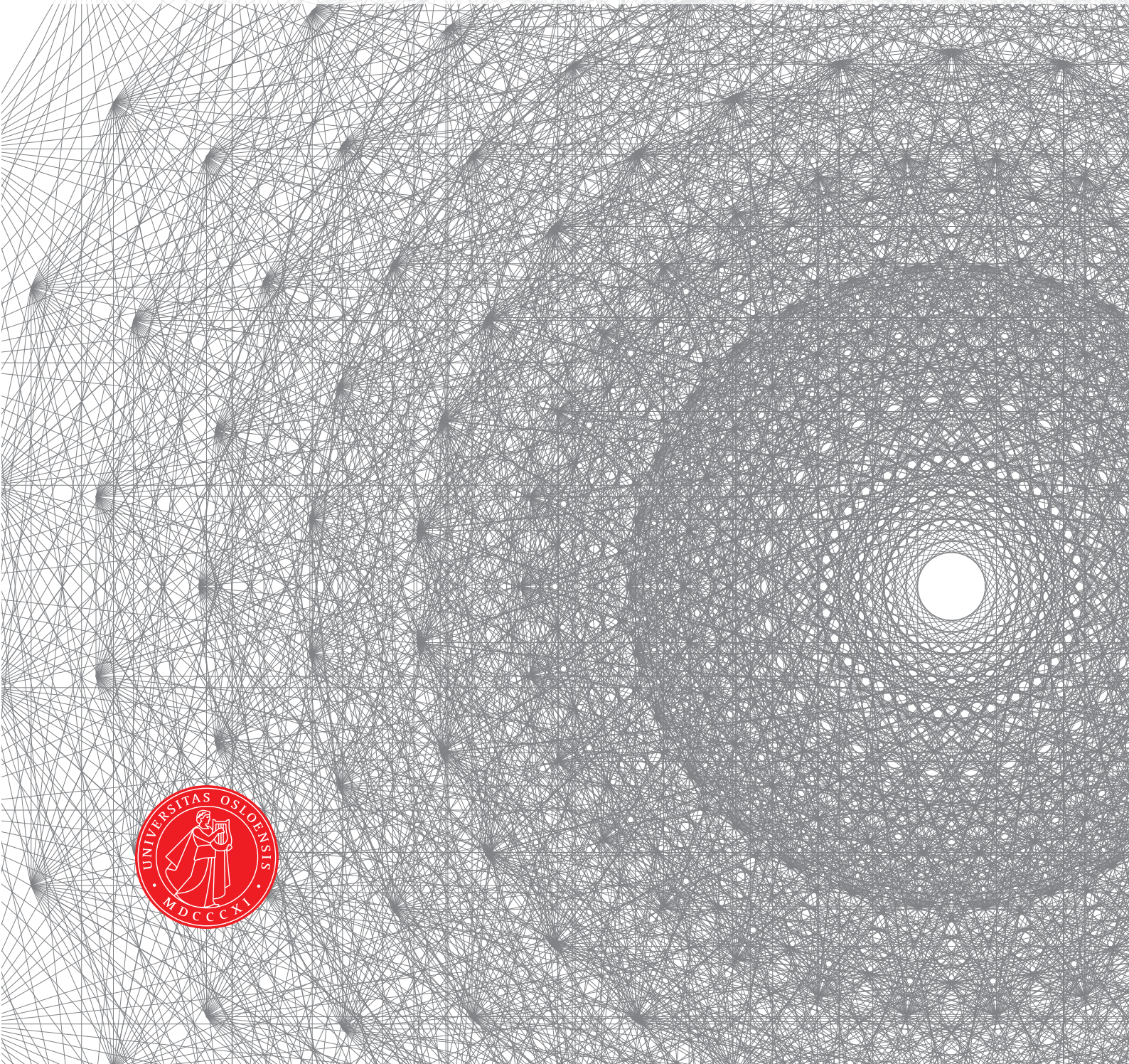
UiO : Department of Mathematics
University of Oslo

Solving PDEs Using Neural Networks

Joar Ole Sætre

Master's Thesis, Spring 2020



This master's thesis is submitted under the master's programme *Computational Science and Engineering*, with programme option *Computational Science*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 30 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Abstract

Lately, there has been a lot of research on using deep learning as an alternative method to solve PDEs. The major benefit is being able to solve in higher dimensions without using a full grid of mesh points. Here we try to implement the algorithm, without utilising any pre-existing machine learning packages, to understand how the process is done. We will try to see if we can extend solutions to higher dimensions than with an explicit finite difference scheme, even with a simple feedforward neural network. A natural application will be the Black–Scholes equation with multiple underlying assets.

Acknowledgements

First and foremost, I want to thank my supervisor, Professor Nils Henrik Risebro of the Department of Mathematics at the University of Oslo, who came up with the topic. It presented the perfect opportunity to combine mathematics with both finance and programming, as well as learn something completely new in deep learning. I have a lot to improve on in seeking guidance and giving updates, and therefore also wish to express gratitude for letting me see where the study lead me, but knowing that any questions would be welcome at any time.

Secondly, I want to thank my girlfriend, Archana, for proof reading and excellent feedback on improving both language and structure of the thesis.

Finally, I sincerely would like to recognise the love and support of my family, especially my mum, Anne Lise, who kept me motivated by showing so much pride in my not yet even achieved accomplishments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial intelligence, more specifically machine learning, is a concept where a computer can "learn" a structure or pattern. It is called intelligence because it works similarly to the human brain in the way that by being exposed to a set of examples the computer can find the right pattern by trial and error. First, it uses it's initial guess of the structure on the examples and then it is penalized more the farther off it is from the target. Next, it updates the parameters and tries again. After the error is small, the pattern is said to have been learned. Hopefully when exposed to a new example not in the training set, the AI will now be able to accurately predict the result.

There are two main groups of machine learning algorithms: supervised learning and unsupervised learning. Unsupervised learning is when the goal is to find a structure in the data. An example is finding a way to group seemingly random examples into categories, called clustering.

Supervised learning, which this work will use, is when there are labeled input that lead to labeled output. In other words switching the order of examples also switches the order of the output. The goal here is to learn this connection between input and output. Examples are regression and solving equations.

Traditionally, when analytical solutions to PDEs are not available, they have been solved by numerical methods such as finite difference or finite element methods. When working in several dimensions, however, such methods become infeasable. The memory and computations required are simply too large. Imagine a finite difference scheme to find a solution of some PDE where the domain is the unit square in two dimensions. The domain is discretised into 100 meshpoints each direction. For every iteration the solution has to be calculated at 10,000 points. No problem. Now make the domain the unit cube in three dimensions and with similar discretisation. The solution now has to be calculated at 1,000,000 points, using roughly 8MB of the computers RAM if using numpy arrays of floats, to store only the solution. Extending to four dimensions it becomes 100,000,000

mesh points with 800MB RAM to store the solution, not to mention the enormous amount of calculations. It is easy to see the problems here, e.g. a similar five dimensional array needing 80GB RAM! Since machine learning uses a random sample of points, adding one more dimension does not equal the same increase in meshpoints required. Thus, the main advantage of machine learning is to extend solutions to higher dimensions. For further reading on artificial intelligence in general see [2].

Chapter 2

Motivation

In this chapter the main ideas behind machine learning will be presented. Examples of modifications for other purposes than PDEs will be explained, although not in great detail. In the next section some theory on the equations solved in chapter 4 will be presented.

2.1 Overview of the Method

2.1.1 Network Architecture

Machine learning uses networks of "neurons" to learn. A dense feedforward neural network is the simplest example. This is a network made up of an input layer, a varying amount of hidden layers and an output layer. Each layer has nodes (neurons) which are connected to every node in the previous layer as well as every node in the next layer. In a network which is not dense it is possibleto have connections to only a select few other nodes. Figure 2.1.1 shows a neural network with two hidden layers. The number of nodes in each layer is referred to as the network's *width*, whereas the number of hidden layers is referred to as the network's *depth*. A *deep* neural network is a network with two or more hidden layers, thereof the term *deep learning*. Generally a network's performance increases with both width and depth, but because of the computational cost of increasing width, an increase in depth is to be preferred if possible.

A node in a feedforward neural network takes the sum of a weighted input vector $\mathbf{x}$, adds a bias b and uses an activation function $\sigma(x)$. The output of this function is then sent to the next layer as input.

$$z = \sigma\left(\sum_{n=1}^{N} x_n w_n + b\right) \tag{2.1}$$

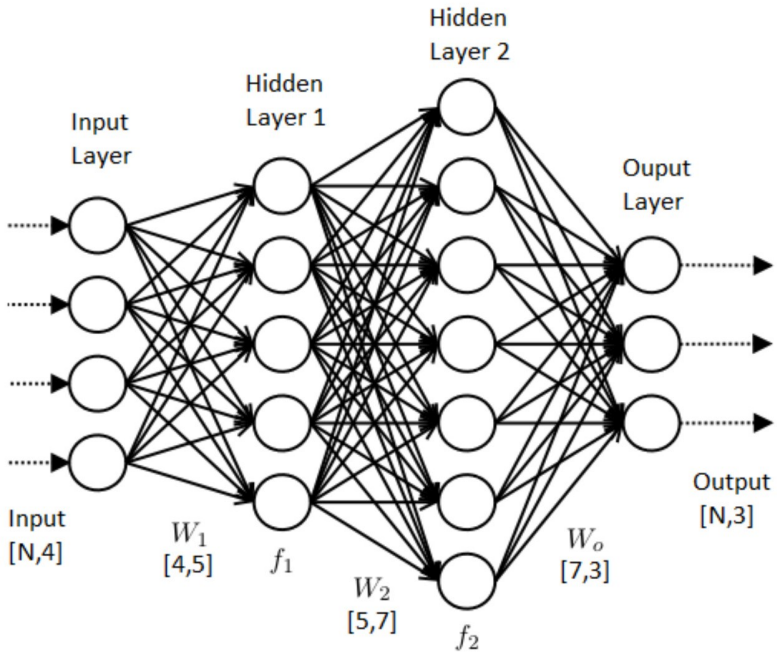


Figure 2.1.1: Dense feedforward neural network with two hidden layers of width 5 and 7. Input has dimension 4 and output has dimension 3. At the bottom the dimensions of weight matrices are shown as well as dimensions of input and output matrices, where N in this case is the number of examples. (Source: `https://dzone.com/articles/the-artificial-neural-networks-handbook-part-1-1`)

where N is the width of the previous layer. The activation function most widely used is the rectifier, or rectified linear unit (ReLU)

$$\sigma(x) = \max(0, x) \tag{2.2}$$

with the identity function $\sigma_0(x) = x$ in the output layer. In other tasks such as classification one can also use the logistic function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

with softmax in the output layer:

$$\sigma(x_n) = \frac{e^{x_n}}{\sum_{n=1}^{N} e^{x_n}} \tag{2.4}$$

How large the error is relative to the target is determined by a loss function $J(\theta)$ where θ is the set of parameters, here consisting of all weights and biases.

The most widely used, and which will be used here, is the mean squared error

$$J(\theta) = \frac{1}{M}\sum_{m=1}^{M}(f^*(x_m) - f(x_m;\theta))^2 \tag{2.5}$$

where M is the number of examples, $f^*(x_m)$ the target and $f(x_m;\theta)$ the network output given current parameters. When solving equations one has to deal with initial and boundary conditions. These can be added in the loss function:

$$\begin{aligned} J(\theta) = &\frac{1}{M_1}\sum_{m_1=1}^{M_1}\left(f^*(x_{m_1}) - f(x_{m_1};\theta)\right)^2 \\ &+ \frac{1}{M_2}\sum_{m_2=1}^{M_2}\left(g(x_{m_2}) - f(x_{m_2};\theta)\right)^2 \end{aligned} \tag{2.6}$$

where $g(x)$ is some boundary condition, M_1 is the number of examples in the interiour and M_2 is the number of examples on the boundary.

2.1.2 Parameter Update

The error is used in updating the parameters. The weights and biases that contributed the most to the error will also change the most. A gradient based optimisation is the standard approach.

$$\theta_{i+1} = \theta_i - \eta\nabla_{\theta_i}J(\theta_i) \tag{2.7}$$

where η is the step size or *learning rate*.

Backpropagation

Finding the gradient of the loss function is done by backpropagation. The forward pass of the network is simply a composition of all the activations:

$$f(X) = \sigma_L(...\sigma_2(\sigma_1(XW_1 + \mathbf{1}_M b_1)W_2 + \mathbf{1}_M b_2)...W_L + \mathbf{1}_M b_L) \tag{2.8}$$

where $X \in \mathbb{R}^{M\times d}$ is the input, W_l denotes the weight matrix from layer $l-1$ to layer l and $\mathbf{1}_M \in \mathbb{R}^M$ with all elements being 1. Knowing this the gradient of $J(\theta)$ w.r.t. all weights and biases is found by applying the chain rule and going backwards in the network. Let the error in the output layer L w.r.t. the inputs to that layer be

$$\delta_L = \nabla_{Z_L}J(\theta) \odot \sigma'(Z_{L-1}W_L + \mathbf{1}_M b_L) \tag{2.9}$$

where Z_l is the output from each layer, $Z_0 = X$ and $\odot$ is element wise multiplication. Since the output layer uses an identity activation the last derivative is just 1. To propagate the error backwards in the network calculate

$$\delta_l = \delta_{l+1} W_{l+1}^T \odot \sigma'(Z_{l-1} W_l + \mathbf{1}_M b_l). \tag{2.10}$$

Because ReLU activation is used, the derivative is still simple: either 0 or 1.

Now finding the gradient of the loss function is straightforward.

$$\frac{\partial J(\theta)}{\partial b_l} = \delta_l \tag{2.11}$$

$$\frac{\partial J(\theta)}{\partial W_l} = \sigma(Z_{l-1} W_l + \mathbf{1}_M b_l)^T \delta_l \tag{2.12}$$

More details can be found in [8, chap. 2].

Stochastic Gradient Descent

With a large number of examples, however, standard gradient descent takes far too much time to run. A popular extension is *stochastic gradient descent*. True stochastic gradient descent is where only the gradient at one random example is used, instead of using the gradient at all examples. The parameters are first updated after the gradient at the random example is backpropagated. Next, the gradient is found at a new random example until the whole set of examples has been run through. Now it is said the algorithm is done with one *epoch* and it continues for how many epochs are desired. This type of convergence turns out to be too noisy, so a middle ground is instead the standard approach called mini-batch gradient descent.

The term stochastic gradient descent (SGD) is mostly understood to mean mini-batch gradient descent. Here the examples are divided into random subsets (mini-batches) and the gradient is calculated using one batch at a time. The parameters are updated and the algorithm is continuing to the next mini-batch. After an epoch is completed the randomisation is redone so that the mini-batches are not the same as those last epoch. With mini-batch size set to one example, this method reduces to true stochastic gradient descent. if batch size is equal to sample size we have only one batch and are back to the original full-batch gradient descent. [10] has more details on stochastic gradient descent and how to implement it with Python.

2.1.3 Testing

When the network is finished learning, it is time to test the result on unseen data. It is, after all, the network's ability to predict based on previous experience we

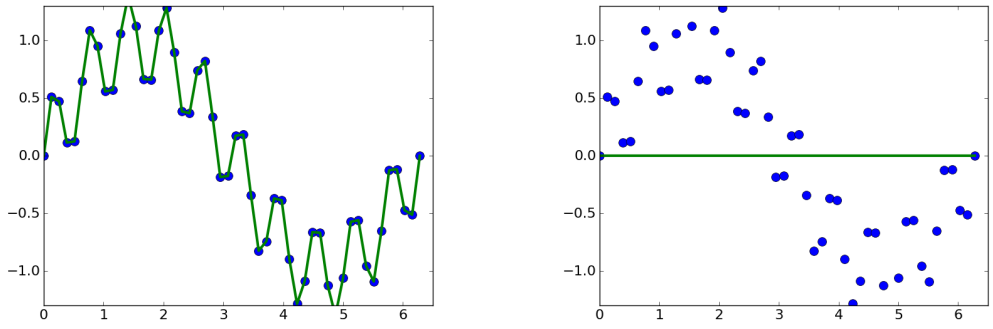


Figure 2.1.2: Overfitting on the left and underfitting on the right.

want to be as accurate as possible. In this scenario, that translates to knowing the solution to the PDE, not only on training points, but also arbitrary points in the given domain. Usually part of the training data should be set aside and used for testing, however, in solving equations there are generally no problems in generating new points in the same domain. To do the test, simply run the new set of points through the same network and calculate the loss. Hopefully test loss is close to the loss found when training.

In most machine learning applications, one has to deal with cases of underfitting and overfitting, which leads to poor generalisation. Overfitting is when the training loss is too small such that in, say, regression, the curve captures all the noise. Where the curve goes in the continuation also has no clear solution. Underfitting, on the other hand, is when the curve is too general as an approximation and the solution becomes useless. Both cases at an arbitrary point in the domain has a high chance of being inaccurate. Figure 2.1.2 is an illustration of overfitting and underfitting. In solving PDEs this is not as big of a problem since inputs are points in a domain we wish to find a solution for and not some noisy observation data.

If the training loss is very close to zero but test loss is not, overfitting can be one of the causes. In the case of PDEs, the solution is then usually too much of an approximation due to small sample size. In other applications with noise, regularisation can help solve overfitting. An other cause of poor generalisation can be related to the boundary and boundary conditions.

2.2 Theory on PDE

2.2.1 The Black–Scholes Equation

Consider a risky asset and a risk-free rate of return, usually a government bond or money in the bank. The risky asset and risk-free rate of return are assumed to have price evolutions following

$$\frac{dS_t}{S_t} = \mu dt + \sigma dW_t \tag{2.13}$$

$$\frac{dB_t}{B_t} = r dt \tag{2.14}$$

where S is the risky asset price, μ it's drift, σ it's volatility, W_t a brownian motion under $\mathbb{P}$, B the risk-free bank account and r the risk free rate of return (interest rate). Let $V(S_t, t)$ be the value of an option on the risky asset. The function $V(s,t)$ must satisfy the Black–Sholes PDE

$$\begin{cases} \dfrac{\partial V}{\partial t} + \dfrac{1}{2}\sigma^2 s^2 \dfrac{\partial^2 V}{\partial s^2} + rs\dfrac{\partial V}{\partial s} - rV = 0, \\ V(s,T) = G(s), \end{cases} \tag{2.15}$$

where the terminal condition $G(s)$ is a pay-off at time of maturity T. In the special case where the option is a European call $C(S_t, t)$ giving terminal condition

$$C(S_T, T) = \max(0, S_T - K), \tag{2.16}$$

where K is the agreed price at maturity T (strike/exercise price), the equation has an exact solution found by the formula:

$$C(S_t, t) = S_t \Phi(d_1) - K e^{-r(T-t)} \Phi(d_2), \tag{2.17}$$

$$d_1 = \frac{1}{\sigma\sqrt{T-t}} \left[\log\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t) \right], \tag{2.18}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}, \tag{2.19}$$

where Φ is the cumulative standard normal distribution. The Black–Scholes equation can also be transformed into the heat equation.

To reduce risk, it is normal to put together a portfolio of e.g. several different options. This portfolio will also satisfy equation 2.15. No exact solution formula exists for such portfolios. In practice they are solved with a weighted average of asset values, thus avoiding the tremendous increase in computing power required when simply extending numerical methods to all dimensions. In fact the method is the same as when calculating the value of a basket option; an option on a collection

of underlying assets where the strike condition depends on a weighted average of the asset prices at maturity. An approximation like this can work assuming linearity. If we want to include, say, strike conditions on individual assets we need to calculate them separately.

2.2.2 Feynman–Kac Formula

Consider a linear parabolic PDE of the form

$$\begin{cases} \dfrac{\partial u}{\partial t} + \ \mu(x,t)\dfrac{\partial u}{\partial x} + \ \dfrac{1}{2}\sigma^2(x,t)\dfrac{\partial^2 u}{\partial x^2} - \ h(x,t)u + f(x,t) = 0, \\ u(x,T) = \psi(x), \end{cases} \tag{2.20}$$

for all $x \in \mathbb{R}$ and $t \in [0,T]$. The Feynman–Kac theorem says that the solution of this equation can be written as:

$$u(x,t) = \mathbb{E}^Q \left[\int_t^T e^{-\int_t^T h(X_\tau,\tau)d\tau} f(X_r,r)dr + \ e^{-\int_t^T h(X_\tau,\tau)d\tau}\psi(X_T)|X_t = x \right], \tag{2.21}$$

where

$$dX_t = \mu(X_t,t)dt + \sigma(X_t,t)dW_t^Q, \tag{2.22}$$

and Q is a probability measure. In other words the solution of a deterministic PDE as above can be found as an expectation of a stochastic process. This way Monte Carlo methods can be used to simulate solutions. The classical formula is limited to linear parabolic PDEs, but extensions can be made to allow for more complex equations to be solved. An example is [9, sec. 2B, p. 3] where they use a nonlinear formula to solve systems of forward-backward stochastic differential equations with deep learning.

Chapter 3

The Algorithm

In this chapter the algorithm will be looked at in more detail. Along the way problems encountered while creating the AI will be mentioned and the solutions found to these will be discussed. There exist premade libraries like TensorFlow, Keras, PyTorch and Theano to help build the program, but in this work the algorithm is coded from scratch to better understand all the parts of the learning process. The resulting machine learning is, of course, less efficient than the standard and also is limited on the complexity of network structures that can be used. The lecture notes of Stanford University's computer science course *CS231n Convolutional Neural Networks for Visual Recognition* [4, Module. 1: Neural Networks] is used as a starting point.

3.1 Initialisation

The first step is the initialisation of the weight matrices and bias vectors. The elements of the weight matrices need to be random numbers to avoid them updating equally at every iteration. If the weights are initially too far away from their target value the gradient will be too large. This can lead to blow-up. Instead, then, a normal distribution with mean zero and small variance seems good since one can expect zero to be the average weight value. With an increasing number of examples, however, the variance of a neuron's output will also increase. To normalise it is shown in [4, module 1: NN Part 2] that dividing by $\sqrt{n}$ where n is the number of inputs to the specific neuron solves the problem. This kind of initialisation is called *Xavier* initialisation. One of the assumptions is that the expectation of inputs to a neuron is zero, which is not the case for ReLu-activation used here. In [3, p. 5] they therefore derive that multiplying by $\sqrt{2/n}$ should instead be used. The importance of good initialisation was found to be crucial, moreso than expected: Throughout most of the work done here the weights were

simply set to be normaly distributed with zero mean and 10^{-4} variance. It seemed impossible to have more than two hidden layers without getting the gradient to either blow up or be close to zero, but the cause was thought to be something else. To compensate for only having two hidden layers we had to use 50 to 100 nodes. When the above suggested initialisations were implemented we could right away increase hidden layers to 20 and reduce nodes to 10 per layer without any immediate problems. This is without considering accuracy, how much further we could increase or if an increase is even beneficial.

Biases can be all set to zero, but from experience it seems here to work better with a small positive number at the start to make sure every node is activated.

3.2 Regularisation

There is a chance to overfit the solution after training. In addition, when training the network, we want to approach a global minimum. In practice we cannot know whether it is a global or local minimum we are approaching, but most likely it is a local one. At the start of the coding, the solution usually ended up being straight and a poor fit. It seemed as if most of the neurons were "turned off", having value zero and stopped updating. The result was only a select few weights having any impact on the outcome. A regularisation in the loss function tried to solve these problems by penalising large weight values such that many small (different from zero) weights are prefered to one large. The modification of equation (2.6) used is

$$\begin{aligned} J(\theta) = &\frac{1}{M_1}\sum_{m_1=1}^{M_1}\left(f^*(x_{m_1}) - f(x_{m_1};\theta)\right)^2 \\ &+ \frac{1}{M_2}\sum_{m_2=1}^{M_2}\left(g(x_{m_2}) - f(x_{m_2};\theta)\right)^2 \\ &+ \frac{\lambda}{2}\sum_{l=1}^{L}\sum_{j,k}(W_l)_{jk}^2 \end{aligned} \tag{3.1}$$

where at the end we add the sum of all squared elements in the weight matrix of each layer. The regularisation strength λ is set to 0.001. If λ is increased it influences the loss too much.

After improving the initialisation such that a deeper network could be utilised and using momentum, this kind of regularisation seems to slow down the training rather than being of any benefit. When stopping the training at different times earlier than necessary it looks like the network tries to fit the curve from one end to the other instead of at every point simultaniously. Large weights are therefore needed to more quickly run through the training process.

In learning derivatives, the greatest problems are sensitivity to small changes. No change at all, ending in a trivial zero solution, or rapidly blow up seemed to be the only outcomes. An effective fix is to set a hard upper limit on the values the weights can take, a max regularisation. Choosing e.g. the value 10 leads to excessively stochastic behaviour in one dimension and simply a slower blow up in more dimensions. A limit of 3 slows training way too much down. In some cases it even leads to inability to learn. A middle ground of 5 will be used.

3.3 Momentum

When observing the evolution of the loss during training it very often seems to be areas where the parameters struggle to update more than small steps at a time or oscillates around a poor local minimum. The reason could be curvature in the optimisation path large enough compared to the step size to get stuck. Finding a learning rate perfect for all such areas is unrealistic. Using a method called *momentum* is an improvement where a moving average of previous gradients is used in the calculation of the next. This can be thought of as a ball rolling down a hill. In case it encounters a hole in it's path, the velocity is high enough to keep rolling through as long as the hole is not so wide that it rolls back down into it. Using momentum also helps in reducing overly stochastic convergence. Let v be a parameter for velocity and the previous discussed parameter update becomes

$$v_i = \beta v_{i-1} - \eta \nabla_{\theta_i} J(\theta_i), \tag{3.2}$$

$$\theta_{i+1} = \theta_i + v_i, \tag{3.3}$$

where the constant β denotes the weight of the moving average i.e. how much of previous velocity is kept. More details can be found in [2, chap. 8, p. 293] in addition to an improved method called *Nesterov momentum* where the gradient used in (3.2) is calculated with the current weighted velocity. Even more methods exist, with Aadam (Adaptive momentum) [5] being the most popular.

3.4 The Derivative

By far the most difficult part of learning a differential equation is the derivative. Without it all the function fitting is easy, straightforward and can be done with the simplest form of network. The program here is written from scratch without the help of premade AI packages, so from the get go we are limiting the possibilities. The goal is instead to see how far we can get with a dense feedforward network.

In solving parabolic equations that can be written in the form of the Feynman–Kac solution formula 2.21, the derivatives in the loss function can be avoided

altogether. The downside is, of course, that each iteration requires numerous simulations to get an accurate enough expectation. If, say, a hundred realisations are used and each of those involves integrals, in addition to each epoch potensially taking a long time to begin with due to a large training set, computing time can take quite a while. This is especially true with a non-zero f together with non-constant h in the Feynman–Kac formula. These types of parabolic equations pose few other problems than training time.

In this work, a general method to solve PDEs of low complexity is sought after. A way of directly learning the differential operators was therefore looked for. In [1, p. 6] it is suggested a method to directly find the derivatives through the forward pass. The first layer's derivatives are readily available analytically and so for every next layer the value is updated to the current layer until the output. In trying to implement it something kept going wrong so a new way had to be found.

A simple, easy to implement and familiar approach is to use a finite difference method. A full grid of meshpoints is still not required, since we only need the finite differences at the training examples. If, say, a first order ODE is to be solved with a simple forward difference the inner part of the loss function will become

$$f^*(x_m) - \frac{f(x_m + h; \theta) - f(x_m; \theta)}{h} \tag{3.4}$$

where h is the step size. With a dense feedforward network it proved extremely difficult in more than one dimension due to the sensitivity of small changes in output.

3.5 Hyperparameters

When reading about machine learning, one often come across the term *hyperparameters* that can sound advanced. These are simply the parameters that need to be calibrated but are not found through the learning algorithm itself. The hyperparameters here are listed in table 3.1. Some of these, like learning rate, regularisation strength and constant of momentum, can be turned from hyperparameters to learnable parameters. By implementing a nested learning algorithm, where these are in a level above the original, the best performing values can be found in the same way as the problem solution. It takes, however, much more computing power and requires a lot more time to run.

A more simple and time saving approach is by trial and error. In choosing this way, one can also easily use non-constant values. The learning rate can e.g. be decaying by a factor of two every thousand epochs. In the continuation a hard coded learning rate schedule is chosen; the initial rate is reduced by specified factors at preselected values of the loss. The different loss values and factors are found through trial and error and varies between the problems.

Parameter		Value used
η	Learning rate/step size:	10^{-4} to start
M	Sample size:	Varies
M_∂	Number of boundary points sampled:	$0.1M$ to $0.2M$
m	Batch size:	64
λ	Regularisation strength:	10^{-5} (mostly 0)
β	Constant of momentum:	0.9
h	Step size of finite differences:	10^{-3}

Table 3.1: Hyperparameters.

Chapter 4

Results

All simulations are done on a laptop from January 2013 with an Intel Core i7-3630QM CPU @2.40GHz and 8GB RAM. No GPU acceleration is used and no parallell computing. Couple this with intermediate programming skills, and the listed training times are therefore included only to be considered relative to each other.

4.1 Sines

To start off we do some simple curve fitting to see how the training works. In figure 4.1.1 a simple sine is learned, but training is stopped prematurely. It looks like the training goes along the x-axis from left to right instead of trying to fit all points at the same time. The takeaways here are that the boundary should be sufficiently sampled, extend training to outside of the boundary if possible and that even premature stopping can visually lead to a good sense of the solution behaviour in parts of the region. Sufficiently sampled boundary on all sides will increase the importance of these areas in determining the loss. Extending the training domain will do the same in the way that a good fit outside will need boundaries to have small loss.

Similary when solving equations we start small. The first equation is the ODE

$$\begin{cases} \dfrac{du}{dx} = \cos(x), \quad x \in [0, 10], \\ u(0) = 0, \end{cases} \tag{4.1}$$

with solution $u(x) = \sin(x)$. In figures 4.1.2 and 4.1.3 are two plots of loss functions: the first with a network consisting of eight hidden layers with twenty nodes each and the second with a network consisting of two hidden layers with a hundred nodes each. We can see that the deeper network reaches the same loss in 10,000

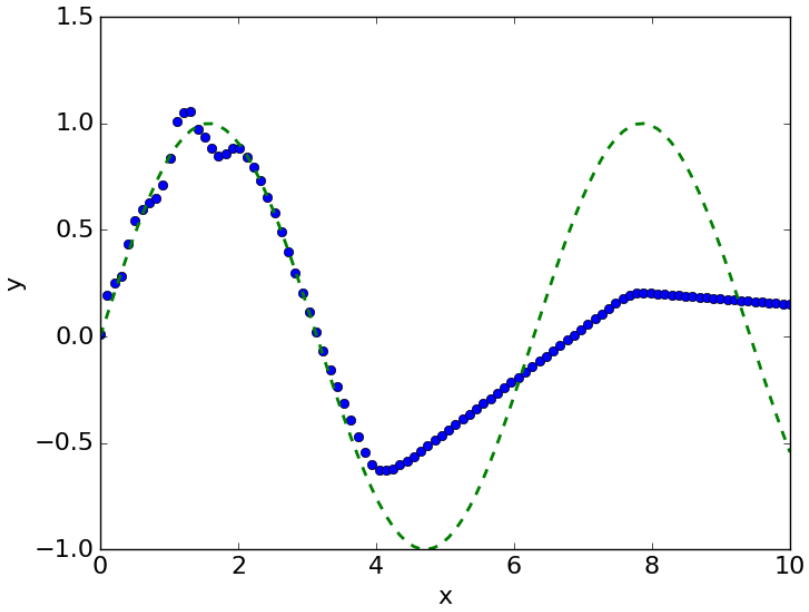


Figure 4.1.1: Early stopping of $u(x) = \sin(x)$.

epochs as the shallower one does in three times as many. The deeper network architecture will be used as the standard going forward, however, however it's convergence is more stochastic. When derivatives get very small and has to propagate through many layers, a case of vanishing gradients can arise. The network of two hidden layers with a hundred nodes each will therefore sometimes later be tested.

Even though overfitting is not a problem encoutered here, overtraining certainly is. Figure 4.1.5 shows what can happen when training for too long. After around 25,000 epochs something happens which the AI needs to correct. It certainly could be that a local minimum was escaped from, but figure 4.1.4 shows the solution. It looks as if some nodes in the network has been set to zero and stopped updating, leading the rest of the nodes to compensate. Notice the number of epochs, 30,000, is the same number the shallower network needed to converge. Having hundred nodes in each layer can be a reason "turning off" nodes is not as noticeable, which is another reason to consider testing this network sometimes.

The reasons sines were used were to have readily available analytical solutions and to have the same range of values for all orders of derivatives. The steps going forward were planned to be trying two dimensions and second derivatives, then later extend to Laplace's and Poisson's equations in more than three dimensions. As described in section 3.4 the derivative is difficult with a simple feedforward network and two dimensions with first derivatives proved too much.

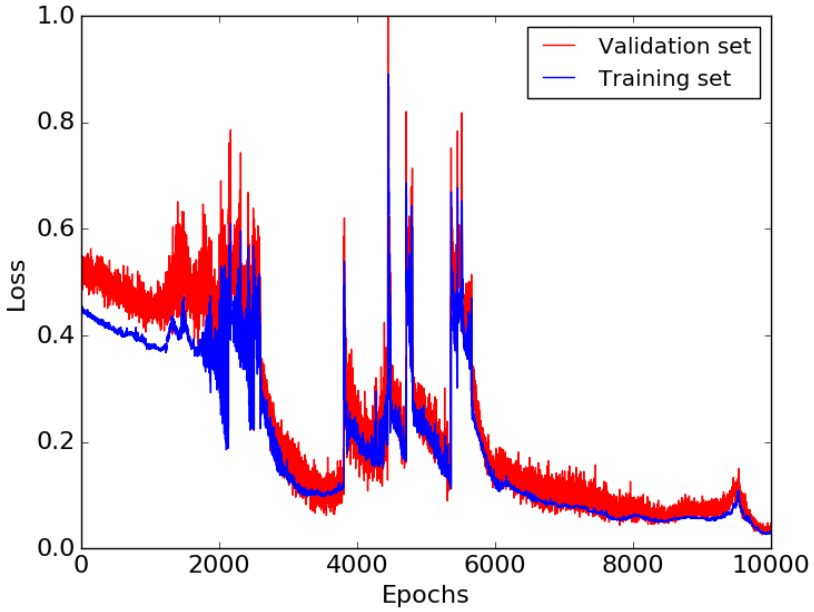


Figure 4.1.2: Loss function of $u'(x) = \cos(x)$, 8 hidden layers with 20 nodes each.

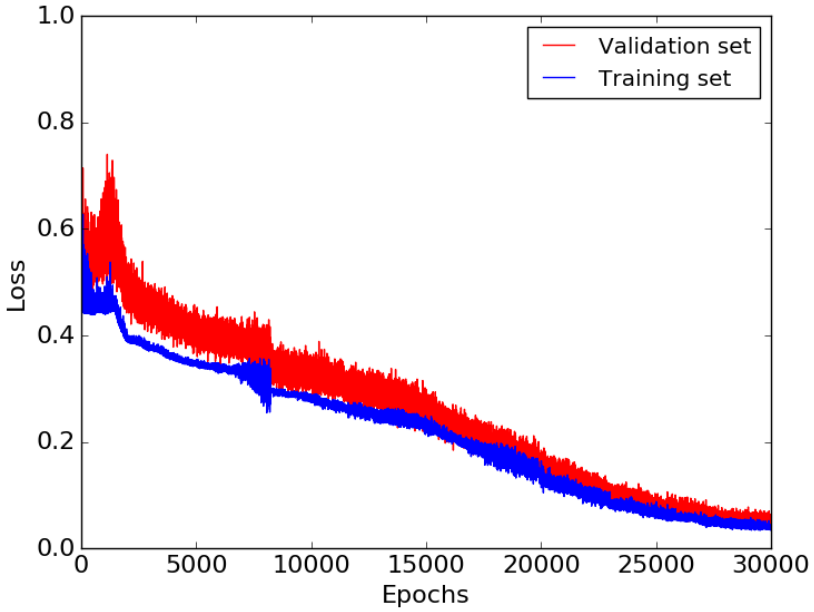


Figure 4.1.3: Loss function of $u'(x) = \cos(x)$, 2 hidden layers with 100 nodes each.

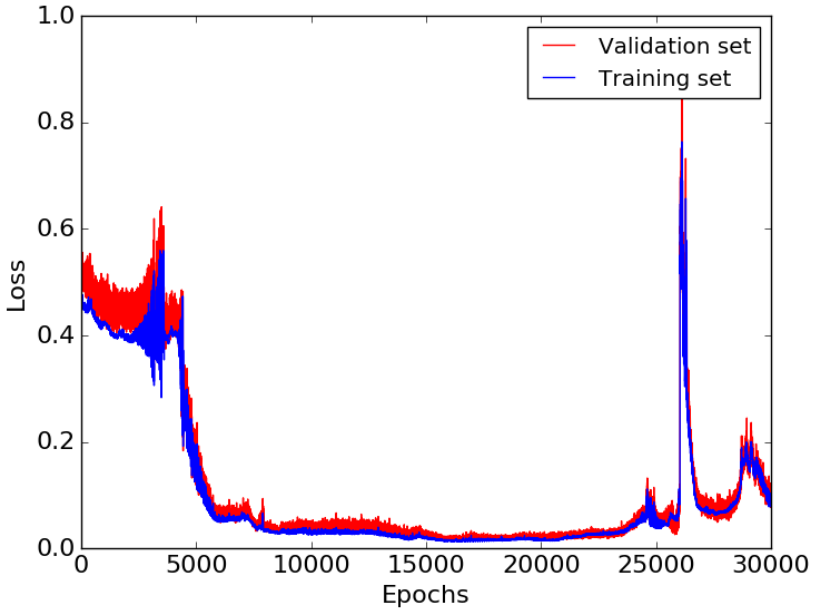


Figure 4.1.4: Loss function of $u'(x) = \cos(x)$, 8 hidden layers with 20 nodes each.

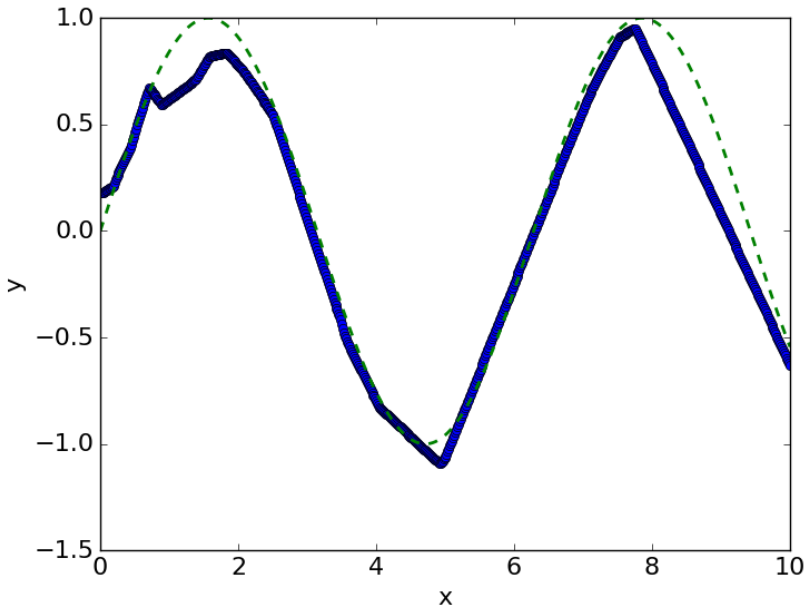


Figure 4.1.5: Solution of $u'(x) = \cos(x)$, 8 hidden layers with 20 nodes each.

Now there are two options. Either a more complex network is required, in which case premade libraries will have to be used, or we can turn our attention towards the Feynman–Kac formula. The latter is chosen, although it limits the types of equations that can be solved.

4.2 Black–Scholes

Consider the Black–Scholes equation for a European call option $C(S_t, t)$

$$\begin{cases} \dfrac{\partial C}{\partial t} + \dfrac{1}{2}\sigma^2 s^2 \dfrac{\partial^2 C}{\partial s^2} + rs\dfrac{\partial C}{\partial s} - rC = 0, \quad (s,t) \in \mathbb{R}_+ \times [0,T], \\ C(S_T, T) = \max(0, S_T - K). \end{cases} \tag{4.2}$$

The numerical solution found using the Feynman–Kac formula becomes

$$\begin{aligned} C(s,t) &= \mathbb{E}^Q \left[e^{-\int_t^T r d\tau} \psi(S_T) | S_t = s \right] \\ &= \mathbb{E}^Q \left[e^{-r(T-t)} \psi(S_T) | S_t = s \right], \end{aligned} \tag{4.3}$$

which can be seen as just the expectation of the option price at maturity discounted to current time under a risk neutral probability measure. The underlying asset price following

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{4.4}$$

is simulated using the Euler–Maruyama method

$$\begin{cases} S_{t+1} = S_t + rS_t \Delta t + \sigma S_t \Delta W_t, \\ S_0 = s. \end{cases} \tag{4.5}$$

It is clear that a constant interest rate r and a constant volatility σ speed up calculation time significantly, especially since every single epoch requires many realisations to get a sufficiently accurate expectation. The term f in the Feynman–Kac formula (2.21), not involving the function to be solved for itself, would take an even greater toll on the simulation time.

Below we start with the data values listed in table 4.1 and 100 realisations of the assets with 100 time steps are simulated to get the expectations. Starting learning rate is increased to $5 \cdot 10^{-4}$ since no derivatives makes the convergence more stable.

4.2.1 One Asset

Black–Scholes with one asset (two dimensions with time) pose no difficulties. Figure 4.2.1 shows the asset price, option price using DL and the analytic solution.

M	Training examples:	640
M_∂	Boundary examples:	$0.1M$
	Sampling domain:	$[0, 10]^2$
r	Interest rate:	0.05
σ	Volatility:	0.15
K	Exercise/Strike Price:	3.0
S_0	Initial Asset Price:	2.5
T	Maturity time:	10

Table 4.1: Data values to start with Black–Scholes.

To better distinguish the two solutions visually figure 4.2.2 does not include the asset price. The training time was 8m:44s for 250 epochs and figure 4.2.3 shows the loss function.

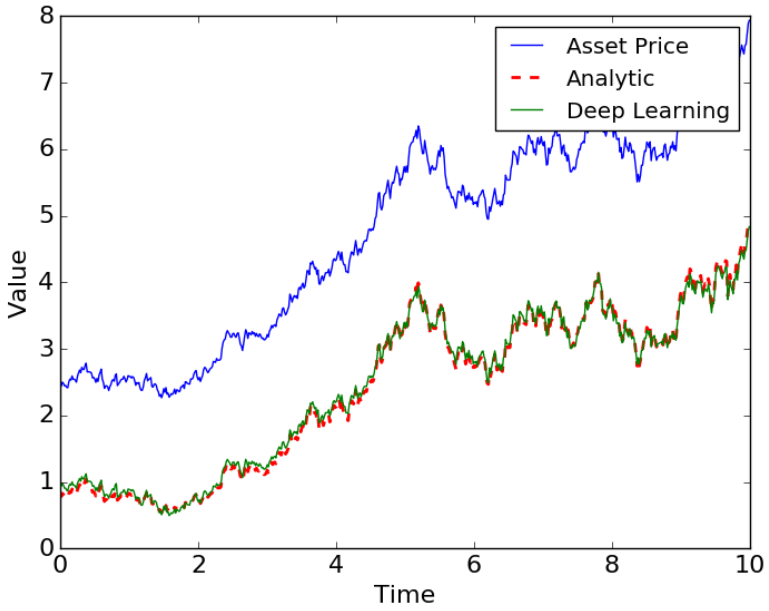


Figure 4.2.1: Black–Scholes with one asset. Training time: 8m:44s.

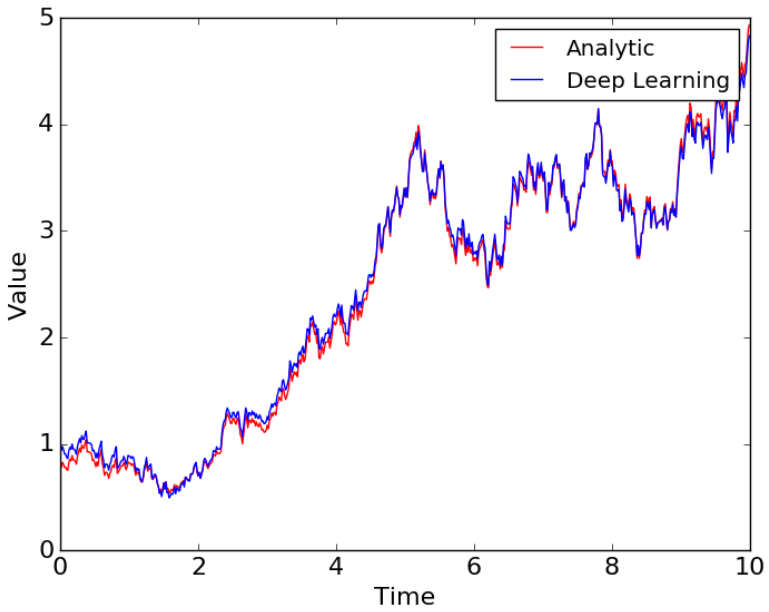


Figure 4.2.2: Black–Scholes with one asset, closeup.

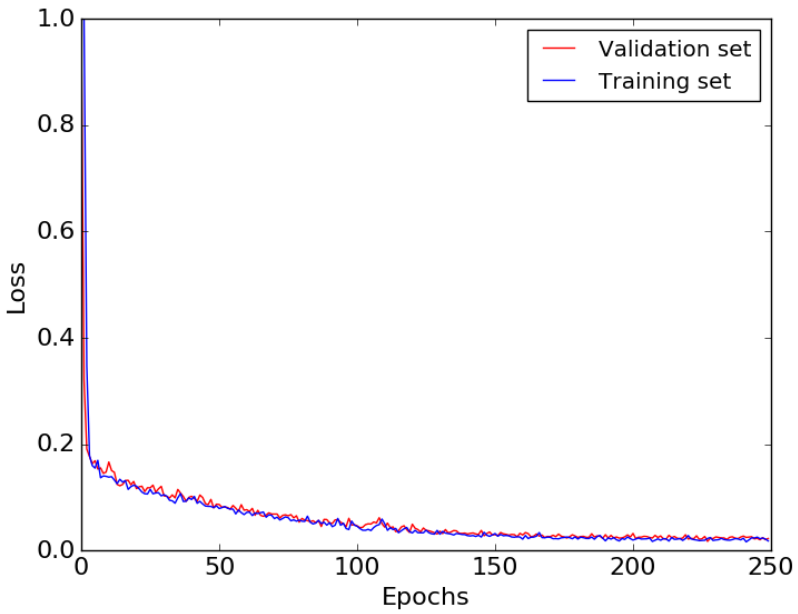


Figure 4.2.3: Black–Scholes with one asset, loss function.

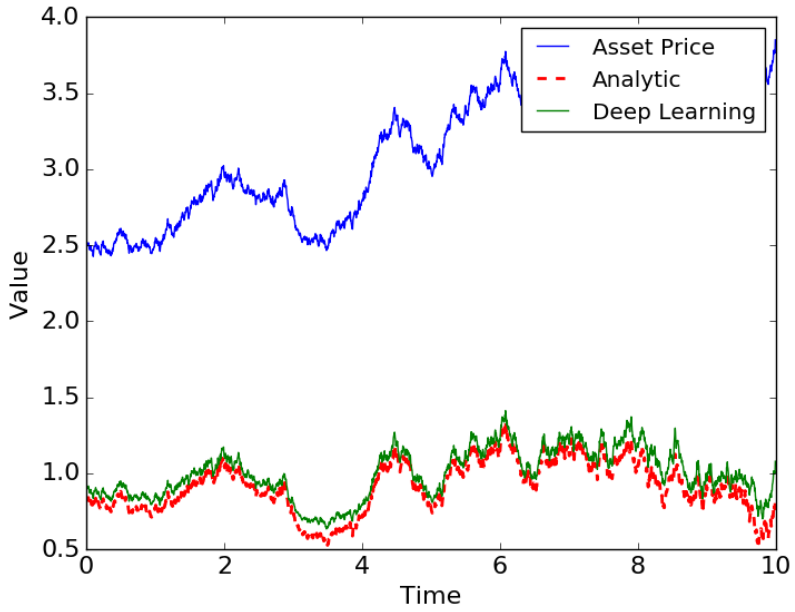


Figure 4.2.4: Black–Scholes with three assets, 2,560 examples and 500 epochs. Training time: 2h:25m:28s.

4.2.2 Three Assets

As increasing dimensions is the main benefit of deep learning we now solve in four dimensions instead of two. In the context of finance this means we have a portfolio with three assets instead of one. The analytic solution of a basket option on three assets should be the same and is used for validation since all assets are assumed similar with zero correlation.

First of all, using 640 examples worked with one asset but not with three. If one were to use a finite difference scheme with the same density in mesh points one would need around 16,190 examples with two assets and 409,586 with three assets. Here we first try 2,560 examples and 500 epochs to see how best to increase accuracy while keeping training time as low as possible. Secondly looking at figure 4.2.1 it seems very unlikely an asset value will get greater than 7.0, so the sampling domain will be changed to $[0, 7]^3 \times [0, 10]$. Lastly the boundary where asset prices tend to zero will be more heavily sampled: we increase total boundary examples to about $0.2M$.

Figure 4.2.4 shows the result after a training time of 2h:25m:28s. The result is poor so we look at the loss in figure 4.2.5 to perhaps get an idea of how to improve accuracy.

The loss function rapidly gets below 0.05, but even with all the time after to

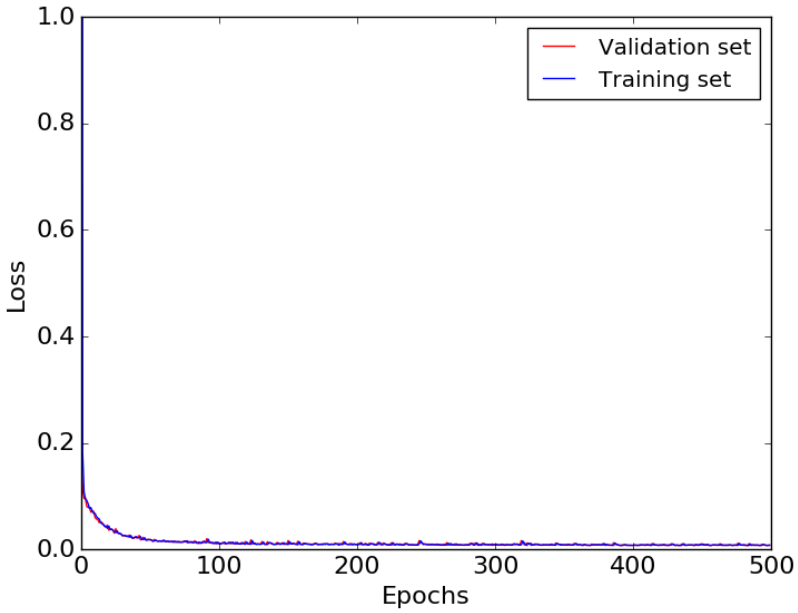


Figure 4.2.5: Black–Scholes loss with three assets, 2,560 examples and 500 epochs.

fine-tune parameters the fit does not look too good. It could be because eight hidden layers make the gradient vanish when updates are small. For this reason we try the shallower network of two hidden layers with a hundred nodes per layer. Figure 4.2.6 shows the result that took 2h:40m:15s. The training time is longer and the performance is not better, perhaps even slightly worse, so we go back to the network of eight hidden layers. The loss function looks the same as the previous one, so the figure is omitted, as will the following loss functions.

Next we double the amount of examples to 5,120 and halve the number of epochs to 250, see figure 4.2.7. The training time was 2h:28m:58s. Both the fit and training time is about the same as with 2,560 examples and 500 epochs except for the boundary which is close to $t = 10$. Looking back at the sines in section 4.1 we see that training time has an upper limit where increasing it is no longer beneficial. Coupled with the training loss of Black–Scholes solutions quickly getting small, it seems unnecessary to increase epochs to more than 250-500 when going forward to solve with better accuracy and in higher dimensions. We choose to stick with 250 epochs while instead increasing the amount of training examples. This is because of long training times and small updates after a while, due to both Monte Carlo simulation in the loss function and the network architecture,

Now we increase to 20,480 examples and figure 4.2.8 shows the result after 9h:30m:23s.

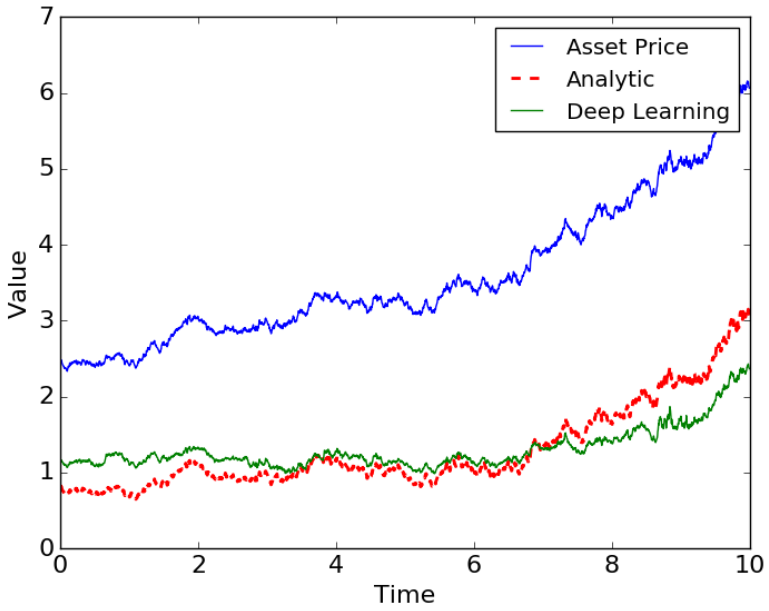


Figure 4.2.6: Black–Scholes with three assets, 2,560 examples, 500 epochs, two hidden layers with 100 nodes each. Training time: 2h:40m:15s.

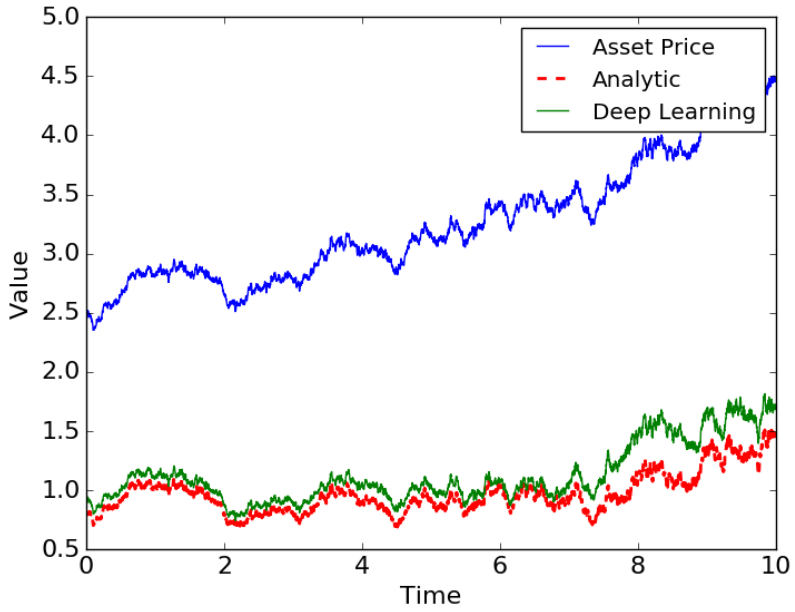


Figure 4.2.7: Black–Scholes with three assets, 5,120 examples and 250 epochs. Training time: 2h:28m:58s.

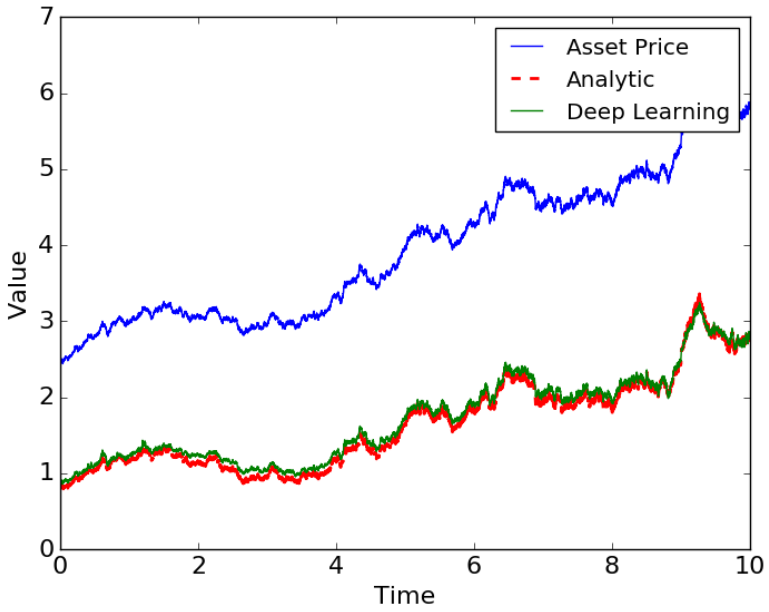


Figure 4.2.8: Black–Scholes with three assets, 20,480 examples and 250 epochs. Training time: 9h:30m:23s

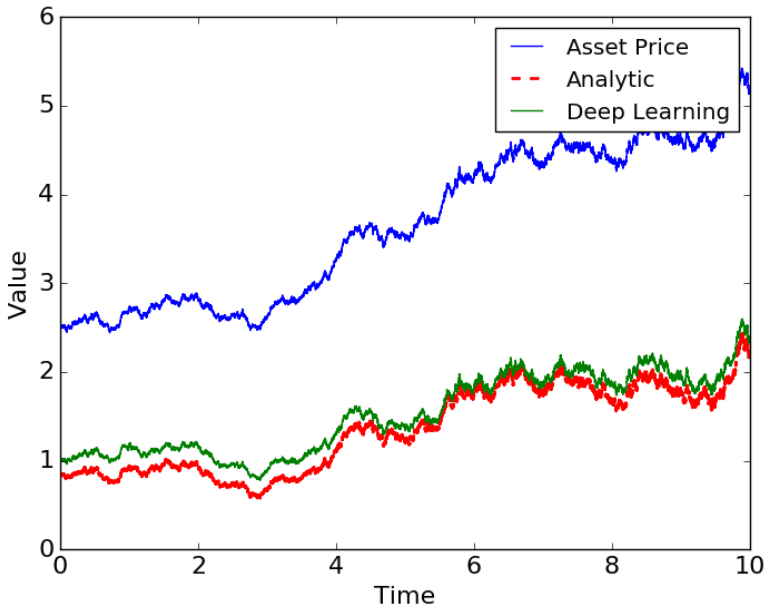


Figure 4.2.9: Black–Scholes with four assets, 61,440 examples and 250 epochs. Training time: 42h:43m:41s

4.2.3 Four Assets

This is where the simple network architecture and loss function show their practical limits. With 61,440 examples and 250 epochs figure 4.2.9 shows the result after a training time of 42h:43m:41s. We can see that the solution is good in some areas but lacking in others, specifically closer to the boundary and further away from maturity. Figure 4.2.10 shows the loss evolution. For the first time we have a clear difference in training loss and validation loss, and it seems validation loss would approach training loss given some more epochs. We could consider running 500 epochs, but with such a long training time it does not seem practical.

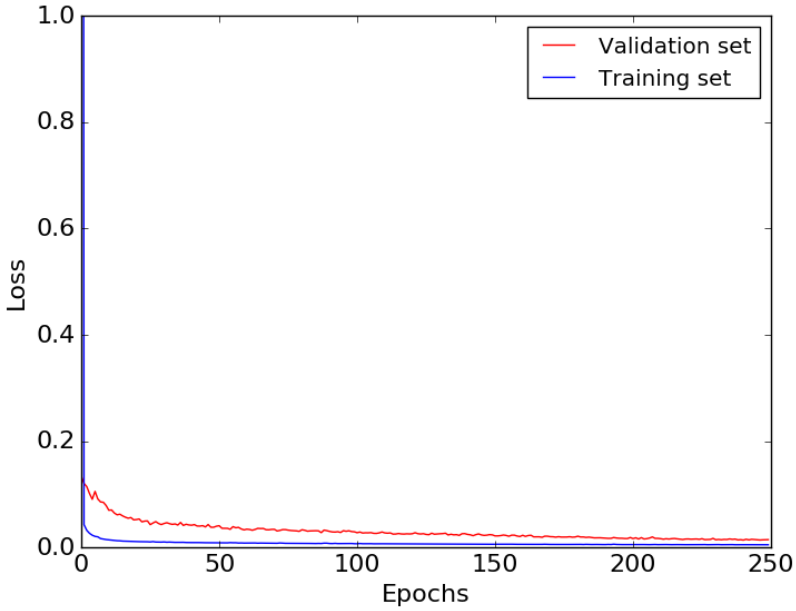


Figure 4.2.10: Black–Scholes loss with four assets, 61,440 examples and 250 epochs.

4.3 Comparing DL with FDM

How does the neural network approach compare to finite difference methods? We use an explicit Euler method backwards in time from the terminal condition with central differences in space. In [7, p. 14] the stability condition for this scheme on the Black-Scholes equation is stated to be

$$\Delta t \leq \frac{\Delta x^2}{\sigma x_i^2} \tag{4.6}$$

for all mesh points x_i. To solve with a uniform grid on the same domain as the deep learning algorithm we use

$$\Delta t \leq \frac{\Delta x^2}{\sigma S_{\max}^2} \tag{4.7}$$

with $S_{\max} = 7$. The boundary is calculated as in [6, p. 54] with

$$C(S_{\max}, t) = \max(0, S_{\max} - Ke^{-r(T-t)}) \tag{4.8}$$

Some comparisons of the two methods are listed in table 4.2. Figure 4.3.1 shows the solution space calculated with Euler's method with one asset or similarly the mean price of several equal assets. Since the FDM has to use so many time steps it becomes impossible to store a solution at all times simultaneously with more assets. The table will therefore also include runtimes of the FDM solutions at $t = 0$, effectively removing the largest dimension from the calculations. For a more complete comparison, the neural network should also have been trained at only $t = 0$, but the variance in expectations with long time to maturity makes the DL approach overly inaccurate.

In our study, we focus on whether results are feasible and how long it takes to find them. The DL times in table 4.2 are taken from plots looking as accurate as the satisfactory ones above, and with the shortest times achieved from several hyperparameter configurations. The deep learning training time of four assets is included even though the result has much to be desired. Given experience with all results so far, a satisfactory fit should be able to be acquired by increasing examples and epochs. A solution is impractical, but certainly possible, in contrast to the finite difference scheme with not enough memory available.

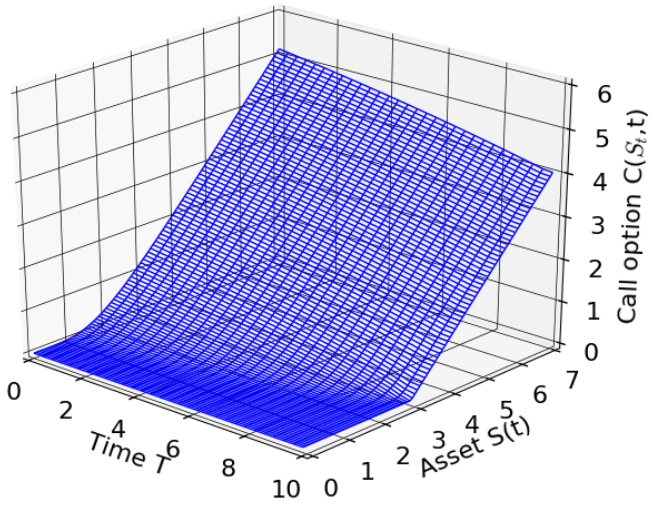


Figure 4.3.1: Solution space of the Black–Scholes equation.

Method	1 Asset	2 Assets	3 Assets	4 Assets
FDM $10n_x$	$\sim 0.00s$	$0.01s$	$0.02s$	*
	$(\sim 0.00s)$	$(0.01s)$	$(0.02s)$	$(0.07s)$
FDM $50n_x$	$0.08s$	$0.60s$	$18.75s$	*
	$(0.07s)$	$(0.52s)$	$(15.50s)$	$(39m:32s)$
FDM $100n_x$	$0.32s$	$5.14s$	*	*
	$0.30s$	$4.53s$	$23m:01s$	Estimated $48h$
DL 250 epochs $640M$	$8m:44s$			
DL 250 epochs $5,120M$		$2h:12m:50s$		
DL 250 epochs $20,480M$			$9h:30m:23s$	
DL 250 epochs $61,440M$				$> 42h:43m:41s$

Table 4.2: Runtimes of DL and FDM with varying amounts of assets, grid points each direction n_x, training examples M and epochs. The times in parenthesis are solutions only at $t = 0$ and * means computer memory error.

Chapter 5

Discussion and Conclusion

5.1 Discussion

From the training times and loss functions, it is clear that a deeper neural network is better than a shallower one for solving PDEs. It is also clear that higher precision, beyond that of visuals on a plot, requires a more complex network architecture. After the loss rapidly decreases to a certain level, simply letting the training go on for a very long time does not make the result any better. A natural next step could be to try networks with long short-term memory that lets the gradient flow through the layers. The vanishing gradient problem would then be much less of an issue.

How the deep learning method performs in solving PDEs heavily rely on how the loss function is set up. The finite difference loss has no variance compared to the simulated one, although it is averaged away over multiple epochs. Since the finite difference based loss function in the sine calculations needed more epochs than the Monte-Carlo based loss function used in the Black–Scholes problems, it is possible a second order equation with the former approach would take an even greater number of epochs. On the other hand the Monte-Carlo based loss function contributes to more time per epoch due to the number of simulations. A cause of limited accuracy could be the number of simulations. In the same Monte-Carlo simulation as shown here, but used on it's own without deep learning, one would use a many times higher amount, say 5,000, whereas here 100 had to be the limit to keep training time reasonable. Although many epochs averages out the variance, it is unclear how many is needed.

When we started this work, it was believed that the most practical result would be attained if a network architecture is found that can handle a finite difference loss function or other direct methods that find derivatives at only the randomly sampled points. In higher dimensions, if the covariances are different from zero

an not all equal, however, the cost of calculating every mixed derivative would be very high. As noted the simulations here are done stepwise all the way until the terminal condition, leading to high enough variance that many realisations are needed, but also uses a lot of time to run. As shown in [11, p. 5] there are Monte-Carlo methods that go only one step in each dimension and take the expectation from these, speeding up training time tremendously. In other words, one step in each of 100 dimensions can be done more quickly than the above 100 steps in each of just a few dimensions. This means that the deep learning approach would heavily outmatch the FDM already from a few dimensions. In essence the above compares a basic FDM with a very suboptimal use of DL. The mentioned method is likely just as sensitive to parameter change as the finite differences in the loss function, so a better network like one with long short-term memory is needed

Which strategy we use for sampling is also crucial in how the deep learning performs. In the calculations with Black–Scholes we notice that more and more examples had to be used in higher dimensions, mostly due to errors close to the boundary. A reason could be the way option value is calculated from all asset values at maturity. The more assets we have the lower the variance of their mean, effectively making the uniform sample draws lead to a non-uniform distribution of points in the solution space. An alternative strategy that might reduce training examples needed in higher dimensions could be for each sample to draw from a uniform distribution, letting this number be an asset mean, then splitting it into random portions between the spatial coordinates.

From an implementation point of view, the deep learning method seems very flexible in how easy it is to change dimensions, domain or boundary conditions. As an example increasing the dimensions by one just means adding a column vector to the input with matching changes in boundary conditions, possible with only a single increase in a dimension variable. Compare this to an FDM scheme that either requires adding more lines of code in the main loop in addition to adding indices for all arrays in case of an explicit scheme, or changing how the system of equations is solved all together in the case of an implicit scheme.

The majority of work done will instead be put into setting up the network and experimenting with different hyperparameter values. There does not seem to be any specific values that should be chosen, nor how they should change depending on problem or relative to each other. From above it seems like boundary samples should change with the overall sample size, but number of dimensions also seems to have an effect. Other hyperparameters such as regularisation strenght might have more widely used standard values, but perhaps should not even be used for PDEs in the first place. In section 3.2 we explain that this kind of regularisation seems to slow down learning instead of increasing performance.

Being mesh free is definitely an advantage that lets us increase dimensions as

well as having the solution everywhere on the domain. In two or three dimensions an FDM can be used depending on problem, but a solution at the desired points need to be interpolated. Flexibility and ease of modification are other advantages. A disadvantage can be training time, however keep in mind the above is excessively slow due to the MC simulations going all the way to the boundary. On the other hand, even if training is very slow, the weights and biases can be stored as to have the solution readily available when needed. In spirit of this work, think of the Black–Scholes equation with many hundred assets that take a very long time to train even with a good method. It only needs to be trained once, then the whole solution space is stored and we effectively have an option price "calculator" by running a single forward pass with given input. In contrast of an FDM solution, weights and bias matrices take negligible space to store.

5.2 Conclusion

To conclude, we were able to solve the Black–Scholes equation in higher dimensions with a basic deep learning approach than with an explicit finite difference method. If we want to solve a quick, one time problem the FDM can still be preferred in up to three dimensions due to easy implementation. The DL approach is just as easy, if not even easier, to modify when already set up and can be preferred on tasks that will be repeated with variations. In higher dimensions a pure MC approach can be preferred if the solution is desired at a select few points, whereas the DL approach will be better to find the whole solution space and if the same equation will be solved many times.

As the goal was to solve PDEs using deep learning, this has been a success. It was chosen to do so without the use of pre-existing libraries to better understand the process from loss evaluation of input to backpropagation and parameter updates. This deep learning implementation also turned out to be successful. The benefit is to have more knowledge about where difficulties can be, for example the vanishing gradient that occurs during backpropagation. If we go on to use deep learning packages next, it will be easier to decide which network architectures to experiment with.

References

[1] V.I. Avrutskiy, *Backpropagation Generalized for Output Derivatives*, arXiv:1712.04185, 2017.

[2] Ian Goodfellow, *Deep Learning*, MIT Press, 2016.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, *Delving Deep into Rectifiers: Surpassing Human- Level Performance on ImageNet Classification*, arXiv:1502.01852, 2015.

[4] Andrej Karpathy, *Convolutional Neural Networks for Visual Recognition*, Stanford University, lecture notes CS231n, 2019.

[5] Diederik P. Kingma, Jimmy Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980, 2017.

[6] Sima Mashayekhi, *Numerical Methods for Nonlinear PDEs in Finance*, University of Copenhagen, PhD Thesis, 2015.

[7] Gunter H. Meyer, *Numerical Methods in Finance*, Georgia Institute of Technology, lecture notes math6635, 2001.

[8] Michael A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015.

[9] Marcus Pereira, Ziyi Wang, Ioannis Exarchos, Evangelos A. Theodorou, *Neural Network Architectures for Stochastic Control using the Nonlinear Feynman–Kac Lemma*, arXiv:1902.03986, 2019.

[10] Adrian Rosebrock, *Stochastic Gradient Descent (SGD) with Python*, Blog post, 2016.

[11] Justin Sirignano, Konstantinos Spiliopoulos, *DGM: A deep learning algorithm for solving partial differential equations*, arXiv:1708.07469, 2018.

Appendix A

Python Implementation

Below is the coded neural network used for the Black-Scholes equation, here with 2 assets, 320 examples and 100 epochs. Runtime is approximately 3m:50s, resulting in a very unaccurate solution.

```
from numpy import *
from matplotlib.pyplot import*
from scipy.stats import norm
import time
set_printoptions(suppress=True)

#Clock
started = time.time()

#Constants
M = 320                 #Num samples
D = 3                   #num dimensions incl. time
L = 9                   #layers (depth)
N = 20                  #nodes per layer (width)
learning_rate = 5e-4
epochs = 100
m = 64                  #mini-batch size
reg = 0*1e-5            #L^2 regularisation
beta = 0.9              #constant of momentum
epsilon = 0.1           #step size of finite difference
r = 0.05                #interest rate
K = 3.                  #Strike price
sigma = 0.15            #Volatility
```

```
#Initializations
X = 7.*random.random_sample((M,D))   #data (input)

#boundary
X[:,0] = 10*random.random_sample(M)
X[0:10,0] = 0.
X[10:20,0] = 10.
X[20:30,1] = 0.
X[30:40,1] = 7.
X[40:50,2] = 0.
X[50:60,2] = 7.

W_in = random.randn(D,N)*sqrt(2.0/D)      #weights layer 1
b_in = ones((1,N))*0.01                   #bias layer 1
W = random.randn(L-1,N,N)*sqrt(2.0/N)     #weights
b = ones((L-1,1,N))*0.01                  #biases
W_out = random.randn(N,1)*sqrt(2.0/N)     #weights output
b_out = 0.01                              #biases output
hidden_layers = zeros((L,M,N))
VdW_in = zeros((D,N))                     #momentum
VdW = zeros((L-1,N,N))
VdW_out = zeros((N,1))
Vdb_in = zeros((1,N))
Vdb = zeros((L-1,1,N))
Vdb_out = 0.

#Functions
def target(out,dout,ddout,X=X,M=M):
    """loss function"""
    t = copy(X[:,0]).reshape(M,1)
    Expectation = zeros_like(out)
    for _ in xrange(100):
        dt = (10-t)/100
        x = copy(X[:,1:D]).reshape(M,D-1)
        for n in xrange(100):
            x = x + x*r*dt + sigma*sqrt(dt)*x*random.randn(M,D-1)
        I = -r*(10-t)
        psi = mean(maximum(0,x-K),axis=1).reshape(M,1)
        Expectation += exp(I)*psi
    Expectation /= 100.
```

```
    target = (out - Expectation)**2
    dtarget = 2*(out - Expectation)

    return  target, dtarget

def ReLU(x):
    """Rectified Linear Unit"""
    return maximum(0,x)

def forwardPass(hidden_layers,X,
        W_in=W_in,W=W,W_out=W_out,
        b_in=b_in,b=b,b_out=b_out):
    """forward pass of neural network"""
    L,M,N = shape(hidden_layers)
    hl = copy(hidden_layers)

    hl[0,:,:] = dot(X,W_in)+b_in
    z = dot(hl[0,:,:],W[0,:,:])+b[0,:,:]
    hl[1,:,:] = ReLU(z)
    for j in xrange(2,L):
        z = dot(hl[j-1,:,:],W[j-1,:,:]) +\
            b[j-1,:,:]
        hl[j,:,:] = ReLU(z)

    #output
    output = dot(hl[L-1],W_out)+b_out

    return output, hl

def SGD(step_size,hidden_layers,
        W_in=W_in,W=W,W_out=W_out,
        b_in=b_in,b=b,b_out=b_out,
        VdW_in=VdW_in,VdW=VdW,VdW_out=VdW_out,
        Vdb_in=Vdb_in,Vdb=Vdb,Vdb_out=Vdb_out):
    """Stochastic Gradient Descent"""

    #Batches
    minibatches = []
    p = arange(M)
    random.shuffle(p)
```

```
X_shuffled = X[p]
hl_shuffled = hidden_layers[:,p]

for k in xrange(0,M,m):
    X_batch = X_shuffled[k:k+m]
    hl_batch = hl_shuffled[:,k:k+m]
    minibatches.append((X_batch,hl_batch))

for X_batch, hl_batch in minibatches:

    out_batch, hl_batch = forwardPass(
    hl_batch,X_batch)[0:2]
    dout_batch, ddout_batch = 0.,0.
    #dout_batch, ddout_batch = derivative(hl_batch,X_batch)

    #gradient at end layer
    dLoss_batch = target(out_batch,dout_batch,ddout_batch,
        X=X_batch,M=m)[1]
    dLoss_batch /= m

    #Backpropagation to find gradient w.r.t W and b
    dW_out = dot(hl_batch[L-1].T,dLoss_batch)
    db_out = sum(dLoss_batch,axis=0,keepdims=False)
    dhidden = dot(dLoss_batch,W_out.T)
    dhidden[hl_batch[L-1] <= 0 ] = 0

    dW = zeros((L-1,N,N))
    db = zeros((L-1,1,N))
    for j in reversed(xrange(0,L-1)):
        dW[j] = dot(hl_batch[j].T,dhidden)
        db[j] = sum(dhidden,axis=0,keepdims=False)
        dhidden = dot(dhidden,W[j].T)
        dhidden[hl_batch[j] <= 0] = 0
    dW_in = dot(X_batch.T,dhidden)
    db_in = sum(dhidden,axis=0,keepdims=True)

    #adding regularization
    dW_in += reg*W_in
    dW += reg*W
    dW_out += reg*W_out
```

```
        #momentum
        VdW_in = beta*VdW_in - step_size*dW_in
        VdW = beta*VdW - step_size*dW
        VdW_out = beta*VdW_out - step_size*dW_out
        Vdb_in = beta*Vdb_in - step_size*db_in
        Vdb = beta*Vdb - step_size*db
        Vdb_out = beta*Vdb_out - step_size*db_out

        #parameter update
        cap = 5.
        W_in += VdW_in
        W_in[W_in > cap] = cap
        W_in[W_in < -cap] = -cap
        b_in += Vdb_in
        W += VdW
        W[W > cap] = cap
        W[W < -cap] = -cap
        b += Vdb
        W_out += VdW_out
        W_out[W_out > cap] = cap
        W_out[W_out < -cap] = -cap
        b_out += Vdb_out

def derivative(hidden_layers,X=X,order='first'):
    """Finite differences of NN"""
    h = epsilon
    u = forwardPass(hidden_layers,X)[0]
    h1 = zeros_like(X)
    h1[:,0] += h
    h2 = zeros_like(X)
    h2[:,1] += h

    u1p = forwardPass(hidden_layers,X+h1)[0]
    u1m = forwardPass(hidden_layers,X-h1)[0]
    du1 = (u1p - u1m)/(2*h)

    u2p = forwardPass(hidden_layers,X+h2)[0]
    u2m = forwardPass(hidden_layers,X-h2)[0]
    du2 = (u2p - u2m)/(2*h)
```

```
    if order =='second':
        ddu1 = (u1p - 2*u + u1m)/h**2
        ddu2 = (u2p - 2*u + u2m)/h**2
    else: ddu1,ddu2 = zeros_like(u),zeros_like(u)

    #du1,du2,ddu1,ddu2 = 0,0,0,0
    return [du1,du2], [ddu1,ddu2]

def test(hidden_layers=hidden_layers):
    """Find loss of NN on random validation set"""
    test_points = 7.*random.random_sample((M,D))
    test_points[:,0] = 10*random.random_sample(M)
    u,_ = forwardPass(
        hidden_layers,test_points)[0:2]
    #du,ddu = derivative(hidden_layers,test_points,'second')
    du = ddu = 0.
    return sum(target(u,du,ddu,X=test_points)[0])/M

def learning_rate_schedule(Loss,learning_rate):
    """Learning rate schedule to reduce
    learning rate during training"""
    if Loss <= .02:
        step_size = learning_rate /2
    else: step_size = learning_rate
    return step_size

#main part
test_array = zeros(epochs)  #To plot loss
loss_array = zeros(epochs)  #To plot loss
Loss = 1.
i = 0
step_size = learning_rate_schedule(Loss,learning_rate)
while Loss >= 0.001 and i < epochs:

    out, hidden_layers = forwardPass(
        hidden_layers,X)[0:2]
    #dout,ddout = derivative(hidden_layers,X)
    #ddout = derivative(hidden_layers,X,'second')
    dout,ddout = 0,0
```

```
        #Loss
        dataLoss = sum(target(out,dout,ddout)[0])/M
        regLoss = 0.5*reg*sum(W_in*W_in) +\
                  0.5*reg*sum(W_out*W_out) +\
                  0.5*reg*sum(W*W)
        Loss = dataLoss + regLoss

        #parameter update
        SGD(step_size,hidden_layers)

        #store loss evolution
        test_array[i] = test()
        loss_array[i] = Loss

        #continously print loss
        if (i%10 == 0):
            print 'Loss at epoch %d = %f, data loss: %f'%(i,Loss,dataLoss)

        i += 1
        step_size = learning_rate_schedule(Loss,learning_rate)
print 'Loss at end = %f'%(Loss)
print 'Data Loss: ',dataLoss,' Reg Loss: ',regLoss
print 'Test loss: ',test()
elapsed = time.time() - started
print 'Time taken: %.2f seconds'%(elapsed)

#Plots
times = linspace(0,9.99,M)
dt = times[1]-times[0]
tau = (10-times).reshape(M,1)

X1 = 2.5*ones((M,D))
X1[:,0] = times
for i in xrange(M-1):
    X1[i+1,1:D] = X1[i,1:D] +\
        X1[i,1:D]*r*dt +\
        sigma*sqrt(dt)*X1[i,1:D]*random.randn(D-1)

Y1 = forwardPass(
```

```
        hidden_layers,X1)[0]
meanX = mean(X1[:,1:D],axis=1).reshape(M,1)
dp = (log(meanX/K)+(r+0.5*sigma**2)*tau)/(sigma*sqrt(tau))
dm = dp - sigma*sqrt(tau)
C1 = norm.cdf(dp)*meanX-norm.cdf(dm)*K*exp(-r*tau)

plot(times,meanX,'-',label='Asset Price')
plot(times,C1,'r--',label='Analytic',lw=2.)
plot(times,Y1,'-',label='Deep Learning',lw=1.)
tick_params(labelsize=16)
xlabel('Time',fontsize=16)
ylabel('Value',fontsize=16)
legend()
show()

plot(arange(epochs),test_array,'r-',label='Validation set',lw=1.)
plot(arange(epochs),loss_array,'-',label='Training set',lw=1.)
tick_params(labelsize=16)
xlabel('Epochs',fontsize=16)
ylabel('Loss',fontsize=16)
axis([0,epochs,0,1])
legend()
show()

#second plot
X2 = 2.5*ones((M,D))
X2[:,0] = times
for i in xrange(M-1):
    X2[i+1,1:D] = X2[i,1:D] +\
        X2[i,1:D]*r*dt +\
        sigma*sqrt(dt)*X2[i,1:D]*random.randn(D-1)

Y2 = forwardPass(
        hidden_layers,X2)[0]
meanX = mean(X2[:,1:D],axis=1).reshape(M,1)
dp = (log(meanX/K)+(r+0.5*sigma**2)*tau)/(sigma*sqrt(tau))
dm = dp - sigma*sqrt(tau)
C2 = norm.cdf(dp)*meanX-norm.cdf(dm)*K*exp(-r*tau)

plot(times,meanX,'-',label='Asset Price')
```

```
plot(times,C2,'r--',label='Analytic',lw=2.)
plot(times,Y2,'-',label='Deep_Learning',lw=1.)
tick_params(labelsize=16)
xlabel('Time',fontsize=16)
ylabel('Value',fontsize=16)
legend()
show()
```