Deeply Learning Derivatives

Ryan Ferguson[*]and Andrew Green[‡]

14/10/2018

Version 2.1

Abstract

This paper uses *deep learning* to value derivatives. The approach is broadly applicable, and we use a call option on a basket of stocks as an example. We show that the deep learning model is accurate and very fast, capable of producing valuations a million times faster than traditional models. We develop a methodology to randomly generate appropriate training data and explore the impact of several parameters including layer width and depth, training data quality and quantity on model speed and accuracy.

1 Introduction

1.1 The need for speed

Quantitative Finance is a demanding taskmaster. Never satisfied with the progress of Moore's Law, a burgeoning cadre of quantitative analysts is employed to find new techniques that deliver more accuracy with lowered computational effort. We have a long list of changes we would make to our models if we had the computational resources. Computationally burdensome valuation adjustments are becoming increasingly important elements of derivative valuation and pricing(see for example Green, 2015). New regulations like the Fundamental Review of the Trading Book, add further computational costs.

One approach to meeting these challenges is the development of approximations to computationally expensive valuation functions. These techniques are commonly applied in XVA models, where trades are typically valued repeatedly inside a Monte Carlo simulation. Techniques from *Approximation Theory* such as Chebyshev interpolation techniques have been applied in the context of XVA and more generally (Zeron and Ruiz, 2017; Gaß et al., 2015).

Deep Neural Networks (DNNs) bring major benefits to the approximation of functions:

- The *Universal Approximation theorem* states that simple feed forward networks can represent a wide variety of functions under mild assumptions about the activation function (Cybenko, 1989; Hornik, 1991)
- DNNs bifurcate accuracy and valuation time. The catch is that computation time must be invested upfront in training the neural networks.

[*]Contact: ryan.ferguson@scotiabank.com. Ryan Ferguson is a Managing Director at Scotiabank™ in Toronto.

[†]Contact: andrew.green2@scotiabank.com. Andrew Green is a Managing Director at Scotiabank™ in London.

[‡]The views expressed in this article are the personal views of the authors and do not necessarily reflect those of Scotiabank™. ™ Trademark of The Bank of Nova Scotia and used under license. Important legal information and additional information on the trademark may be accessed here: `http://www.gbm.scotiabank.com/LegalNotices.htm`

- DNNs do not suffer from the curse of dimensionality. The techniques developed in the paper can be applied to valuation models with hundreds of input parameters.
- DNNs are broadly applicable. They can be trained by a wide variety of traditional models that employ Monte Carlo simulation, finite differences, binomial trees, etc. as their underlying framework.

1.2 Neural Networks in Finance

Neural networks are perhaps best known in finance in the context of predictive algorithms for use as trading strategies (see for example Tenti, 1996). Credit scoring and bankruptcy prediction also have also seen application of neural network models. Neural networks have recently been applied to CVA, alongside other machine learning techniques, where a classifier approach has been used to map CDS to illiquid counterparties (Brummelhuis and Luo, 2017). Neural networks previously have been applied to the approximation of derivative valuations. Hutchinson et al. (1994) applied neural networks to the Black-Scholes model in an early financial application. More recently, Culkin and Das (2017) applied Deep Neural Networks to the same problem. Recently the development of the *Deep BSDE solver* offers a new way of solving Backward Stochastic Differential Equations in high dimension (E et al., 2017; Henry-Labordere, 2017). BSDE have applications in finance, stochastic control and elsewhere

1.3 Applying Deep Learning to Derivatives Valuation

This paper makes the following contributions:

1. Demonstrates the use of deep neural network models as approximations to derivative valuation routines and provides a basket option as an example.
2. Develops a training methodology for neural network models, where the training and test set are generated using Monte Carlo simulation.
3. Shows that the final deep neural network has substantially lower error that the random error of the Monte Carlo models used to train it. The neural network learns to remove random Monte Carlo noise.
4. Explores a range of geometries for neural network models.
5. Provides an assessment of computational performance, while making use of CPU and GPU parallelism during training set generation and network parameter fitting.

The remainder of this paper is organised as follows. In Section 2, we provide a brief overview of deep neural network models and the associated fitting algorithm. In Section 3 we describe the application of deep neural networks to valuing derivatives by using a call option on a basket of stocks as an example. The example is used to demonstrate the accuracy and performance of various deep learning models. Section 4 concludes with applications and future work.

2 Deep Neural Networks

2.1 Introducing Neural Networks

An artificial neural network consists of a series of layers of *artificial neurons*.[1][2] Each neuron takes a vector of inputs from the previous layer, $a_j^{[l-1]}$ and applies a weight vector, $W^{[l]j}$ and

[1]For a detailed introduction to Deep Learning, see Goodfellow et al. (2016) and Ng (2018).
[2]A summary of the notation used in this article can be found in Table 1.

Notation	Description
$\boldsymbol{x}; x_j$	input layer, jth element of input layer
$\boldsymbol{y}$	Output (vector or scalar depending on problem context)
$\hat{y}^{(i)}$	Output value from the neural network for training example i
$(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$	ith training example
$(\boldsymbol{X}, \boldsymbol{Y})$	Matrices of all training examples
$\boldsymbol{W}^{[l]}; W_j^{[l]}$	Weight matrix for the lth layer and jth component vector
$\boldsymbol{b}^{[l]}; b_j^{[l]}$	Bias vector for the lth layer and jth element
L	Number of layers in the neural network
$g(z)$	Non-linear activation function
$\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[l]} X + \boldsymbol{b}^{[l]}$	Result of matrix operations on layer l
$\boldsymbol{a}^{[l]} = g(\boldsymbol{Z}^{[l]})$	Results of activation function on layer l
$\boldsymbol{a}^{[0]} = \boldsymbol{x}; \boldsymbol{a}^{[L]} = \boldsymbol{y}$	Input and output layers
$J(\boldsymbol{W}^{[l]}, \boldsymbol{b}^{[l]})$	Cost function
$\odot$	Element-wise multiplication

Table 1: Neural network notation used in this article.

Function	Definition
Sigmoid	$1/(1+e^{-z})$
tanh	$(e^z - e^{-z})/(e^z + e^{-z})$
ReLU	$\max(0, z)$
Leaky ReLU	$\max(0.01z, z)$

Table 2: The four main types of activation function defined

a bias $b_j^{[l]}$ so that

$$Z_j^{[l]} = W_j^{[l]} a^{[l-1]} + b_j^{[l]} \tag{1}$$

or in matrix notation

$$\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[l]} \boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}. \tag{2}$$

The result from layer l is then given by applying a non-linear *activation function* $g(Z)$,

$$\boldsymbol{a}^{[l]} = g(Z^{[l]}). \tag{3}$$

A number of different activation functions can be used and the main types are listed in Table 2. In order to produce non-linear outputs it is necessary that a non-linear function be used. All neurons in the same layer use the same activation function but different layers often use different activation functions. Equations (2) and (3) together describe how the input $\boldsymbol{a}^{[0]} = \boldsymbol{x}$ vector is *forward propagated* through the network to give the final output $\boldsymbol{a}^{[L]} = \boldsymbol{y}$.

The number of inputs to the model is dictated by the number of input *features*, while the number of neurons in the output layer is determined by the problem. For a regression problem with one real-valued output, as described in this paper, there will be a single node in the output layer. If a neural network model is used in a classifier problem with multiple classes, there will be one node per class. The number of layers in the model, L and the number of neurons in each layer can be considered *hyperparameters* and these are normally set using hyperparameter tuning over a portion of the training data set, as described in Section 2.3. Models with many hidden layers are known as *deep neural networks*. A example neural network geometry is illustrated in Figure 1.

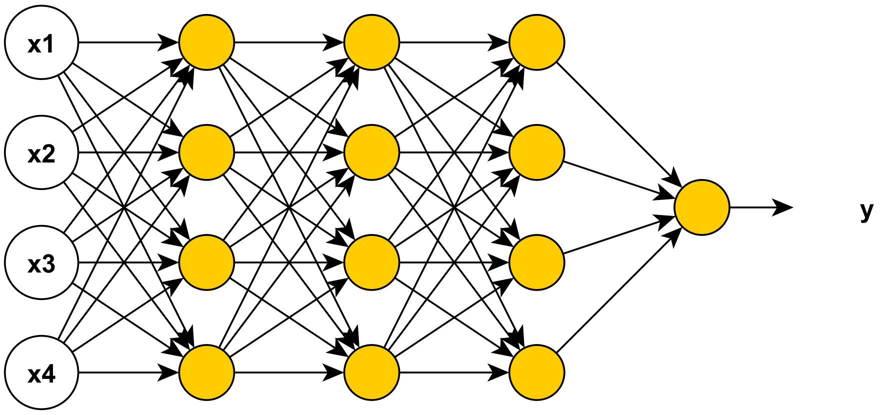


Figure 1: A Neural Network with four inputs, one output and three hidden layers.

2.2 Training the Model and Back Propagation

During the training phase, the weights and bias parameters of the model are systematically updated to minimize the error between the training data and estimates generated by the model. A set of m_{train} training examples is used and the error is given by a cost function J,

$$J(\boldsymbol{W}^{[l]}, \boldsymbol{b}^{[l]}) = \sum_{i}^{m_{\text{train}}} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \boldsymbol{\mathcal{L}}(\hat{\boldsymbol{y}}, \boldsymbol{y}), \tag{4}$$

where the function $\mathcal{L}$, depends on the choice of loss measure. A number of choices are available for the loss measure including the L2 or L1 norm.

The minimum is found using an optimization procedure. The standard approach uses batch gradient descent or a variant such as mini-batch gradient descent.[3] Such optimization procedures require the gradients of the cost function,

$$d\boldsymbol{W}^{[l]} = \frac{\partial J}{\partial \boldsymbol{W}^{[l]}} \tag{5}$$
$$d\boldsymbol{b}^{[l]} = \frac{\partial J}{\partial \boldsymbol{b}^{[l]}}, \tag{6}$$

which are used to update the weights,

$$\boldsymbol{W}^{[l]} = \boldsymbol{W}^{[l]} - \alpha d\boldsymbol{W}^{[l]} \tag{7}$$
$$\boldsymbol{b}^{[l]} = \boldsymbol{b}^{[l]} - \alpha d\boldsymbol{b}^{[l]}. \tag{8}$$

To obtain the gradients a specialised type of *algorithmic differentiation* is used, known as *backpropagation*. This algorithm recursively applies the chain rule to propagate derivatives back through the computational graph of the neural network model. Beginning with the output layer, the derivative $d\boldsymbol{a}^{[L]}$ is defined by

$$d\boldsymbol{a}^{[L]} = \nabla_{\boldsymbol{a}^{[L]}=\hat{\boldsymbol{y}}} \boldsymbol{\mathcal{L}}(\hat{\boldsymbol{y}}, \boldsymbol{y}). \tag{9}$$

The derivative $d\boldsymbol{Z}^{[L]}$ is then given by,

$$d\boldsymbol{Z}^{[L]} = d\boldsymbol{a}^{[L]} \odot g'(\boldsymbol{Z}^{[L]}). \tag{10}$$

[3] In this paper we use mini-batch gradient descent where the size of each mini-batch is a hyperparameter.

$$\begin{tikzcd}
a^{[0]} \arrow[r] & \boxed{W^{[1]}, b^{[1]}} \arrow[r] \arrow[d] & a^{[1]} \arrow[r] & \cdots \arrow[r] & \boxed{W^{[L]}, b^{[L]}} \arrow[r] \arrow[d] & a^{[L]} = \hat{y} \\
& Z^{[1]} \arrow[d] & & & Z^{[L]} \arrow[d] & \\
& \boxed{dZ^{[1]}} \arrow[d] & da^{[1]} \arrow[l] & \cdots \arrow[l] & \boxed{dZ^{[L]}} \arrow[l] \arrow[d] & da^{[L]} \arrow[l] \\
& dW^{[1]}, db^{[1]} & & & dW^{[L]}, db^{[L]} &
\end{tikzcd}$$

Figure 2: Forward and backward propagation algorithms for a Neural Network with L layers (see text for full description)

Hence the derivatives of the weights for layer L are given by,

$$d\boldsymbol{W}^{[L]} = \frac{1}{m} d\boldsymbol{Z}^{[L]}.\boldsymbol{a}^{[L-1]} \tag{11}$$

$$d\boldsymbol{b}^{[L]} = \frac{1}{m} d\boldsymbol{Z}^{[L]}. \tag{12}$$

The derivative $d\boldsymbol{a}^{[L-1]}$ for the next layer are then given by,

$$d\boldsymbol{a}^{[L-1]} = (\boldsymbol{W}^{[L]})^T.d\boldsymbol{Z}^{[L]}, \tag{13}$$

and hence the derivatives can be propagated backwards through the network to obtain all the $d\boldsymbol{W}^{[l]}$ and $d\boldsymbol{b}^{[l]}$. Forward and backward propagation are illustrated in Figure 2.

A set of data is used to train the model. Typically, a data set of m examples is divided into three independent groups, m_{train} , which is used to train the network, m_{test} which is used to test the trained network and m_{dev} which is used to develop or cross-validate the hyperparameters of the model. The fraction of data divided into each group depends on how much training data is available, with m_{test} and m_{dev} forming a relatively larger proportion if m is relatively small. The training stops after a fixed number of iterations, or when the loss function is no longer decreasing.

The weights are initialized using a pseudo-random number generator, with samples drawn either from the uniform distribution or the standard normal distribution. One common procedure with the ReLU activation function initializes the weights in layer l using a standard normal distribution with mean zero and variance $2/n^{[l]}$.

2.3 Hyperparameter Tuning

While the weights, $\boldsymbol{W}$ and the bias parameters, $\boldsymbol{b}$, are defined during training, other parameters such as the *learning rate* α and the topology of the neural network are *hyperparameters* that are not directly trained. The hyperparameters could be specified exogenously, however, in general their optimal values are not known *a priori*. Hence hyperparameters are optimised by assessing model performance, once trained, on a separate independent set of samples.

In this paper we focus on a few different axes of hyperparameter tuning by varying the number of nodes per hidden layer, the number of hidden layers, the size of each batch of training data, the total amount of data available for training, the quality of the training data. We demonstrate the impact these hyperparameters have on accuracy and performance.

3 Derivatives Valuation and Deep Learning

A derivatives valuation model is ultimately just a function which maps inputs, consisting of market data and trade specific terms, to a single output representing the value. That function may have an known analytic form, though frequently numerical approaches including Monte Carlo simulation, binomial trees or finite difference techniques must be used. Simple European stock options can be valued with as few as five inputs, while more complex products like Bermudan swaptions have valuation functions requiring many more inputs, involving all the properties of the underlying swap and option exercise schedule. The number of parameters could be in the hundreds or thousands for such complex products. Having a large number of inputs to a neural network model is certainly feasible and many current practical applications have inputs of a similar order of magnitude. For example, an image recognition application might use images 64 x 64 pixels in size with 3 colours, with each sub-pixel in itself a model input. However, the dimensionality of the input does scale the requirements for the size training dataset.

When training a model to approximate a derivatives valuation function, one of the first decisions is the domain of application. We can choose to train on all the parameters of the valuation model or only some of them. Even then, we can opt to train over a large domain for a given parameter, or a much smaller domain. The trade-off is one of generality versus model-complexity and training time. For example, in the case of a Bermudan swaption we may choose to take as given the properties of a specific trade and then only train the model against a variety of input market data scenarios. Given the reduction in size of the parameter space, the models will be smaller and their training requirements such as the size of training data set, the amount of time spent training, etc lower.

It is also important to note that the model can only be trusted to approximate the function well over the parameter ranges that were used to train it. Should inputs move outside of these ranges then it is unlikely the approximation will perform well. It is prudent to monitor the use of the model and retrain should inputs move outside the training range.

In this paper we train models which can generalize over a large number of parameters with broad ranges of applicability and leave the development of domain-specific models for future work.

3.1 Derivatives Pricing Example: Basket Options

To demonstrate how a network can learn a derivative valuation model we trained a network to price a European call option on a worst-of basket with six underlying stocks:

$$V = \max(0, \min(Stock1, Stock2, Stock3, Stock4, Stock5, Stock6) - K) \tag{14}$$

This option is commonly valued using a Monte Carlo simulation with each of the stocks following a geometric Brownian motion. This example was selected for four reasons: First the dimensionality of the function is moderate, with the number of input parameters equal to $\frac{1}{2}n(n+3)+1$. In our example with six underlying stocks, we have a 28-dimensional input space. Second, the use of Monte Carlo explores the use of deep learning with a computationally intensive valuation routine. Third, it will be easy to scale up the dimensionality of this problem by adding more stocks to the basket. Fourth, the use of a numerical method like Monte Carlo means that the function to be replicated has numerical noise.

To fit a neural network model we need to generate a set of training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$, with the $\boldsymbol{x}^{(i)}$ being input model parameters and the $y^{(i)}$ being the output value. To generate the training data we use Monte Carlo simulation to generate random sets of inputs within appropriate ranges selected for each parameter. In the context of this example all parameters were generated independently with the distribution selected individually for each parameter. The choice of sampling distribution is important and should be made on a case by case basis. For example, there is little point in generating a large volume of examples which yield

Parameter Type	**Number**	**Distribution**
Forward stock prices	6	100(lognormal(0.5, 0.25))
Stock volatilities	6	uniform(0,1)
Maturity	1	uniform(1,43)**2
Correlations	15	2($\beta(5,2)$-0.5)

Table 3: Parameters of the basket option with six underlyings.

zero option value. Hence we anticipate that different choices of sampling distribution, where the parameters are not independent will be needed for some derivative models. In other cases, stratification or importance sampling may be appropriate but we leave this for further research.

Once a random set of parameter values is created, the function to be replicated is called to generate a value. In this way a large number of training examples can be created. This can be computationally intensive as the derivative valuation function is called many times. However, this process needs only be done once for the initial training of the model. Given all training examples are independent it can also be readily parallelized.

3.2 Training for Basket Options

For our basket option example, we reduced the number of input parameters by assuming flat implied volatility surfaces for each of the stocks in the basket. Further simplification was made by assuming both zero interest rates and dividends. For the case of six underlying stocks we have the parameter breakdown as shown in Table 3.

The parameters were sampled randomly and independently, with the exception of correlation matrices which were handled separately. Stock prices were generated using $100 * \exp(z)$, with $z \sim N(0.5, 0.25)$. The correlation matrices were generated using the C-vine method of Lewandowski et al. (2009).

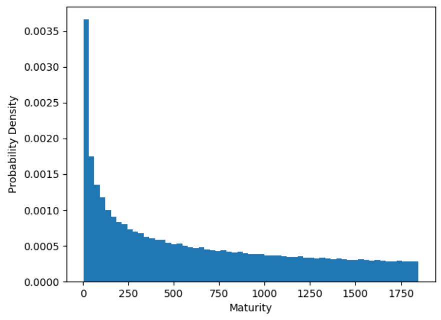
Since a neural network model will minimize the error between its estimates and that of the training data that it is presented, it is important to choose a representative data set. Another consideration when developing training data is the nature of the function being learned. Areas of rapidly changing function value need to be assigned more training data. We generated more data for short maturities since the convexity of the basket option valuation function is greatest for at the money short dated options. The sample distributions of the inputs are illustrated in Figure 3.

We generated three different training sets using the same distributions as in Table 3, with properties given in Table 4. The time taken to generate each was approximately the same, approximately a week of server time using all of the cores.

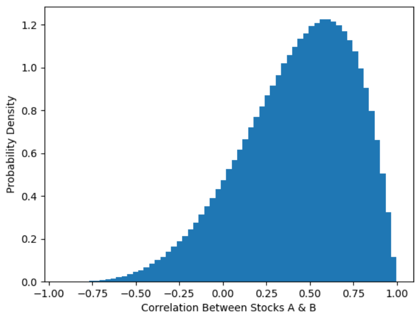
Training Set	**# Examples**	**# MC Paths**
A	5M	1M
B	50M	100k
C	500M	10k

Table 4: Properties of the three training sets.

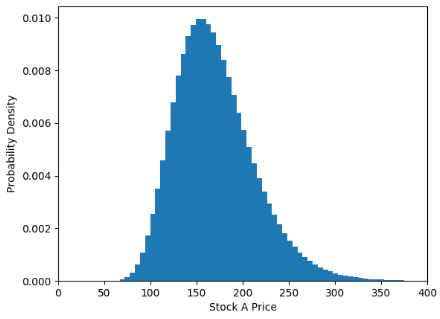
The models were trained with minibatch sizes of 50,000 samples. The test set consisted of 5,000 samples drawn at random from a separate set of highly accurate MC generated data (100mm paths) used only for testing. Using a 24-core AMD EPYC 7401P server we computed values for the test set in a little more than a week by fully using all of the cores. By using 100 million paths we can obtain Monte Carlo accuracy to within 1 cent. However, the valuation of a single option took approximately 300 seconds making it of little use in a production environment. If 5 cent accuracy (1 million paths) or 20 cent accuracy (100k



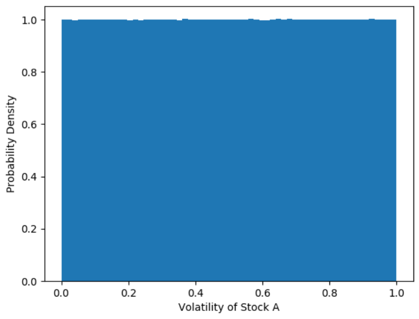
(a)



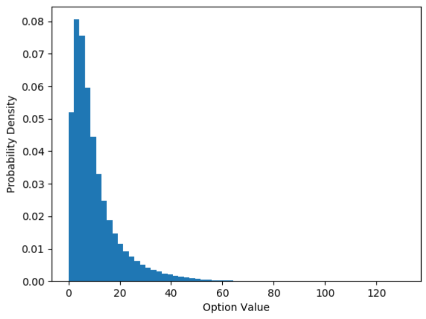
(b)



(c)



(d)



(e)

Figure 3: Sample distributions of input variables used in the basket option training and test sets. Given there a six input assets to the basket and the sampling was independent, only one representative example is illustrated for stock, volatility and correlation. The individual figures illustrate (a) maturity (in days), (b) correlation between stock A and stock B, (c) stock price A, (d) volatility of stocks and (e) option value.

paths) is acceptable, then valuing the test set can be completed in an hour and 45 minutes, or 10 minutes, respectively.

All models were trained using an Adam optimizer (Kingma and Ba, 2014) with the learning rate set to 1e-3, $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

3.3 Results

3.3.1 Training Performance

Different models, all with six hidden layers were trained. The models were differentiated by the training set used and number of nodes in each hidden layer. Each hidden layer used ReLU activation functions with the final output layer a simple linear function to give a real valued output.

The first experiment used models trained with training set A. Figure 4 illustrates the learning capacity as a function of the number of mini-batch iterations for each of the five models. Two curves are presented for each model, giving the learning curve for the test and training sets. The curves represent the minimum loss from each iteration of the Adam optimizer. This is a classic example of overtraining; the training set error continues to decrease while the test set error remains stubbornly high even for the best models. For the largest model, with 600 nodes per layer, the overfitting is so pronounced that the generalization on the out-of-sample is materially worse than that of the smaller models.

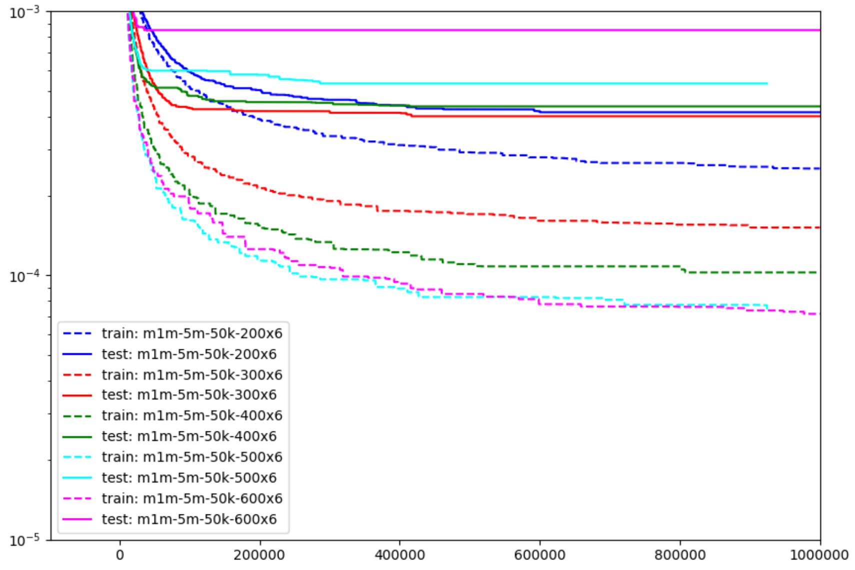


Figure 4: Minimum loss as a function of mini-batch iteration for five different models with 200, 300, 400, 500 and 600 nodes per layer.

Training set B was used to train the second set of models. These models used 400, 600, 1200, 1400 and 1600 nodes per layer with the learning curves illustrated in figure 5.

The final set of experiments were conducted training models with the same number of neurons per layer as case B, but using training set C, with only 10k Monte Carlo paths, and

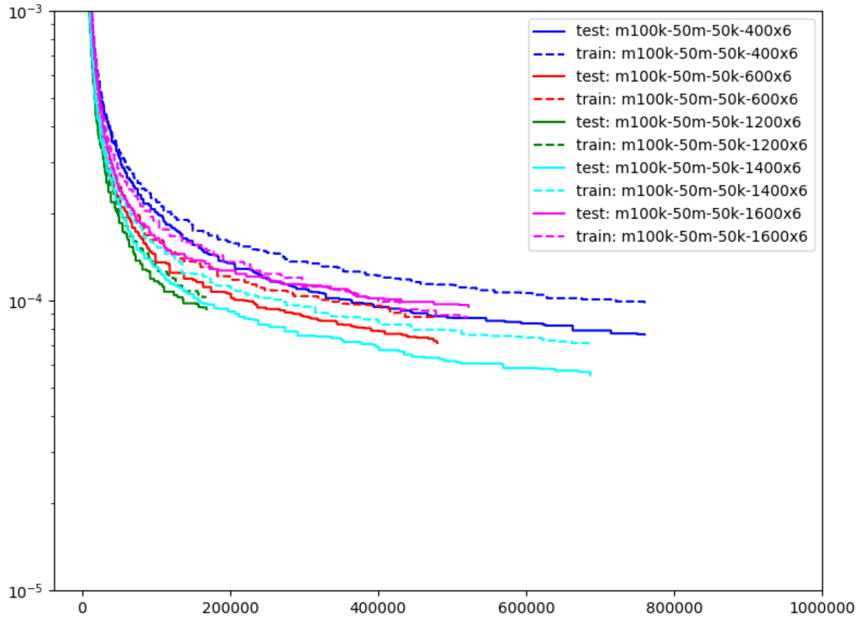


Figure 5: Minimum loss as a function of mini-batch iteration for five different models with 400, 600, 1200, 1400 and 1600 nodes per layer.

the results are illustrated in figure 6. With a larger training set with more Monte Carlo noise in each training example, the results were significantly better. Empirically it appears better to distribute the Monte Carlo scenarios broadly through the model domain rather than to concentrate them at points generating highly accurate training data. With more data, larger models with more capacity can be trained before overfitting sets in. A key observation is that the test results are better than the training results; that is the models have learned to average out the numerical noise associated with the lower-quality training data.

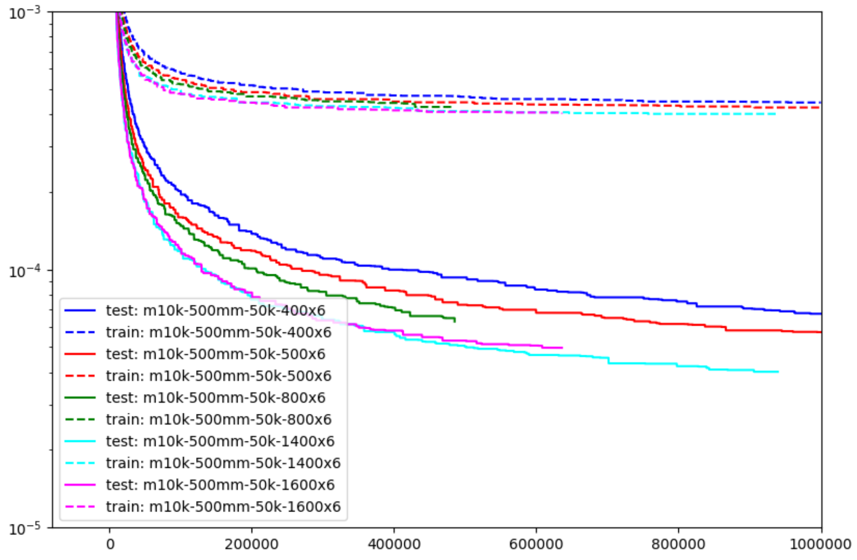
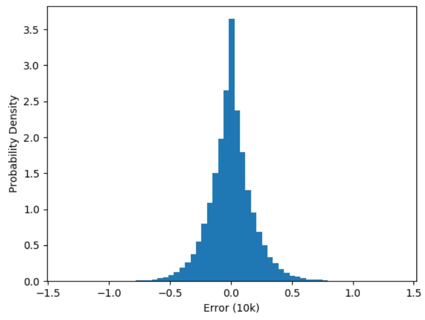


Figure 6: Minimum loss as a function of mini-batch iteration for five different models with 400, 600, 1200, 1400 and 1600 nodes per layer.

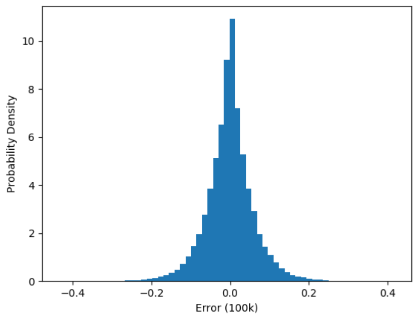
The logical conclusion of this suggests that the best approach may be obtained using single Monte Carlo paths directly. Since each of our training samples are each approximately 300 bytes in size, each training set would consume 1.5 petabytes of disk space. While this is not impossibly large, training approaches using on the fly data generation seem more reasonable. We propose training Neural Networks directly from the dynamic Monte Carlo simulation, with paths grouped into independent mini-batches. However, we leave this for further work.

3.3.2 Accuracy

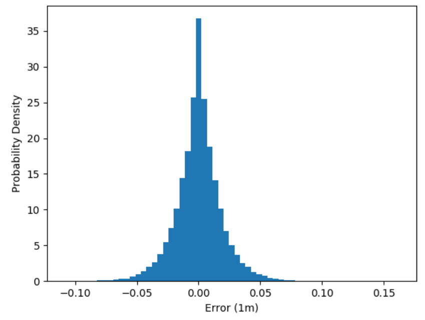
In order to benchmark our deep learning approach against classical Monte Carlo valuation techniques, we valued these test cases using Quantlib's MCEuropeanBasketEngine with 10k, 100k, 1mm, 10mm and 100mm scenarios. The histogram of differences compared with a second MC evaluation of 100mm paths is presented in Figure 7. The figure also shows the results of one of the models trained during this study. The results are quite similar to the Monte Carlo examples with 100k scenarios. However, there is one big difference. As noted earlier, The Monte Carlo results took a little over 10 minutes to calculate using all the cores of our Epyc 7401P server. The deep learning inference results took less than 6 milliseconds to compute on an NVIDIA GTX 1080ti GPU. This was not a case of using more expensive hardware. The GPU costs less than the CPU.



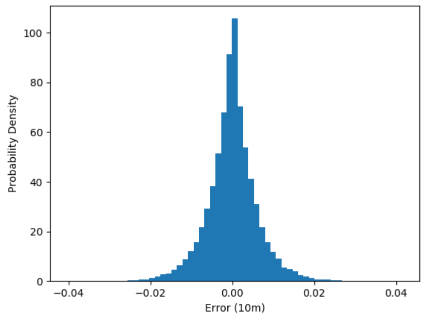
(a)



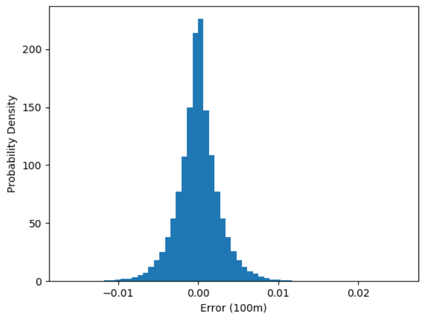
(b)



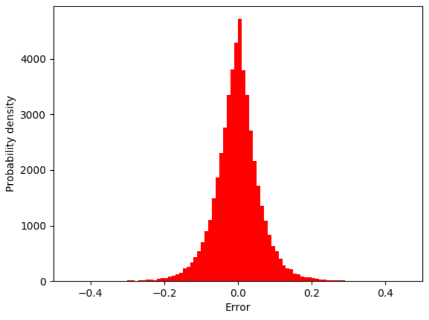
(c)



(d)



(e)



(f)

Figure 7: Error distributions over the 50,000 sample test set evaluated by traditional Monte Carlo techniques and a deep learning model. The individual figures represent differing amounts of scenarios used in each valuation: (a) 10k, (b) 100k, (c) 1 million, (d) 10 million, (e) 100 million, and (f) Deep Learning Model. The error of the deep learning model is similar to that of the Monte Carlo simulation with 100k scenarios. The MC results with 100k scenarios took a little over 10 minutes to calculate. The deep learning model calculated its results in less than 6 milliseconds.

We also investigated the samples in the tails of the histogram as they often came from the edges of the training distribution. Later work will investigate techniques to score the likelihood that a sample of test data will produce an accurate valuation with kernel density estimation and statistical measures of distance showing promise.

3.3.3 Speed

The overall performance can be broken down into the three key phases, generation of the training set, training the neural network and the final inference step.

We generated three training datasets, trading off the size of the dataset against the quality of data (ie number of Monte Carlo scenarios per sample), while holding the total number of Monte Carlo scenarios at 5x10^{12}. Each dataset took approximately 1 week to generate using all 24 cores of an AMD EPYC 7401P server.

Neural network model training is a function of the number of layers, the nodes per layer, the size of the training set, the size of mini-batch and the learning rate. The models presented in this paper were trained for between 3 hours and 1 week.

The time spent generating the training set and training the model are one-off costs. Once completed they do not have to be repeated. These phases are most accurately compared to the time spent developing traditional models. Rarely do we discuss how long it took code and test our basket MC model.

The time we are truly concerned with is the inference time: how quickly can a trained model return a valuation. This varies with the size of the model. The small models we trained (6 layers with 300 nodes per layer) were capable of returning more than 20,000 valuations in parallel in less than 50 microseconds. The large models (6 layers with 1400 nodes per layer) can compute 50,000 valuations in less than 6 milliseconds.

4 Conclusions and Opportunities for Future Work

In this paper we have demonstrated that deep neural networks can be used to provide highly accurate derivatives valuations using a basket option example. These models compute valuations approximately 1 million times faster than traditional Monte Carlo models. We have developed a successful methodology to generate a set of training data, which can be adapted to suit other valuation models. Finally we have also demonstrated that using small numbers of Monte Carlo paths in the training set is very effective and the the neural network learns to average out the random error component of the Monte Carlo model found in the training set.

We are actively investigating several areas:

1. Scaling the dimensionality of the derivatives pricing model. Some models require many hundreds of inputs across market data and trade specifics.

2. Developing scoring techniques to determine the suitability of the trained model to input data being appplied.

3. Providing risk sensitivities through back propagation of neural networks as an alternative or complement to algorithmic differentiation.

4. Using DNNs to approximate both valuation model and model calibration steps independently as a pipeline.

5. Applying transfer learning to accelerate the training of related derivative products.

6. Using DNNs in practice to accelerate derivative valuations for XVA.

7. The capacity of a model is a function of the number of layers and the number of nodes per layer. Future work will also involve changing the depth of the models while holding the nodes per layer constant.[4].

8. Exploring the impact of mini-batch size. Smaller mini-batches allow GPU memory to be used for other purposes, such as building larger models.[5]

References

Brummelhuis, R. and Z. Luo (2017). CDS Rate Construction Methods by Machine Learning Techniques. *SSRN*.

Culkin, R. and S. R. Das (2017). Machine Learning in Finance: The Case of Deep Learning for Option Pricing. *Journal of Investment Management*.

Cybenko, G. (1989, Dec). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems 2*(4), 303–314.

E, W., J. Han, and A. Jentzen (2017, June). Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *ArXiv e-prints*.

Gaß, M., K. Glau, M. Mahlstedt, and M. Mair (2015, May). Chebyshev Interpolation for Parametric Option Pricing. *ArXiv e-prints*.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Green, A. (2015). *XVA: Credit, Funding and Capital Valuation Adjustments*. Wiley.

Guss, W. H. and R. Salakhutdinov (2018). On characterizing the capacity of neural networks using algebraic topology. *CoRR abs/1802.04443*.

Henry-Labordere, P. (2017). Deep Primal-Dual Algorithm for BSDEs: Applications of Machine Learning to CVA and IM. *SSRN*.

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. *The Journal of Finance 49*(3), 851–889.

Kingma, D. P. and J. Ba (2014). Adam: A method for stochastic optimization. *CoRR abs/1412.6980*.

Lewandowski, D., D. Kurowicka, and H. Joe (2009, 10). Generating random correlation matrices based on vines and extended onion method. *Journal of Multivariate Analysis 100*, 1989–2001.

Ng, A. (2018). Neural Networks and Deep Learning. In *Coursera*.

QuantLib (2000). The quantlib home page: An open source framework for quantitative finance. http://www.quantlib.org.

Tenti, P. (1996). Forecasting foreign exchange rates using recurrent neural networks. *Applied Artificial Intelligence 10*, 567–581.

Zeron, M. and I. Ruiz (2017). Chebyshev Methods for Ultra - efficient Risk Calculations. *MoCaX Intelligence Working Paper*.

[4]Further discussion of learning capacity can be found in Guss and Salakhutdinov (2018)

[5]Early indications suggest this is not particularly important for the accuracy of the model.