Machine Learning in Finance: The Case of Deep Learning for Option Pricing

Robert Culkin & Sanjiv R. Das
Santa Clara University*

August 2, 2017

Abstract

Modern advancements in mathematical analysis, computational hardware and software, and availability of big data have made possible commoditized machines that can learn to operate as investment managers, financial analysts, and traders. We briefly survey how and why AI and deep learning can influence the field of Finance in a very general way. Revisiting original work from the 1990s, we summarize a framework within which machine learning may be used for finance, with specific application to option pricing. We train a fully-connected feed-forward deep learning neural network to reproduce the Black and Scholes (1973) option pricing formula to a high degree of accuracy. We also offer a brief introduction to neural networks and some detail on the various choices of hyper-parameters that make the model as accurate as possible. This exercise suggests that deep learning nets may be used to learn option pricing models from the markets, and could be trained to mimic option pricing traders who specialize in a single stock or index.

1 Artificial Intelligence: A Reincarnation

Artificial intelligence (AI) is in its second new age. While the notion that machines are capable of exhibiting human levels of intelligence saw its beginning implementations in the 1950s, success in creating AI machines was thin, and mostly, AI was deemed to be a failed enterprise. In the past few years, there has been a resurgence of AI,

*Robbie Culkin is an undergraduate Computer Science and Engineering student, and Sanjiv Das is Professor of Finance and Data Science. They may be reached at `rculkin@scu.edu`, `srdas@scu.edu`.

and this article revisits one such application with a view to understand how it may be used in Finance.

AI today mostly revolves around implementing machine learning on very large neural networks (NNs) with many layers. These NNs aim to mathematically mimic the way the human brain works, i.e., by taking in wide-ranging stimuli, and then parsing it through banks (layers) of neurons that learn to associate the input with output, by experience. For example, a child learns that touching a hot plate results in pain, and quickly learns not to go near one. With sufficient data, we can train a NN to learn the best relationship between inputs and outputs, and then it can be set to perform such a task repeatedly. The input and output sets may be extremely large, as in the case of self-driving cars. But large-scale NNs have been found to be rather adept at associating big data with multitudinous outcomes, through training. In fact, machines may be trained a lot faster than humans, as in the case of chess playing computers that can play millions of games with each other over a weekend, and learn from them. No human can accumulate experience as fast.

The huge success of AI has arisen from the confluence of three factors. First, the mathematics of NNs supports very fast calibration (training). Since NNs are calibrated by minimizing some loss function, optimization of this function is needed, and the bigger the NN, the larger the number of parameters that need to be fit. For large NNs, also known as "deep neural nets", there may be millions of parameters. To find these, taking numerical gradients for optimization is computationally infeasible, and luckily, we are able to do so analytically with linear computation, using the backpropagation algorithm (Kelley (1960); Bryson (1961); Dreyfus (1973); Rumelhart et al. (1986)). This algorithm is pleasing and succinct, and without it, so called "Deep Learning" (DL) would be impossible. In short, learning is an optimization problem, and large-scale learning is much more facile when undertaken analytically, rather than numerically.

Second, DL refers to the number of layers in the NN, often as high as 40 layers. Decades ago neural nets were only able to handle 1 or 2 layers. The ability to compute large NNs is possible today because of rapid increases in computational speed. Even then, today's CPUs may not offer ideal improvements in speed. The nature of mathematical calculations in NNs involves matrix calculations in many dimensions, i.e., using tensors (matrices of dimension greater than 2, usually), and such calculations are faster when undertaken on graphics processing units (GPUs), which are now widely used. Indeed, even these are being surpassed by special purpose computer chips, such as Google's tensor processing units (or TPUs), so known because they support tensor calculations. For this reason, Google's open source deep learning software is known as TensorFlow.[1]

Third, training machines to replicate and improve on complex human behavior using NNs is only possible with big data. Learning from experience improves as more and more examples are considered. We note of course, that too much data may also be a curse, as the model may be overfit in-sample, and hence work poorly out-of-sample. There is a vast deep learning literature that deals with handling the overfitting problem. However, reasonable sizes of data are needed, and this too has become much more available today than it ever was before. Ubiquitous data is a major driver of the success of DL, and a shining example of this success lies in image recognition, digit recognition, etc. Much of the DL enhancements we have today have come about because of big, high-quality data, such as that on Image-Net.[2] Millions of labeled images are used annually in the ImageNet challenge to find the current best algorithm.

Overall, DL has revolutionized AI, and resulted in its Second New Age. The three pillars of AI as we have seen are mathematical analysis, computational tractability using special purpose hardware, and the availability of big data. In the next section we will consider how Dl may be used to address finance problems, and examine an implementation of one such algorithm for option pricing.

2 A Very Brief Overview of Neural Nets

For readers that are unfamiliar with the field of deep learning neural nets, we now provide a brief introduction. Artificial Neural Networks (ANNs), sometimes referred

[1]https://www.tensorflow.org/.
[2]http://www.image-net.org/.

to as multilayer perceptrons (MLPs), have become an increasingly popular tool in the field of machine learning. While the ideas for ANNs were first introduced in McCulloch and Pitts (1943), the application of backpropagation in the 1980s, see Werbos (1975); Rumelhart et al. (1986), and recent advancements in processor speed and memory have enabled more widespread use of these models in a diverse set of fields, including medical care, autonomous navigation, and marketing analytics. These networks, once trained on large enough sets of training examples, can be used for discrete classification and continuous value prediction.

Inspired by biology, ANNs simulate neurons in the brain, each of which applies receives a number of inputs x_i, applies a function $h_i(x_i)$, and produces an output o, shown in **Figure 1**. Each neuron applies weights to its inputs and includes an activation function. The activation function introduces nonlinearities to the output, which are required for any nonlinear regression.

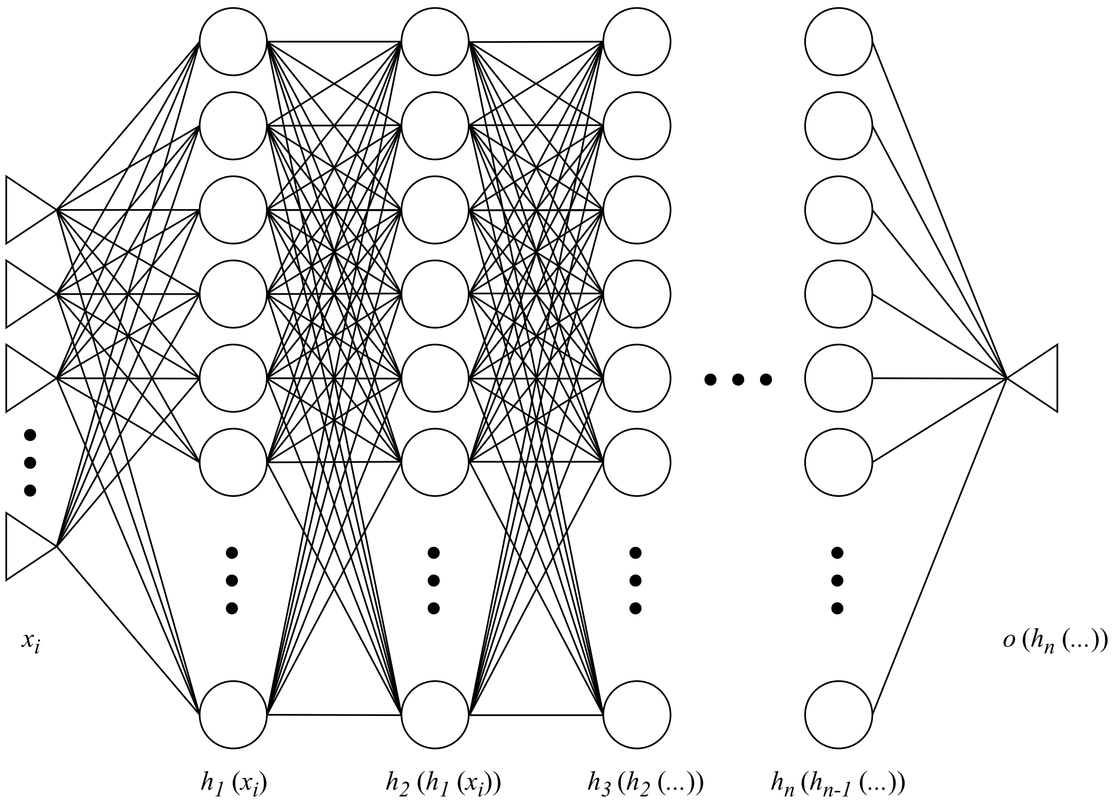

Figure 1: Feedforward Network Structure

These neurons are connected via their inputs and outputs in varying ways; we used a feedforward network for our research. A feedforward network is organized in a series of layers, each of which containing at least one neuron. The first layer of neurons consists of the original inputs to the model, and the last layer the outputs. Any layer between the input and output layer is called a hidden layer. Each of these neurons receives each output from the layer preceding it as input, and passes its own output to every neuron in the following layer.

Each node $h_n^{(j)}$ in layer n of the NN, takes in inputs $h_{n-1}^{(i)}, i = 1, 2, ..., M_{n-1}$ from the previous layer, and creates a function called the "net input":

$$a_n^{(j)} = \sum_{i=1}^{M_{n-1}} w_{ij} h_{n-1}^{(i)} + b_n^{(j)}$$

where M_{n-1} is the number of nodes at layer $(n-1)$ of the NN, and the superscripts $(i), (j)$ denote the node number in any layer. The values $b_n^{(j)}$ are known as the "bias" parameters of the node. The parameters (coefficients) of the model that are to be fit at the n-th layer are as follows:

$$w_{ij}, \quad i = 1, 2, ..., M_{n-1}, \quad j = 1, 2, ..., M_n$$

and

$$b_j, \quad j = 1, 2, ..., M_n$$

A quick count shows that between layers $(n-1)$ and n, there are $M_n \times M_{n-1} + M_n$ parameters. The net input $a_n^{(j)}$ is then passed into a nonlinear function to generate a single output:

$$h_n^{(j)} = h(a_n^{(j)})$$

The function $h(a_n^{(j)})$ if known as the "activation function" and we use three types in our paper.

- ELU (exponential linear unit). The function is $h(x) = x$ if $x \geq 0$, else $h(x) = \alpha(\exp(x) - 1)$.

- ReLU (rectified linear unit). Here the function is $h(x) = \max(x, 0)$. The gradient of the function is zero in the region where x is negative, and the neuron is not active.

- Leaky ReLU, for which the function is $h(x) = x$, if $x \geq 0$, else $h(x) = \alpha x$. In this case, the function permits a weak gradient, when the neuron is not active, i.e., α is small.

Initial weights for each neuron are randomly generated, then updated to lower error. An objective function, which describes the error, is defined by the designer. The goal of the learning task is to find the global minimum in the objective function. Our chosen objective function is the *mean squared error* (MSE), an objective function often chosen for regression problems.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(c_i - \bar{c}_i)^2$$

The minimum of the objective function is found by computing the gradient of the objective function at each neuron with respect to its weights, and then moving its weights in the opposite direction, by a factor of η, the learning rate. These gradients are computed efficiently thanks to backpropagation.

The backpropagation algorithm enables us to compute every neuron's parameters' loss gradients for the network in one pass, from the output layer back to the input layer. Otherwise, we would need to compute each loss gradient individually, a computationally expensive task. The backpropagation algorithm is effective because a feedforward network is in practice a large nested function. This means the gradient at each neuron can be calculated using the chain rule. Once the gradients for all parameters are calculated, we are then able to move each parameter in the direction that lowers the loss function. In order to prevent overshooting in adjusting the parameters, we move them by a small factor, known as the learning rate, usually set to lie between 0.05 and 0.10. Each time we update parameters using gradients, we are said to complete one "epoch" of gradient descent.

We also do not train the model to fully minimize the loss function, as that may result in overfitting to training data, leading to poor performance of the NN on new data. To prevent overfitting in our experiments later, we will choose (i) the number of epochs to use, and (ii) we may use only some nodes of the network in each epoch, as this approach, known as "dropout", also reduces overfitting. (iii) We also use only some of the data to compute gradients, in a process known as "batch gradient descent" and this requires choosing a batch size. These settings are known as "hyper-parameters" of the model, and are judgment calls made by the analyst, as are the choices made as to how many layers are to be used in the NN, and the choice of the number of nodes at each layer.

Now that we have seen briefly how NNs are implemented, it is useful to point out the Universal Approximation Theorem. This theorem states that NNs with a single hidden layer with a very large number of nodes and activation functions that are continuous, bounded, and non-constant, can approximate any function arbitrarily closely, Hornik (1991). Recent theoretical work, by Telgarsky (2016) argues that this is not true unless the hidden layers are exponentially large. While the universal approximation debate continues, the fact is that deep learning NNs are proving to extremely effective in practice.

With this brief introduction to NN architecture, we proceed on to a more specific exploration of DL in Finance.

3 Deep Learning in Finance

Deep learning tools have become commoditized. In addition to TensorFlow there are other popular open source implementations of deep learning such as MXNet[3], supported on Amazon Web Services (AWS), and h2o[4], the latter being a general purpose platform supporting many different DL implementations, including Caffe[5], Torch[6], and Theano[7]. DL tools are supported in many programming languages such

[3]http://mxnet.io/
[4]https://www.h2o.ai/
[5]http://caffe.berkeleyvision.org/
[6]http://torch.ch/
[7]http://deeplearning.net/software/theano/

as Python and R. Therefore, whereas tools are a commodity, ideas and DL models are not, and need creative thought to define the goals of the NN and the structure to be used. This is where DL becomes an art form.

There are a many conditions that make DL an important architecture for the field of Finance. First, the availability of large data, often streaming in at rates in no other field. Second, several finance applications depend on speed, and the advent of efficacious hardware in DL makes it possible to achieve response levels that are key to making trading algorithms viable. Third, much of finance involves pattern recognition using data, where multifarious inputs are modeled to predict outputs. For example, stock market prediction may be based on many variables (streaming data on stock prices, interest rates, volatilities, etc.). Another case is in consumer banking, where customers are characterized by myriad variables to determine what products to offer them, or to compute their probabilities of retention. We note that this pattern recognition on big data is analogous to the ImageNet problem. Therefore, DL architectures that can learn to recognize an image can be directly used to learn to recognize (for example) signatures in the stock market that predict direction of the index. Or it may be used to train a model to learn how the market prices options, the example we will implement in this paper.

What does DL uncover that is not possible with standard econometric models? The answer is "non linearities". Most econometric models today are linear functions, or simple transformations of linear functions. However, the relationship between inputs and outputs may be hugely nonlinear. This is what a deep learning NN is adept at picking up. As the data is passed from one layer to the next, it is transformed into new data, and the layers of nonlinearity get peeled away. This suggests that a deep NN can learn almost any function to a high degree of accuracy (some caveats on this later). There are therefore a wide range of applications in Finance to which DL applies. We next consider a simple one, i.e., learning option pricing.

4 Deep Learning for Option Pricing

The Black and Scholes (1973) and Merton (1973) formula (BSM) is probably one of the most widely cited and used in finance, and has led to a huge number of variations and extensions. The formula is used to price call and put options and is the solution to a special partial differential equation (PDE) first derived in Black and Scholes (1973). The fact that there is a closed-form solution to this PDE is remarkable. The model starts from assuming a specific form of movement for the stock price, namely a geometric Brownian motion (GBM), and based on the conditional payment at maturity of the option, derives the PDE that must be satisfied to meet specific economic constraints, such that arbitrage is inadmissible.

Of course, options traders are well aware that the GBM assumption is violated in practice, and therefore, they use other models that are extensions of the BSM. In addition, prices from these extended models are further adjusted using traders' judgment, and therefore, prices in the markets do not come exactly from the BSM, nor is it possible to state what the exact mathematical function is that generates market prices, as it would be an amalgam of prices from different traders using varied models.

However, market data is plentiful, so it is possible to train an algorithm to "learn" the function that is collectively generating option prices in the market. This was first attempted by Hutchinson et al. (1994) and they demonstrated that a neural network is an excellent algorithm with which to approximate the market's option pricing function. Since the time of this paper, we have now had several developments in neural network technology that we summarized earlier in this article. We will show that it is very easy for a novice financial modeler to use neural networks to achieve high levels of predictive performance.

To revisit the main ideas, the main improvements are twofold. First, mathematical innovations have vastly expanded the type, scope, and our understanding of neural networks, so that we may use these new methods to revisit the original problem that Hutchinson et al. (1994) examined. We use a neural network that is much larger than the one they used, not feasible twenty years ago. Second, neural networks with many layers are able to capture subtle nonlinearities in the data that were not possible with more or less linear statistical approaches. Because these neural nets contain many layers, they are known as "deep learning nets" and posed severe computational difficulties in the past. However, today, advancements in hardware have rendered this problem computable, so that training a deep learning net on high performance CPUs, GPUs, and other specialized hardware is completely feasible. As nets have become deeper, they are providing remarkable performance on tasks that were thought to be beyond the reach of machines, such as language translation, image recognition, driving cars, etc.

We begin with the specific exercise of learning the Black-Scholes option pricing model from simulated data. If this is possible to achieve with high accuracy, then it suggests that market data may be used to train an option pricing model that better matches market prices than the BSM.

5 Analysis

Hutchinson et al. (1994) explored using neural nets to learn the famous Black and Scholes (1973) option pricing formula, also developed and analyzed by Merton (1973). Using limited computing power and relatively small neural networks, they were able to show remarkably good performance in mimicking (learning) the following equation from simulated data. We present the call option (C) pricing equation here.

$$C = Se^{-qT}N(d_1) - Ke^{-rT}N(d_2) \tag{1}$$

$$d_1 = \frac{\ln(S/K) + (r - 0.5\sigma^2)T}{\sigma\sqrt{T}} \tag{2}$$

$$d_2 = d_1 - \sigma\sqrt{T} \tag{3}$$

where S is the current stock price, K is the option strike, T is option maturity, q, r are the annualized dividend and risk free rates, respectively. Finally, σ is the annualized volatility, i.e., the standard deviation of the return on the stock.

In order to create data for the assessment of how a deep neural net would learn this equation, we simulated a range of call option prices using a range of parameters shown in **Table 1**.

Table 1: The range of parameters used to simulate 300,000 call option prices. The strike prices K were chosen to lie within the vicinity of the stock price S, so as to be realistic.

Parameter	Range
Stock price (S)	\$10 – \$500
Strike price (K)	\$7 – \$650
Maturity (T)	1 day to 3 years
Dividend rate (q)	0% – 3%
Risk free rate (r)	1% – 3%
Volatility (σ)	5% – 90%
Call price (C)	\$0 – \$328

We divided this data into two random sets, one for training, comprising 240,000 option prices, and we held out the remaining 60,000 prices for validation.

Before passing the prices to the deep learning net, we exploited a facet of the Black-Scholes call option function, i.e., that the pricing function is linear homogenous in (S, K), i.e., $C(S, K) = K \cdot C(S/K, 1)$. Therefore,

$$C(S, K)/K = C(S/K, 1)$$

Accordingly, we modified our data by dividing both stock price S and call price C by strike price K. This normalized data was then fed into the deep learning net to fit the input variables S, K, T, q, r, σ (the feature set) to the output prices C.

The details of the deep learning net are as follows. The size of the input is 6 parameters. These are passed through 4 hidden layers of 100 neurons each. The neurons at each layer are chosen based on different "activation" functions that are respectively the following: LeakyReLU, ELU, ReLU, ELU. The final output layer comprises a single output neuron which we set to be the standard exponential function $\exp(\cdot)$ because we need the output of the neural net to be non-negative with certainty, as option prices cannot take negative values.

We chose some simple "hyper-parameters" for the deep learning net. At each hidden layer we used a dropout rate of 25% so as to ameliorate overfitting. The loss function used for optimization is mean-squared error (MSE) and we implemented a batch size of 64, with 10 epochs. The entire exercise results in fitting a total of 31,101 coefficients (weights) for the deep learning model. The model was trained using Google's TensorFlow package. The results are as follows.

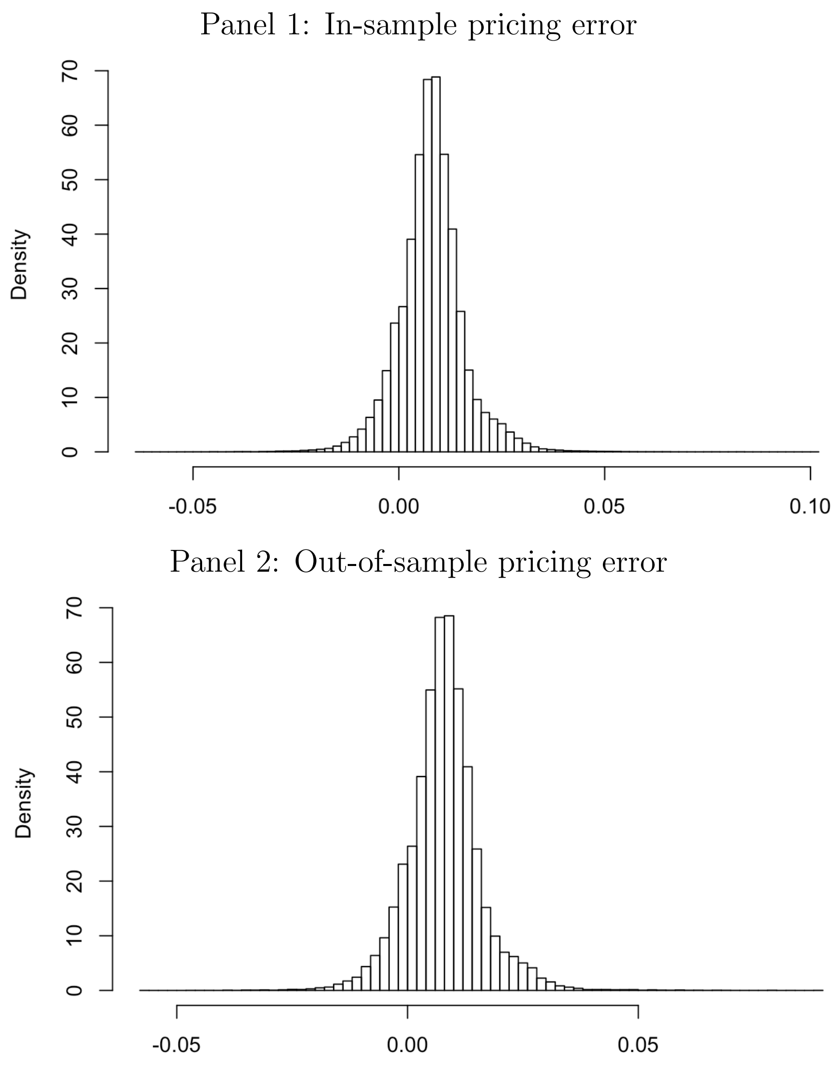

Figure 2: Call option pricing errors from the fitted deep learning neural net.

1. *In-sample*: The root mean-squared error (RMSE) is 0.0112, which may be compared to the fact that the strike prices are all normalized to \$1. Hence the average error is ±1% of the strike. The average percentage pricing error (error divided by option price) is 0.0420, i.e., 4%. The histogram of pricing errors is shown in Panel 1 of **Figure 2**. We see that the errors are very small. Further,

we estimated a regression of the model values of the Black-Scholes equation on the true values, and attached an $R^2 = 0.9982$, which is very high.

2. *Out-of-sample*: The root mean-squared error (RMSE) is also 0.0112, with an average error of $\pm 1\%$ of the strike. The average percentage pricing error (error divided by option price) is 0.0421, i.e., 4%. The histogram of pricing errors is shown in Panel 2 of **Figure 2**. Again, the errors are very small. Further, we estimated a regression of the model values of the Black-Scholes equation on the true values, and attached an $R^2 = 0.9982$, which is very high. Since the results from the in-sample test and those out-of-sample are the same, no overfitting was evidenced.

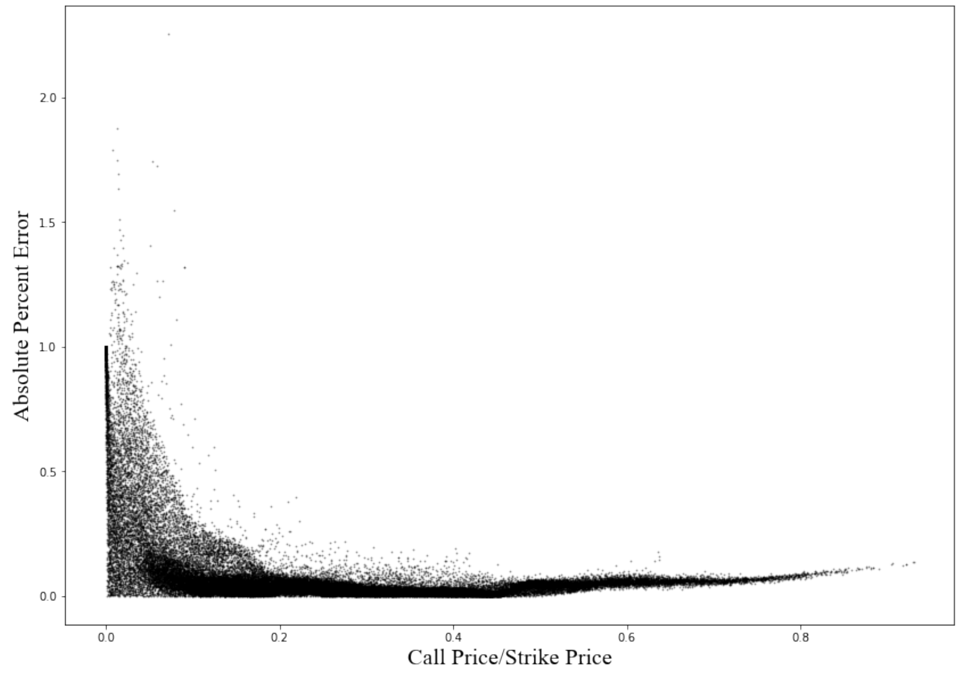

Figure 3: Percent Error by Moneyness

We also investigated if the error was correlated with the option price or the moneyness of the option, see **Figure 3**. Other than for very small option prices, where the percentage error tends to be magnified, moneyness is not correlated with pricing error.

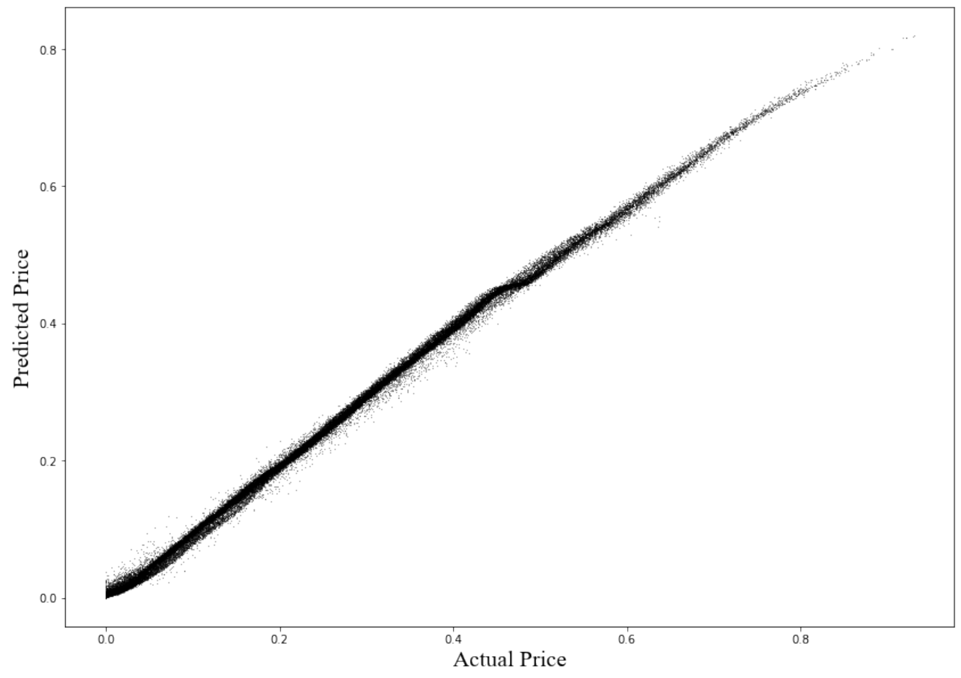

Figure 4: Actual vs. Predicted Price

Figure 4 plots the actual price against the predicted price of each option, yielding a narrow line with very few deviations, indicating very few significant errors in pricing.

6 Concluding Discussion

The confluence of mathematical analysis, improvements in hardware and software, and the ubiquity of big data to train neural network models have brought artificial intelligence into its second new age. We argue that deep learning applications such as image classification are machine learning problems that may be ported to finance applications. We offer a brief overview of the ideas in AI, deep learning, and then proceed to show how these models may be mapped to a canonical problem in finance, that of option pricing. In this brief survey, we revisit the original work of Hutchinson et al. (1994) that used neural networks for option pricing. Our experiments show that simple deep learning nets can learn to price options with very low error. This architecture can easily be extended to pricing options in the real world, with no knowledge of the theory of option pricing. New technology has commoditized the use of deep learning, so it is very easy for an investment manager or trader to implement these models.

References

Black, F. and M. Scholes (1973). The pricing of options and corporate liabilities. *Journal of Political Economy 81*, 637–659.

Bryson, A. E. (1961, April). A gradient method for optimizing multi-stage allocation processes. In *Proceedings of the Harvard Univ. Symposium on digital computers and their applications.*

Dreyfus, S. (1973). The computational solution of optimal control problems with time lag. *IEEE Transactions on Automatic Control 18*(4), 383–385.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks 4*(2), 251–257.

Hutchinson, J. M., A. W. Lo, and T. Poggio (1994). A nonparametric approach to pricing and hedging derivative securities via learning networks. *Journal of Finance 49*(3), 851–889.

Kelley, H. J. (1960). Gradient theory of optimal flight paths. *Ars Journal 30*(10), 947–954.

McCulloch, W. and W. Pitts (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 5*(4), 115–133.

Merton, R. C. (1973). Rational theory of option pricing. *Bell Journal of Economics and Management Science 4*, 141–183.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning representations by back-propagating errors. *Nature 323*(6088), 533–536.

Telgarsky, M. (2016, May). Benefits of depth in neural networks. *JMLR: Workshop and Conference Proceedings 49*, 1–23.

Werbos, P. (1975, October). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE 78*(10), 1550–1560.