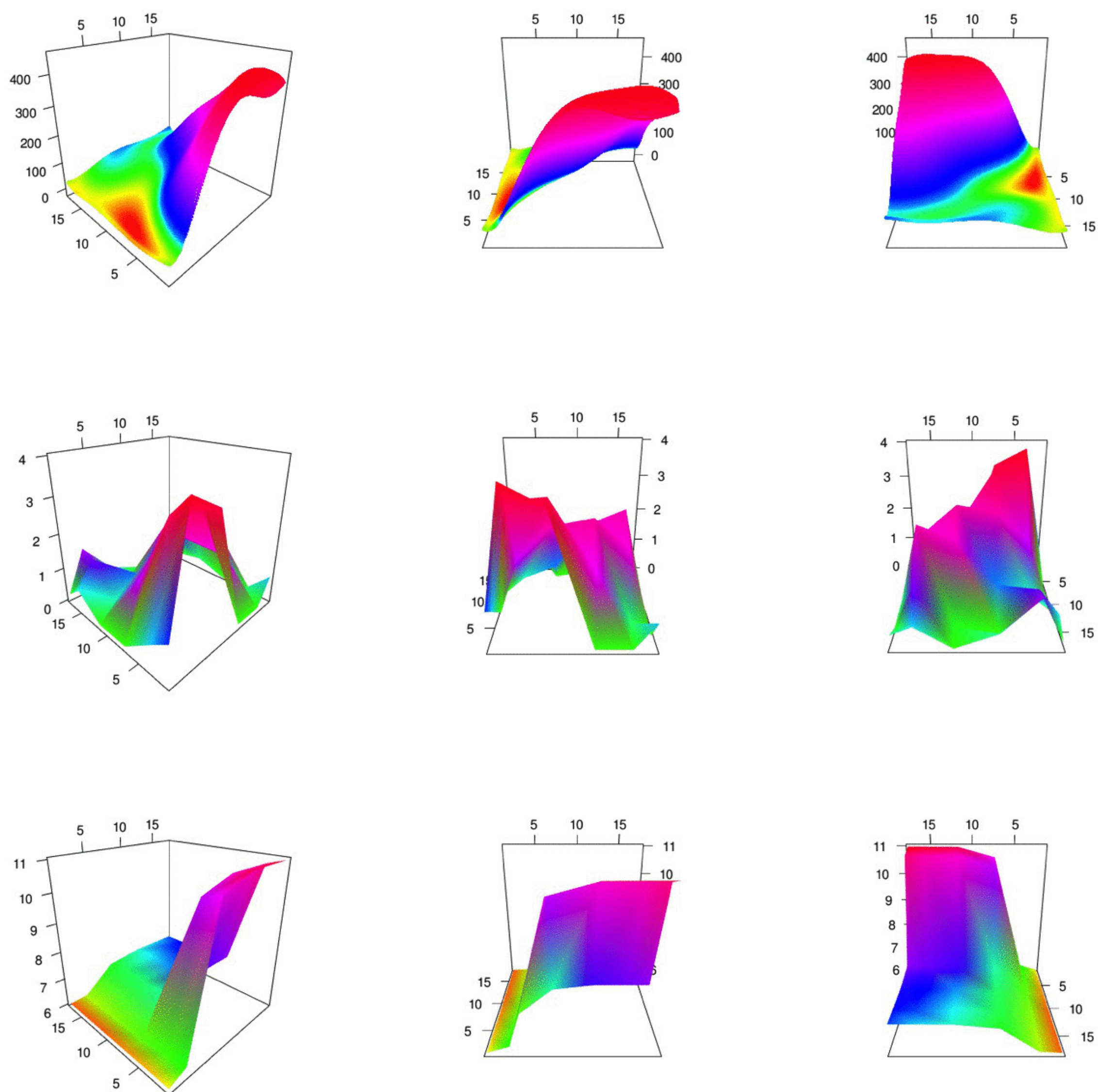

Figure A.3: The equilibrium value, investment policy and pricing policy fit on an 6x6 grid with 6 degree multidimensional Chebyshev polynomials

المنارة للاستشارات
www.manaraa.com

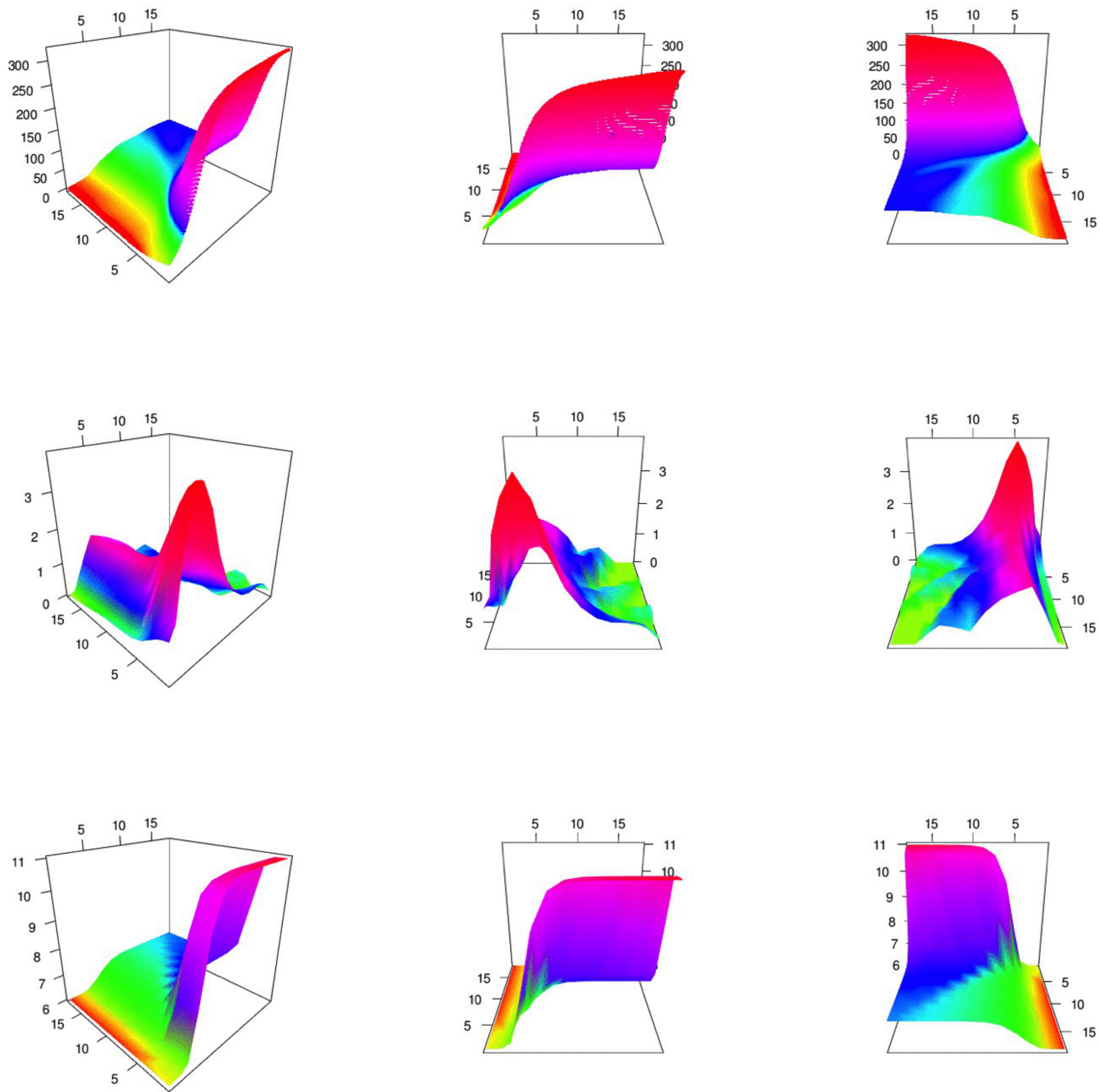

Figure A.4: The equilibrium value, investment policy and pricing policy fit on an 18x18 grid using a neural net with a single fully connected layer with 324 nodes and tanh activation function

المنارة للاستشارات
www.manaraa.com

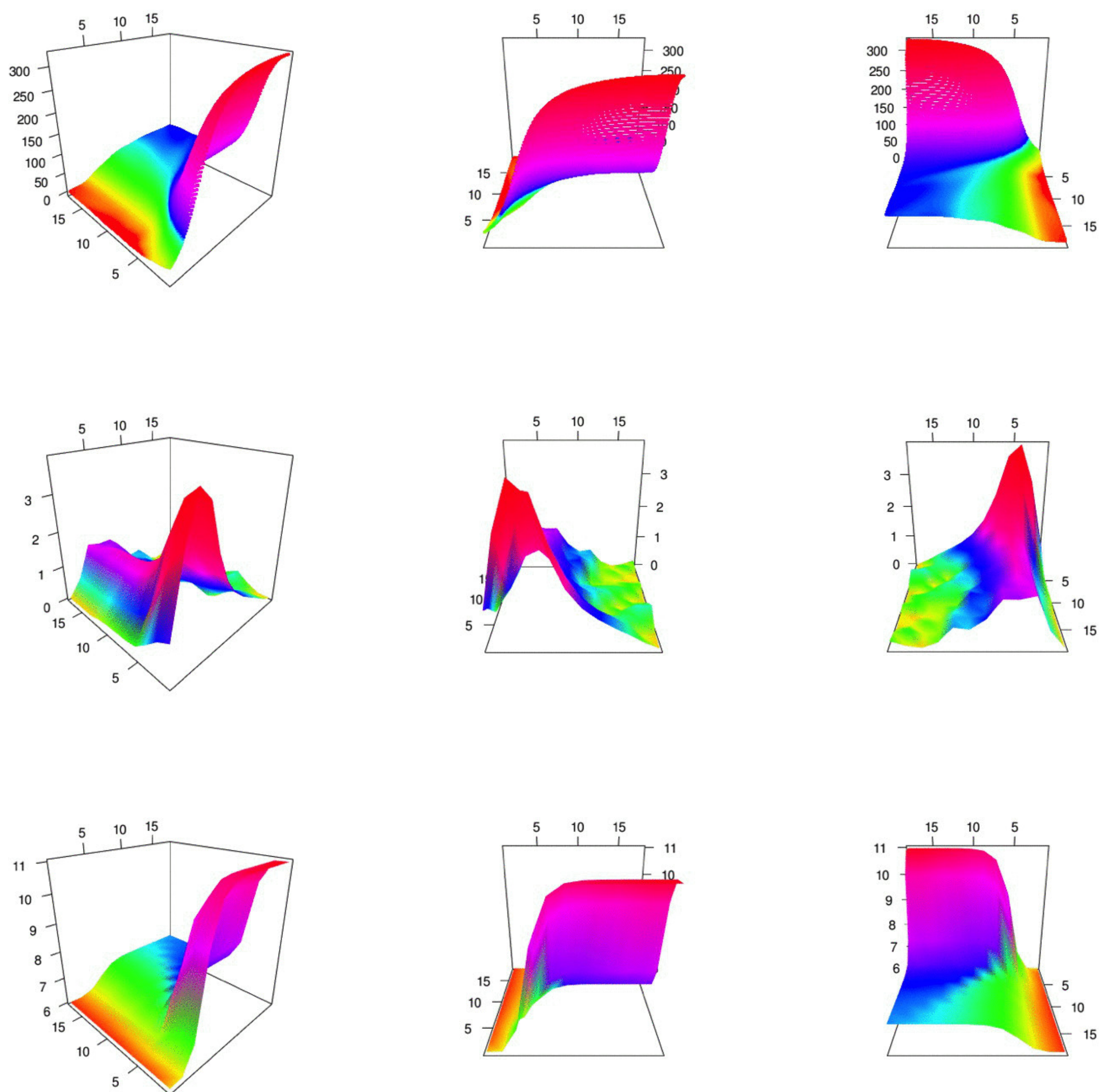

Figure A.5: The equilibrium value, investment policy and pricing policy fit on an 12x12 grid using a neural net with a single fully connected layer with 144 nodes and tanh activation function

المنارة للاستشارات
www.manaraa.com

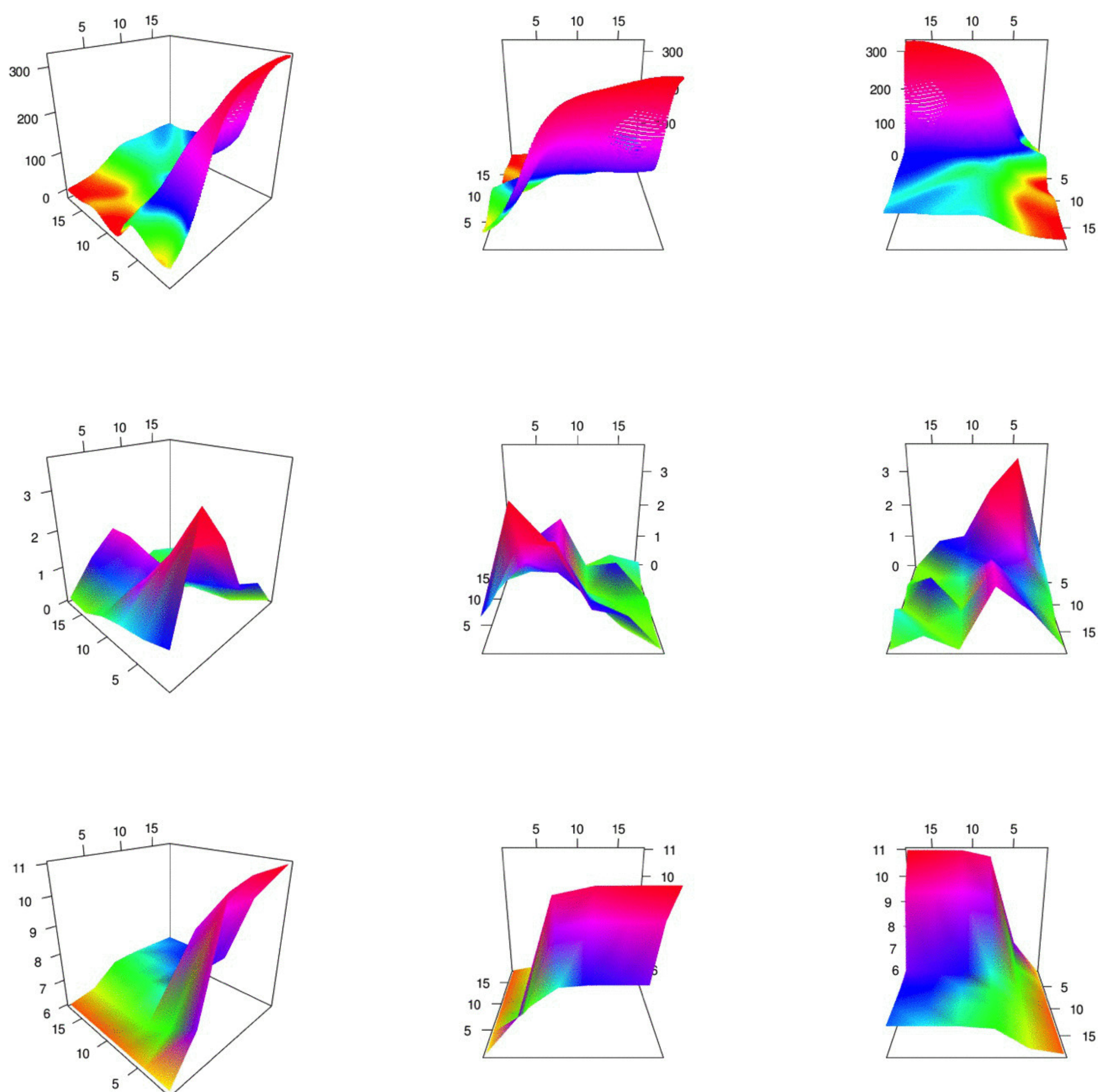

Figure A.6: The equilibrium value, investment policy and pricing policy fit on an 6x6 grid using a neural net with a single fully connected layer with 144 nodes and tanh activation function

المنارة للاستشارات
www.manaraa.com

A.3 Multidimensional complete Chebyshev polynomial approximation

This section is reproduced from Cai and Judd (2015) for convenience. Let the domain for the approximation be

$$\{\boldsymbol{x} = (x_1, \ldots, x_d) : x_{min,j} \leq x_j \leq x_{max,j}, j = 1, \ldots, d\},$$

for some real numbers $x_{min,j}$ and $x_{max,j}$ with $x_{max,j} > x_{min,j}$ for $j = 1, \ldots, d$. Let $\boldsymbol{x}_{min} = (x_{min,1}, \ldots, x_{min,d})$ and $\boldsymbol{x}_{max} = (x_{max,1}, \ldots, x_{max,d})$. Then we denote $[\boldsymbol{x}_{min}, \boldsymbol{x}_{max}]$ as the approximation domain. Let $\alpha = (\alpha_1, \ldots, \alpha_d)$ be a vector of non-negative integers. Let $T_\alpha(\boldsymbol{z})$ denote the product $\prod_{1\leq j\leq d} T_{\alpha_j}(z_j)$ for $\boldsymbol{z} = (z_1, \ldots, z_d) \in [-1, 1]^d$. Let

$$\boldsymbol{Z}(\boldsymbol{x}) = \left(\frac{2x_1 - x_{min,1} - x_{max,1}}{x_{max,1} - x_{min,1}}, \ldots, \frac{2x_1 - x_{min,d} - x_{max,d}}{x_{max,d} - x_{min,d}}\right)$$

for any $\boldsymbol{x} = (x_1, \ldots, x_d) \in [\boldsymbol{x}_{min}, \boldsymbol{x}_{max}]$.

Using this notation, the degree n complete Chebyshev approximation for $V(\boldsymbol{x})$ is

$$\hat{V}_n(\boldsymbol{x}; \boldsymbol{b}) = \sum_{0\leq|\alpha|\leq n} b_\alpha T_\alpha(\boldsymbol{Z}(\boldsymbol{x})),$$

where $|\alpha| = \sum_{j=1}^d \alpha_j$. The number of terms with $0 \leq |\alpha| = \sum_{j=1}^d \alpha_i \leq n$ is $\binom{n+d}{d}$ for the degree n complete Chebyshev approximation in $\mathbb{R}^d$.

المنارة للاستشارات
www.manaraa.com

APPENDIX B
APPENDIX TO CHAPTER 2

B.1 Policy Gradient Method

The policy gradient theorem states that if we let

$$J(\theta) \doteq v_{\pi_\theta}(s_0)$$

denote the true value function for π_θ, the policy determined by θ, then

$$\nabla J(\theta) = \sum_s \mu_\pi(s) \sum_a \nabla \pi_\theta(a|s) r_{\pi_\theta}(s,a)$$

where $r_\pi(s,a)$ is the reward from taking action a in state s, and $\mu_\pi(s)$ is the expected number of time steps t on which $S_t = s$ given s_0 and following π_θ (Sutton, Barto, et al., 1998). This theorem is valuable because we can sample this expected value which is done in the REINFORCE algorithm where

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\beta^t R_t \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)}\Big]$$

where R_t is the observed reward from step t to the end, A_t is the sampled action in the sampled state S_t, and β is the discount parameter (Williams, 1992). Because software implementations of neural networks use Autograd which will automatically analytically compute the gradient of the network with respect to its trainable parameters θ through backpropagation this equation is now easy to compute.

Improvements to this algorithm include the addition of a separate trainable network called a critic network. This network is used to decrease the variance of the policy parameter

المنارة للاستشارات
www.manaraa.com

update. The addition of a critic network changes the gradient calculation to

$$\nabla_\theta J(\theta|s) = \mathbb{E}_{\pi\sim p_\theta(.|s)}\Big[(L(\pi|s) - \hat{v}_\phi(s))\nabla_\theta \log p_\theta(\pi|s)\Big]$$

where $\hat{v}_\phi(s)$ is a neural network parameterized by ϕ.

Algorithm 4 Actor-Critic Method

procedure Given $p_\theta(.|s)$, $\hat{v}_\phi(s)$

Initialize policy parameter θ and state-value parameters ϕ

while Training **do**

Generate an episode $S_0, A_0, r_1, \ldots, S_{T-1}, A_{T-1}, r_T$ following $\pi_\theta(\cdot|\cdot)$.

for $t = 1, \ldots, T$ **do**

$R_t \leftarrow$ return from step t

$\phi \leftarrow \text{ADAM}(\phi, \nabla_\phi \|R_t - \hat{v}_\phi(S_t)\|)$

$\theta \leftarrow \text{ADAM}(\theta, \beta^t(R_t - \hat{v}_\phi(S_t))\nabla_\theta \log(\pi_\theta(A_t|S_t)))$

end for

end while

end procedure

This method can be parallelized in order to more efficiently use multiple CPU threads. This method was introduced in Mnih et al. (2016) and is called Asynchronous Advantage Actor-Critic (A3C) and has been shown to increase the speed for which the policy can be trained and the parallelization has also been shown to help stabilize the parameter updates.

www.manaraa.com

See (Sutton et al., 1998) for a more detailed introduction to the REINFORCE, and Actor-Critic methods and see (Mnih et al., 2016) for a detailed description of A3C.

B.2 Recurrent Neural Networks

A recurrent neural network (RNN) is a class of artificial neural networks where output of a cell is directed back as input in to the cell again. This allows the network to exhibit temporal dynamic behavior for a sequence. Unlike feedforward neural networks RNN's maintain a hidden state h_t that is changed as the network iteratively processes input which allows the network to process sequences.

In this paper I use long short-term memory (LSTM) as the central component of my RNN. LSTM cells have parameterized functions for processing the input x_t, updating the cell's hidden state h_t, and determining the output of the cell y_t (Hochreiter & Schmidhuber, 1997). These functions are called the input gate, forget gate, and output gate respectively. The input gate is a sigmoid layer which decides which elements will be potentially added to the current hidden state as a function of the previous hidden state and the current input. Next the forget gate is composed of a sigmoid layer that decides which components of the cell's hidden state to keep for the next step as a function of the previous cell's hidden state and the current input. Finally, the output gate determines the output of the cell as a function of the input and the new hidden state. The parameters of these functions are then trained to minimize a loss function using a gradient decent optimizer such as ADAM.

Please see http://colah.github.io/posts/2015-08-Understanding-LSTMs/ for an introduction to LSTM Networks.

المنارة للاستشارات
www.manaraa.com

B.3 Additional Architecture and Hyperparameter Choices

Due to the computational requirements of each sample run I excluded from this experimental design some architecture and hyperparameter choices that are of interest for this problem. This appendix lists hyperparamters that were tuned before the experiment was conducted and neural network architectures that were studied but not included in the experimental design.

While the learning rate used in the ADAM optimizer is the most commonly changed hyperparameter for tuning performance, β_1, β_2 and ϵ are also hyperparameters that should be tuned for the problem of interest. The values chosen for these parameters were determined by using the settings recommended by Google's Tensorflow software. I chose to hold fixed the mini-batch size used for training at 10 since I observed the fastest training with this setting after trying larger and smaller batch sizes. I studied the use of an exponential moving average as a critic network however, the networks trained with this critic network consistently became stuck in local minima and thus I did not include this critic network in the experimental design. I also studied the use of a stochastic decoder to replace the greedy decoder and used the same methodology that (Bello et al., 2016) used to vary the degree of exploration with a softmax temperature parameter. I found that this only increased the training time without qualitatively changing the performance and so this architecture change was not included in the experiment. The addition of an embedding layer between the state and input into each LSTM cell using the same methodology that words are embeded in the NMT literature lead to a decrease in performance and thus was not included. The use of the Proximal Policy Objective introduced in (Schulman et al., 2017) did not perform better

المنارة للاستشارات
www.manaraa.com

than the traditional REINFORCE objective and so I did not include this methodology in the experimental design. Grid LSTM cells introduced in (Kalchbrenner, Danihelka, & Graves, 2015) were not able to train using supervised or reinforcement learning and so were excluded from this experimental design. Stack Bidirectional LSTM networks did not yield qualitatively different results from the Bidirectional LSTM network and thus were excluded from this experiment. In order to increase performance once the network was trained I studied using a beam search decoder with various beam widths. This decoder was not able to increase performance due to an inability to preserve the route obtained through greedy decoding as one of the potential routes. I examined the use of a sampling decoder which used evaluated the stochastic decoder multiple times and selected the route with the shortest length. This was able to yield performance improvements but did not qualitatively change the results from the experiment and so I did not include this decoder architecture. I found that the use of clipping the gradient of the neural network did not lead to an improvement on the networks ability to be trained. Deudon et al. (2018) found that the use of Principal Component Analysis to reorder the state was beneficial however, the addition of this preprocessing step made the architectures I studied unable to train successfully.

المنارة للاستشارات
www.manaraa.com

References

Aguirregabiria, V., & Nevo, A. (2013). Recent developments in empirical io: Dynamic demand and dynamic games. In *Advances in economics and econometrics* (Vol. 3). Tenth World Congress of the Econometric Society.

Amir, R. (1996a). Continuous stochastic games of capital accumulation with convex transitions. *Games and Economic Behavior*, *15*(2), 111–131.

Amir, R. (1996b). Strategic intergenerational bequests with stochastic convex production. *Economic Theory*, *8*(2), 367–376.

Amir, R. (1997). A new look at optimal growth under uncertainty. *Journal of Economic Dynamics and Control*, *22*(1), 67–86.

Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., & Kautz, J. (2016). Ga3c: Gpu-based a3c for deep reinforcement learning. *CoRR abs/1611.06256*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

Benkard, C. L., Jeziorski, P., & Weintraub, G. Y. (2015). Oblivious equilibrium for concentrated industries. *The RAND Journal of Economics*, *46*(4), 671–708.

Bradie, B. (2006). *A friendly introduction to numerical analysis: with c and matlab materials on website*. Prentice-Hall.

Cai, Y., & Judd, K. L. (2015). Dynamic programming with hermite approximation. *Mathematical Methods of Operations Research*, *81*(3), 245–267.

المنارة للاستشارات
www.manaraa.com

Chen, J., & Doraszelski, U. (2006). Network effects, compatibility choice, and industry dynamics: The duopoly case.

Chen, J., Doraszelski, U., & Harrington Jr, J. E. (2009). Avoiding market dominance: Product compatibility in markets with network effects. *The RAND Journal of Economics*, *40*(3), 455–485.

Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., & Rousseau, L.-M. (2018). Learning heuristics for the tsp by policy gradient. In *International conference on the integration of constraint programming, artificial intelligence, and operations research* (pp. 170–181).

Doraszelski, U., & Judd, K. (2007). *Dynamic stochastic games with sequential state-to-state transitions* (Tech. Rep.). Working paper, Harvard University, Cambridge.

Doraszelski, U., & Judd, K. L. (2012). Avoiding the curse of dimensionality in dynamic stochastic games. *Quantitative Economics*, *3*(1), 53–93.

Doraszelski, U., & Pakes, A. (2007). A framework for applied dynamic analysis in io. *Handbook of industrial organization*, *3*, 1887–1966.

Doraszelski, U., & Satterthwaite, M. (2010). Computable markov-perfect industry dynamics. *The RAND Journal of Economics*, *41*(2), 215–243.

Ericson, R., & Pakes, A. (1995). Markov-perfect industry dynamics: A framework for empirical work. *The Review of Economic Studies*, *62*(1), 53–82.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., … Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*.

المنارة للاستشارات
www.manaraa.com

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Jain, A. K., Mao, J., & Mohiuddin, K. M. (1996). Artificial neural networks: A tutorial. *Computer*, *29*(3), 31–44.

Johnson, S. G. (2017). The nlopt nonlinear-optimization package. Retrieved from `http://ab-initio.mit.edu/wiki/index.php/NLopt` (NLopt Version 2.4.2)

Jones, D. R., Perttunen, C. D., & Stuckman, B. E. (1993). Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, *79*(1), 157–181.

Kalchbrenner, N., Danihelka, I., & Graves, A. (2015). Grid long short-term memory. *arXiv preprint arXiv:1507.01526*.

Liang, S., & Srikant, R. (2016). Why deep neural networks? *CoRR*, *abs/1610.04161*. Retrieved from `http://arxiv.org/abs/1610.04161`

Liu, D. C., & Nocedal, J. (1989). On the limited memory bfgs method for large scale optimization. *Mathematical programming*, *45*(1), 503–528.

Llanas, B., & Lantarón, S. (2007). Hermite interpolation by neural networks. *Applied Mathematics and Computation*, *191*(2), 429–439.

Llanas, B., Lantarón, S., & Sáinz, F. J. (2008). Constructive approximation of discontinuous functions by neural networks. *Neural Processing Letters*, *27*(3), 209–226.

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

المنارة للاستشارات
www.manaraa.com

Menger, K. (1932). Das botenproblem. *Ergebnisse eines mathematischen kolloquiums*, *2*, 11–12.

Mitchell, M. F., & Skrzypacz, A. (2006). Network externalities and long-run market shares. *Economic Theory*, *29*(3), 621–648.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., … Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928–1937).

Nazari, M., Oroojlooy, A., Snyder, L. V., & Takáč, M. (2018). Deep reinforcement learning for solving the vehicle routing problem. *arXiv preprint arXiv:1802.04240*.

Nocedal, J. (1980). Updating quasi-newton matrices with limited storage. *Mathematics of computation*, *35*(151), 773–782.

Pakes, A., & McGuire, P. (1992). *Computing markov perfect nash equilibria: Numerical implications of a dynamic differentiated product model.* National Bureau of Economic Research Cambridge, Mass., USA.

Pakes, A., & McGuire, P. (2001). Stochastic algorithms, symmetric markov perfect equilibrium, and the 'curse' of dimensionality. *Econometrica*, *69*(5), 1261–1281.

Park, J., & Sandberg, I. W. (1991). Universal approximation using radial-basis-function networks. *Neural computation*, *3*(2), 246–257.

Powell, M. (1998). Direct search algorithms for optimization calculations. *Acta numerica*, *7*, 287–336.

المنارة للاستشارات

www.manaraa.com

Powell, M. J. (1994). A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis* (pp. 51–67). Springer.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... others (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (pp. 3104–3112).

Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement learning: An introduction*. MIT press.

Turner, R. (2017). deldir: Delaunay triangulation and dirichlet (voronoi) tessellation [Computer software manual]. Retrieved from `https://CRAN.R-project.org/package=deldir` (R package version 0.1-14)

Ulmer, M. W., Mattfeld, D. C., & Köster, F. (2017). Budgeting time for dynamic vehicle routing with stochastic customer requests. *Transportation Science*.

Vinyals, O., Fortunato, M., & Jaitly, N. (2015). Pointer networks. In *Advances in neural information processing systems* (pp. 2692–2700).

المنارة للاستشارات
www.manaraa.com

Wachter, A., & Biegler, L. T. (2006). On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, *106*(1), 25–57.

Weintraub, G. Y., Benkard, C. L., & Van Roy, B. (2008). Markov perfect industry dynamics with many firms. *Econometrica*, *76*(6), 1375–1411.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3-4), 229–256.

Zainuddin, Z., & Pauline, O. (2008). Function approximation using artificial neural networks. *WSEAS Transactions on Mathematics*, *7*(6), 333–338.

المنارة للاستشارات

www.manaraa.com