IEOR E4742 Deep Learning

Pricing American Options via Feedforward Neural Network

Authors:

Kaiqi Wu, kw2791@columbia.edu

Qing Ye, qy2209@columbia.edu

Tianle Zhao, tz2384@columbia.edu

Department of IEOR, Columbia University

December 2019

Contents

1. Executive Summary

The topic of our project is American option pricing using feedforward neural network. We attempt to build an architecture with a relatively small numbers of layers and neurons, therefore making the neural network shallow and reducing computational cost. We trained our feedforward neural network under different combinations of layers, neurons and activation functions in both local machine (12 cores) and google cloud VM (96 cores). It turns out that the architecture with 2 layers of 20 neurons and relu & elu activation functions yields the best result after training 500,000 sample data, with an out-sample R^2 of 98.09%.

In order to find the best architecture, we had to run many different experiments, and for that we built a system to help us keep track. This system also allows us to apply parallel computing into both data generation and network training. This helped us a great deal as we were running thousands of experiments each using up to 500,000 samples. The entire computation time is shortened down to around three hours while it could have easily taken weeks to run without the parallel computing.

Keywords:

Option Pricing, Feedforward Neural Network, Whaley Approximation, Ju-Zhong Approximation, Quasi-Sampling, Parallel Computing

2. Introduction

American option is a type of options that allows holders to exercise it at any time before and including the day of expiration. Compared to European option, it is more likely for American option holder to make profits, even if price movement may not be accurately predicted. Because of the early exercise advantage, the value of an American option (put) can be expressed as

$$\mathrm{V}(S_t, \mathrm{t}; \mathrm{K}, \mathrm{T}) = \sup_{t \le \tau \le T} E_t[e^{-r(T-t)}(K - S_\tau)^+]$$

It has been proved that for each time t, there exists a corresponding stock price $S^*(t)$ such that if $\mathrm{S}(t) > S^*(t)$ the value exceeds this immediate exercise value. While if $\mathrm{S}(t) \le S^*(t)$, the value of the American put option is the same as its immediate exercise payoff: $\mathrm{K} - \mathrm{S}(\mathrm{t})$. This is the beauty of American Options. The curve $S^*(t)$ is called exercise boundary and the region $\mathrm{C} = \{ (\mathrm{S}, \mathrm{t}) \mid \mathrm{S} > S^*(t) \}$ is called continuation region while the complement of it is called exercise region.

This is the core of pricing American options. As the value of American option should be equal to European option plus the early exercise premium, it is very important to estimate the early exercise boundary.

As introduced in the book *"Computational Methods in Finance"* written by Professor Hirsa, there are several techniques that can be utilized for pricing American options[1], such as Bermudan approximation, Black–Scholes PDE with a synthetic dividend process, and Brennan–Schwartz algorithm. Some of these techniques have been discussed and explored in detail in the Monte Carlo Simulation class. While the accuracies of these techniques are decent, they are relatively time-consuming.

Thus, we want to try a novel technique for this problem: feedforward neural networks. Basically, we want to build a shallow neural network in the context of option pricing to achieve speed-ups while maintaining very high accuracy.

3. Data

In data generating process, the first step is to set up a reasonable range for the market parameters, which will be kept consistent throughout the whole experiment. We choose the following common parameters: underlying stock price (S_0), strike price (K), maturity (T), interest rate (r), and dividend yield (q). As the option price is homogeneous of degree one in S_0 and K, we therefore use $\frac{S_0}{K}$ instead of two separate parameters S_0 and K.

[1] I highly recommend the book "Computational Method in Finance" written by Ali Hirsa. The Chapter 4 provides the readers with a high-level overview of pricing American Options and the detailed explanations are introduced in later chapters

We also include the European option price, which can be easily obtained from Black-Scholes formula, as part of our input data. In our experiments, we want to explore whether the European option price captures any information on the early exercise premium.

In our experiments, our label is generated in the following ranges:

Parameters	Range
Moneyness $\frac{S_0}{K}$	0.8 – 1.2
Maturity (T)	1/12 – 1 year
Dividend rate (q)	0 – 0.03
Risk free rate (r)	0.01 – 0.03

Table 1 Market Parameters and their Ranges

The labels of the training data for our experiments are generated from both Ju-Zhong Approximation and Whaley Approximation. The main difference between these two approximation schemes are: Whaley Approximation works better for options with shorter maturity while Ju-Zhong Approximation performs better for options with longer maturity. In our experiments, we choose to use Ju-Zhong Approximation when the maturity of an option is greater than $\frac{1}{2}$ and turns to Whaley in the other case.

The size of the training data in our experiments ranges from 2,000 to 500,000. We expect the performance of the feedforward neural networks to improve as the size of the training data increases.

4. Methodology

4.1 Quasi-Sampling

Though Quasi-Sampling has to do with data generating process, it is worth discussing in this section separately.

As opposed to the Non-Quasi-Sampling technique where the market parameters are centered at random and uniformly over the given ranges, Quasi-Sampling is a systematic way of sampling. The training data are generated using a random starting point and a fixed, periodic interval. Here Halton sequences are applied to generate points in space. Although these sequences are "systematic", they are of low discrepancy.

4.2 Feedforward Neural Network

Feedforward neural network is a type of neural network where the connections between nodes do not form a cycle. It consists of the following components: an input layer (x), some hidden layers, an output layer ($\hat{y}$), a set of weights and biases (W and b) between each layer, and an activation function for each hidden layer.

A feedforward neural network maps a given input to an output. Graphically, the following architecture is an example of feedforward neural network.

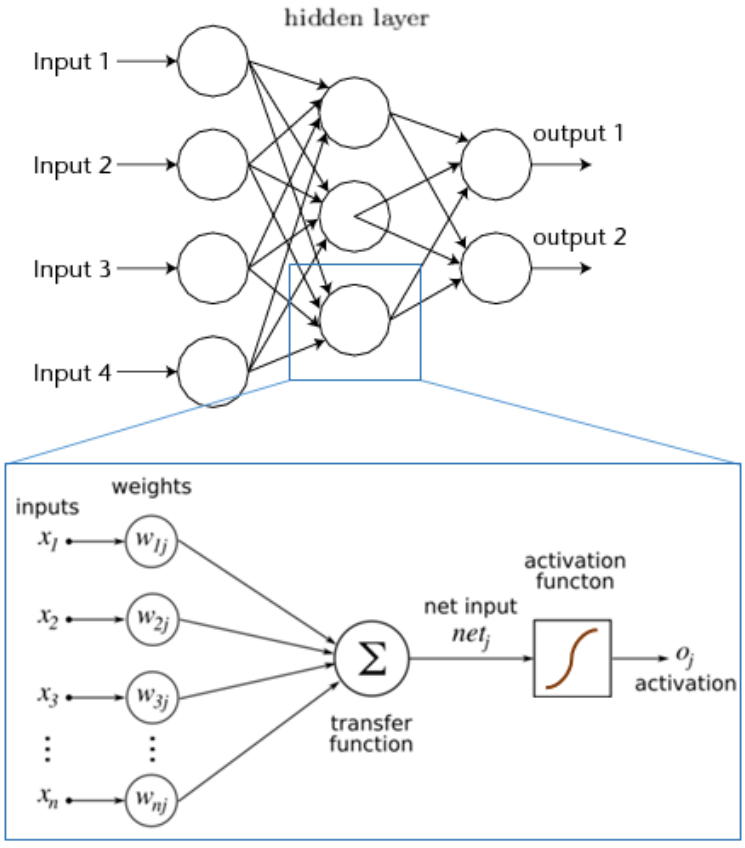

Figure 1 An Instance of Feedforward Neural Network

As is seen from the graph above, the feedforward neural network is a chain of several functions which defines a mapping. It needs to learn the appropriate parameter sets of the model that fits the training data well. This leads to an optimization problem, for which we use mean square error as the cost function.

5. Network Architectures

After we have fixed the data and methodology, we need to pick an architecture for the neural network. In our experiments, we train different types of models with different number of layers, neurons, and use different combinations of activation functions. However, since we want to build a shallow feedforward neural network, we set the number of neurons to be no more than 20 and the number of layers to be no more than 2. Specifically, in our experiments, we chose the number of neurons in each layer to be 5, 10, 20 respectively and the number of layers to be 1 and 2.

In terms of activation functions, we consider the following: Tanh, Sigmoid, Relu, Elu and LeakyRelu. Via various combinations of activation functions, non-linearity has been introduced to our model.

6. Implementation

We implemented a comprehensive system to help us keep track of all the different experiments we wanted to perform. The system is mainly split into three parts, namely “data generation”, “neural networks”, and “experiments”.

In “data generation”, we built a system to manage the variety of datasets to be used. These datasets vary in terms of how the raw parameters are generated, which pricers are used for pricing European and American options, and the input output specifications (such as fixing $k = 1$, and whether to use the European price as part of inputs). We separated input and output generation. This way, we can generate inputs on a Windows computer using Windows-only libraries (such as Ghalton), then move the inputs onto Google cloud machines and generate the outputs using multiprocessing on a 96 core CPU.

In “neural networks”, we built a user-friendly system that can instantiate different neural networks based on the architecture given by the user. The user can specify how many neurons are in each layer, which activation function is used by each layer, and the dropout rate for each layer. We save the model for every epoch during training. This way we can compute different metrics at each epoch later. This also allows us to pick up and continue training the model in the future if necessary.

In “experiments”, we built a system to create, execute, and keep track of experiments, each of which involves a given dataset, model, and training size. The experiments files were made so that they can run in parallel.

We then made several python notebooks that uses this system. ‘GenerateData_inputs.ipynb’ and ‘GenerateData_outputs.ipynb’ are used to generate the datasets. A few sets of experiments were performed, and each was carried out with ‘Add_Experiments_setXXX.ipynb’, ‘Run_Experiments_setXXX.ipynb’ and ‘Analyse_setXXX.ipynb’.

7. Results

7.1. Assessing the Effects of Input Size and Drop-out Rate

In order to evaluate the effects of input size and drop-out rate, we selected the dataset (named as Set 1), for which the inputs were generated from Halton sequence, and the labels were generated using both Juzhong and Whaley approximation. We also fixed K = 1 and included European Options in the inputs.

We first fixed the size of training data to be 25,000 and set the drop-out rate to be 0.25. The table below shows the best architecture for each scenario:

Neurons \ Layers	**1**	**2**
5	72.18% (relu)	79.18% (elu, relu)
10	71.26% (relu)	86.57% (elu, relu)
20	81.44% (relu)	94.10% (relu, leaky_relu)

Table 2 (25,000 samples and 0.25 drop-out rate)

Because we are building a shallow neural network, and the possibility of overfitting such a model is small. It might be helpful if we reduce the drop-out rate. We used the best architecture in the previous experiment to assess the effect of drop-out rate. By changing different drop-out rates, we obtain the table below:

Drop-out Rates	$\mathbf{R^2}$
0	97.12%
0.05	96.54%
0.1	94.55%
0.15	94.91%
0.2	92.28%
0.25	93.21%
0.3	92.83%

Table 3 (25,000 samples and various drop-out rates, with 2 X 20 architecture (relu, leaky_relu))

As shown in the table above, the model performs best when the drop-out rate is 0. This suggests that increasing drop-out rate make our models become worse. we therefore try to rerun our model again but with drop-out rate to be 0 this time. The table below summaries the results.

Neurons \ Layers	1	2
5	70.57% (tanh)	84.52% (tanh,relu)
10	75.56% (tanh)	92.11% (relu, leaky_relu)
20	82.18% (relu)	95.18% (tanh, leaky_relu)

Table 4 (25,000 samples and 0 drop-out rate)

It is verified from this table that our model performs slightly better while keeping the drop-out rate to be 0.

Furthermore, it is natural to think that increasing sample size may improve model performance. Our next step is to fix sample size to be 500,000 while keeping drop-out rate to be 0. And we obtain the following table:

Neurons \ Layers	1	2
5	76.52% (relu)	85.95% (tanh, leaky_relu)
10	80.90% (relu)	93.24% (relu, leaky_relu)
20	85.62% (relu)	98.09% (relu, elu)

Table 5 (500,000 samples and 0 drop-out rate)

From this table, we figure out that the architecture with 2 layers and 20 neurons yields the best result, with an out-sample R^2 98.09%.

We also plot the graph in which we assess the model performance with respect to the number of epochs.

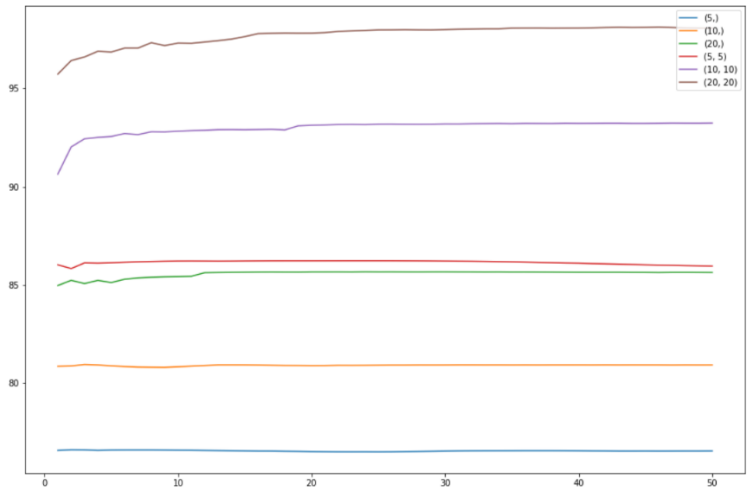

Figure 2 Model Performance versus Number of Epochs

It can be seen from the graph above that the performance of our model becomes stable after the tenth epoch.

Lastly, we plot the "True versus Predicted Labels" graph for the best model (2 layers and 20 neurons with relu and elu activation functions) using out-of-sample data.

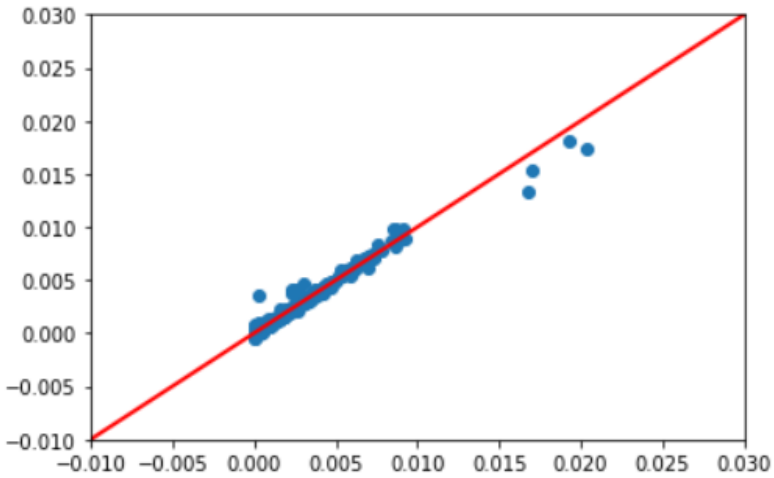

Figure 3 True versus Predicted Labels ($R^2 = 98.09\%$)

7.2. Assessing the Effects of Using the Halton Quasi-Monte Carlo Generator

Previously, we performed experiments mainly on the dataset named as Set 1, which was generated with Halton Quasi-Monte Carlo sequence, Juzhong and Whaley approximation, with a fixed K = 1 and including European Options prices in inputs. However, we also want to try using uniform data generator in our experiments.

We selected the dataset (named as Set 2), which was generated by Uniform Random generator, while keeping other attributes the same. Based on the above research, we set dropout rate as 0, and trained on 25,000 data for different architectures.

For the architectures with different combinations of number of layers and number of neurons per layer, we selected the best choices of activation functions, which achieved the highest R^2. The best architectures using Set 2 is shown as following:

Neurons \ Layers	**1**	**2**
5	70.96% (tanh)	85.67% (tanh, relu)
10	76.51% (tanh)	92.81% (relu, leaky)
20	83.51% (relu)	96.39% (elu, relu)

Table 6 (25,000 Samples, Uniform Data Generator)

The best R^2 = 96.39% is achieved by the 2 layers & 20 neurons/layer, with activation functions elu and relu respectively. This performance is a little bit better than the best result of R^2 = 95.18% from using Set 1 as training data, which may seem inconsistent with the idea that using Halton generator could improve the performance. However, if we check the values of the approximated premiums (labels), taking 1000 samples for simplicity, it can be inferred that the distribution of generated premiums with Uniform random generator is more concentrated, which means Halton generator can approximate a wider range of scenarios. In light of this, Halton quasi-Monte Carlo sequence may be a more reliable generator.

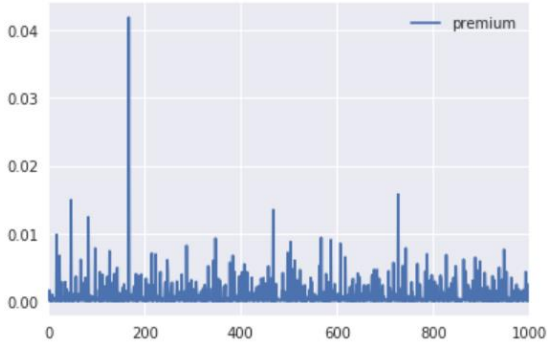

Figure 4 1000 Premiums Generated from Set 1

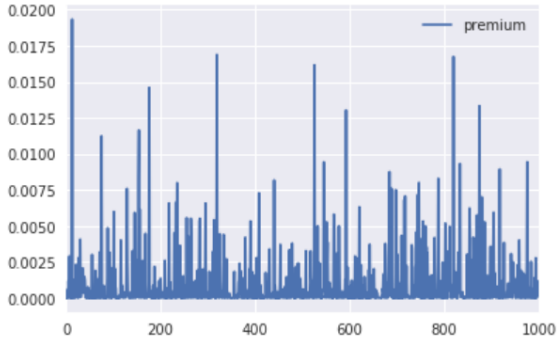

Figure 5 1000 Premiums Generated from Set 2

7.3. Assessing the Effects of Including the European Option Prices in Inputs

To assess the effects of using European Option prices, we selected the dataset (named as Set 3), where we fixed K = 1 but did not include European option prices into inputs, while keeping other attributes as the same. Similarly, we set dropout rate as 0, and trained on 25,000 data for different architectures.

For the architectures with different combinations of number of layers and number of neurons per layer, we selected the best choices of activation functions, which achieved the highest R^2. The best architectures using Set 3 is shown as following:

Neurons \ Layers	1	2
5	74.31% (tanh)	88.43% (relu, leaky_relu)
10	81.71% (relu)	94.12% (relu, elu)
20	85.79% (relu)	97.38% (relu, tanh)

Table 7 (25,000 Samples, Excluding European Option Price from Input)

The best R^2 = 97.38% was achieved by the 2 layers & 20 neurons/layer, with activation functions relu and tanh respectively. This performance is a little bit better than the best result of R^2 = 95.18% using Set 1 as training data, which seems inconsistent with the idea that including European prices in inputs could improve the performance. An educated guess for this phenomenon could be that the original 4 inputs (moneyness $\frac{S_0}{K}$, maturity (T), interest rate (r), and dividend yield (q)) are already capturing the information provided by the European option price, thus including the European option price into inputs may be only adding more noises to the model. Since there exists no clear trend between the European option prices and early-exercise premiums (demonstrated by the following graph), it is possible that their relationship is not captured by shallow neural networks.

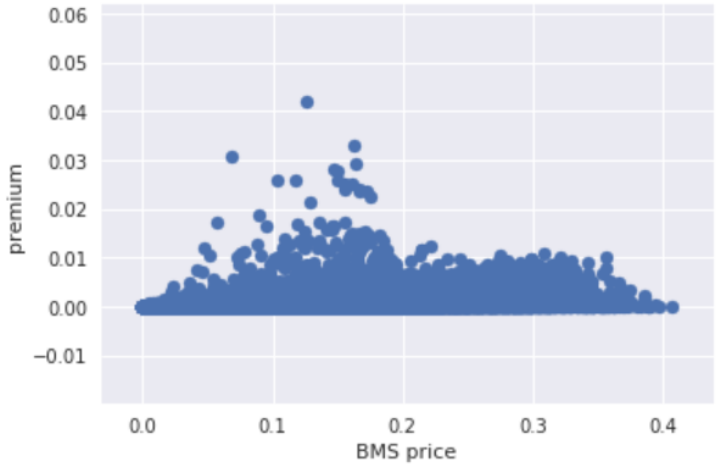

Figure 6 European Option Price versus Early Exercise Premium

8. Extension

This project can be extended in the following ways. Firstly, we could add more experiments to test various other behaviors such as assessing whether using both S_0 and K as parameters instead of moneyness $\frac{S_0}{K}$ would impact the results. We could also generate option prices under VGSA or Merton-Jump processes or retrieve data directly from the market. We could also test out more neuron architectures. We could also study the exercise boundary problem and see if that gives us intuitions on what other features we could include in our model.

Acknowledgement

This project was completed as part of the coursework for IEOR4742 (Deep Learning) at Columbia University, taught by Prof. Ali Hirsa. The main topic of this project was inspired by course materials covered during the semester.

We thank Tugce Karatas who guided us through Ju-Zhong Approximation and Whaley Approximation in her free time, which was tremendously helpful for our project. Moreover, we would like to appreciate our TA, Xiao Xu, for his wonderful assistance in office hours throughout this semester. Finally, we would like to sincerely express our appreciation for Prof. Ali Hirsa, from whose course we have witnessed the beauty of deep learning.

It is a pleasure to thank all these people, and it is a pleasure to conclude our master's program in financial engineering with this unforgettable project.

Reference

[1] Hirsa, Ali (2016). Computational Methods in Finance, CRC, Boca Raton, FL.

[2] Hirsa, Ali & Karatas, Tugce & Oskoui, Amir. (2019). Supervised Deep Neural Networks (DNNs) for Pricing/Calibration of Vanilla/Exotic Options Under Various Different Processes.