Seminar Paper

Applying Deep Learning to Financial Derivatives

Supervisor:

Assist. Prof. Dipl.-Ing. Dr.techn. Stefan Gerhold

Author:

Tin Marin Tunjić

01635848

Vienna, July 2019

Contents

1 Introduction

Quantitative finance was always an interesting field, full of intelligent ideas and revolutionary formulas. Today, because of big data availability and higher computing power, we starting using new methods that were unimaginable in the past. One of them is artificial intelligence. My main goal will be to introduce you to one of the main AI techniques called Deep Neural Networks. I will be using them to show you their effect in evaluating financial derivatives (specifically European Call Options).

There are a lot of different definitions of artificial intelligence and it is quite hard to get a grasp on it. Representing it as a machine "intelligence" that tries to mimic human cognitive functions would be one of it. While many still question the notion of artificial intelligence in quantitative finance, it is progressing faster than ever before. I will try to provide some of the essential information on why, how and when do we use it. Machine learning is a subset of AI technique that gives computer the ability to automatically learn from data without being explicitly programmed. It is divided into 4 large groups: supervised, unsupervised, semi-supervised and reinforcement learning. In this paper we will be focused on supervised learning (regression). Deep learning, also known as hierarchical learning, is one of machine learning methods based on artificial neural networks, that got its name imitating the looks of connections in the human brain. In the second section I will describe you the main principles behind neural networks before I give you an idea of how would implementation of neural networks look like in python.

2 Neural Networks

2.1 Introduction to Neural Networks

Artificial Neural Networks are one of the most powerful machine learning techniques. They became extremely popular with advancement in computing power and large amount of data. One of the main differences between traditional machine learning and deep learning (neural network with a lot of layers) is in the feature extraction. What I mean is that neural networks make our lives easier extracting features by itself from a big amount of data, so we don't have to do it manually. What can we classify as a big amount of data? Well, I would start talking about it if we had 1 million samples or more. In **Figure 1** we can see how performance of deep learning algorithm grows proportionally to the amount of training data. It is quite clear why we would consider using neural networks when we have a lot of data. We must take into consideration that too much data can also be a bad thing leading us to the overfitting problem. It is worth mentioning that computation time must be invested upfront (training of the neural networks) and it can take up to months, but once is finished deep learning model is incredibly fast. Since our computational cost is rising proportionally to the depth of NNs and a number of data samples, we started using powerful graphic processing units, better known as GPUs.

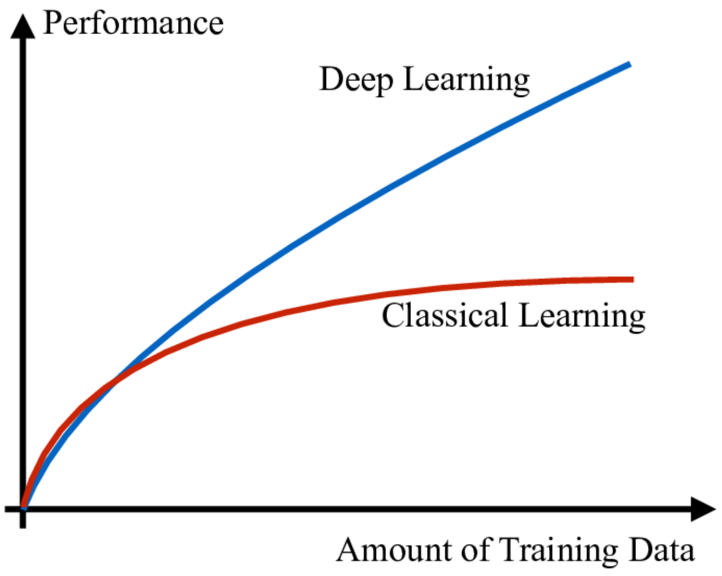

Figure 1: Performance difference between Neural Networks and classical machine learning methods

Artificial Neural Network consists of three different kinds of layers: input, hidden and output layer. Every layer consists of main building blocks called neurons which are connected with each neuron in the next layer if the model is dense. Except neurons, we have weights telling us how each node (neuron) is "significant" for the one in the next layer, biases, activation functions solving the problem of nonlinearity and an error function showing us how good our prediction with respect to the real value is. **Figure 2** shows us how one neural network for regression problem looks like.

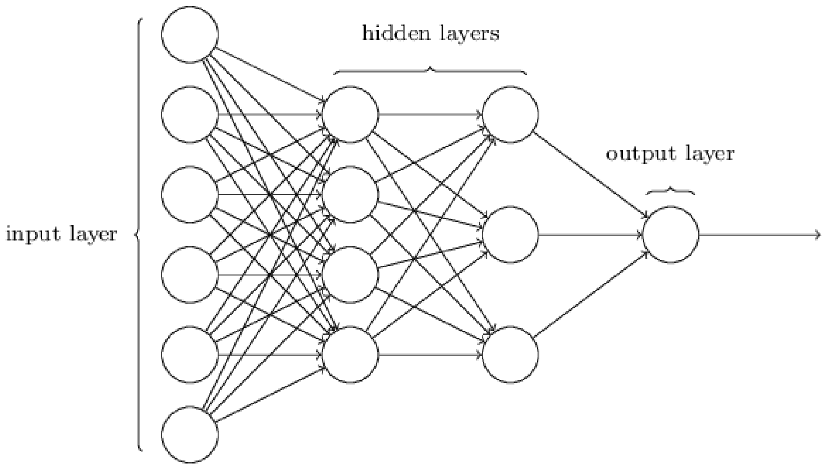

Figure 2: Neural Network with 6 inputs,1 output and 2 hidden layers

2.2 Forward Propagation

Table 1: Notations

Notation	**Definition**
$\mathbf{x}; \mathbf{x_j}$	Input, j-th element of input layer
$\mathbf{y}$	Output
$\mathbf{\hat{y}^{(i)}}$	Predicted output for the i-th training example
$\mathbf{L}$	Number of layers
$\mathbf{b_j^{[l]}}$	Bias vector for the l-th layer and j-th element
$\mathbf{W_j^{[l]}}$	Weight matrix on the l-th layer and j-th element
$\boldsymbol{\sigma(z)}$	Activation function
$\mathbf{a^{[l]}}$	Result of the l-th neuron
$\mathbf{a^{[0]}}; \mathbf{a^{[L]}}$	Input, output layer
$\mathbf{C(W^{[l]}, b^{[l]})}$	Cost function

Each neuron in the new layer $a_j^{[l]}$ takes a vector of inputs from the previous one (except when it is the input layer) $a_j^{[l-1]}$, gets multiplied by a weight vector $W_j^{[l]}$, we add a bias $b_j^{[l]}$ and get:

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

or in a matrix notation:

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

Table 2: Most popular non-linear activation functions

Function	**Definition**	**Type**
Rectified linear unit (ReLU)	max(0, z)	Regression
Leaky ReLu	max(0.01z, z)	Regression
Sigmoid	$1/(1+e^{-z})$	Classification
tanH	$(e^z - e^{-z})/(e^z + e^{-z})$	Classification

When we apply non-linear activation function $\sigma(z)$, we get:

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

Every neuron in the same layer uses the same activation function but different layers can have different activation functions. They have to be non-linear and continuously differentiable in order to be useful. We "give" input features to our neural network and then they are "forward propagated" through our layers to the end point, our output (take a look at **Figure 2**). In mathematical sense equations (1) and (2) describe this forward propagation from $a^{[0]} = x$ (input) to the $a^{[L]}$ (ouput). Different number of layers and different number of nodes (neurons) are one of the hyperparameters we have to think about when we want to make the best possible prediction. Because of them training time can vary quite a lot and it is not always true that more layers and nodes mean better prediction. Quite often as many as two or three layers can give us the best results. In this paper we will always talk about one output but neural networks are also used to solve classification problems with multiple discrete outputs. Different problems use different activation functions (**Table 2**).

2.3 Backward Propagation

The main goal behind backward propagation is to optimize the value of our weights so that we find the global minimum of our cost function. Initial weights are most of the time randomly generated and then updated to minimize the error. There are a lot of cost functions for regression problems (root mean squared error, mean absolute error...), but we are going to focus ourselves to mean squared error (MSE):

$$C(W^{[l]}, b^{[l]}) = \frac{1}{2N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2. \tag{4}$$

Why did we change MSE and add 2 in front of N. The $\frac{1}{2}$ term doesn't really matter because the optimal value for the weights would remain the same in both cases. When we get a derivative of a squared function terms $\frac{1}{2}$ and 2 cancel each other out and we get a prettier expression.

By calculating the derivative of our cost function $C(W^{[l]}, b^{[l]})$ with respect to each weight $W_j^{[l]}$ we attain:

$$\frac{\partial C(W^{[l]}, b^{[l]})}{\partial W_j^{[l]}} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial W_j^{[l]}}\Big(\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2\Big) = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial C_i}{\partial W_j^{[l]}}. \tag{5}$$

Since, for the purposes of derivation, we want to make our equations as simple as possible and that is why we will concern ourselves with only one input-output pair. Thus,with help of equation (5), the cost function in question for derivation is:

$$C = \frac{1}{2}(\hat{y} - y)^2. \tag{6}$$

In order to explain a derivative of the cost function with respect to weights, we first derive:

$$\begin{aligned}
\frac{\partial Z^{[L]}}{\partial W^{[L]}} &= \frac{\partial}{\partial W^{[l]}}\Big(W^{[L]}a^{[L-1]} + b^{[L]}\Big) && (7)\\
&= a^{[L-1]} && (8)\\
\frac{\partial a^{[L]}}{\partial Z^{[L]}} &= \frac{\partial}{\partial Z^{[L]}}\Big(\sigma(Z^{[L]})\Big) && (9)\\
&= \sigma'(Z^{[L]}) && (10)\\
\frac{\partial C}{\partial a^{[L]}} &= \frac{\partial}{\partial a^{[L]}}\frac{1}{2}\Big(\hat{y} - y\Big)^2 && (11)\\
&= \frac{\partial}{\partial a^{[L]}}\frac{1}{2}\Big(a^{[L]} - y\Big)^2 && (12)\\
&= (\hat{y} - y) && (13)
\end{aligned}$$

I have to note that $\hat{y} = \sigma(a^{[L]})$ in equation (11-13), but since we are using linear activation function in the last layer equality of the equation is justified. With help of these derivatives (equations 7-13) and using the chain rule we get:

$$\frac{\partial C}{\partial W^{[L]}} = \frac{\partial Z^{[L]}}{\partial W^{[L]}}\frac{\partial a^{[L]}}{\partial Z^{[L]}}\frac{\partial C}{\partial a^{[L]}} = a^{[L-1]}\sigma'(Z^{[L]})(\hat{y} - y). \tag{14}$$

We are going to do the same thing for our bias:

$$\frac{\partial C}{\partial b^{[L]}} = \frac{\partial Z^{[L]}}{\partial b^{[L]}}\frac{\partial a^{[L]}}{\partial Z^{[L]}}\frac{\partial C}{\partial a^{[L]}} = \sigma'(Z^{[L]})(\hat{y} - y) \tag{15}$$

with derivative of $Z^{[L]}$ with respect to $b^{[L]}$ equal to 1:

$$\frac{\partial Z^{[L]}}{\partial b^{[L]}} = \frac{\partial}{\partial b^{[L]}}\left(W^{[L]}a^{[L-1]} + b^{[L]}\right) = 1. \tag{16}$$

Just to make things as clear as possible, with the same logic, going back one more step, we attain:

$$\begin{align}
\frac{\partial C}{\partial a^{[L-1]}} &= \frac{\partial Z^{[L]}}{\partial a^{[L-1]}}\frac{\partial a^{[L]}}{\partial Z^{[L]}}\frac{\partial C}{\partial a^{[L]}} \tag{17}\\
&= W^{[L]}\sigma'(Z^{[L]})(\hat{y} - y) \tag{18}\\
\frac{\partial C}{\partial W^{[L-1]}} &= \frac{\partial Z^{[L-1]}}{\partial W^{[L-1]}}\frac{\partial a^{[L-1]}}{\partial Z^{[L-1]}}\frac{\partial C}{\partial a^{[L-1]}} \tag{19}\\
&= a^{[L-2]}\sigma'(Z^{[L-1]})W^{[L]}\sigma'(Z^{[L]})(\hat{y} - y). \tag{20}
\end{align}$$

We "backpropagate" our algorithm all the way up until we get to the input. **Figure 3** nicely shows us how one neural network looks like with all the weights and nodes (we don't have any bias here). As the formulas showed us backpropagation begins with the last layer and error made by the prediction of the neural network.

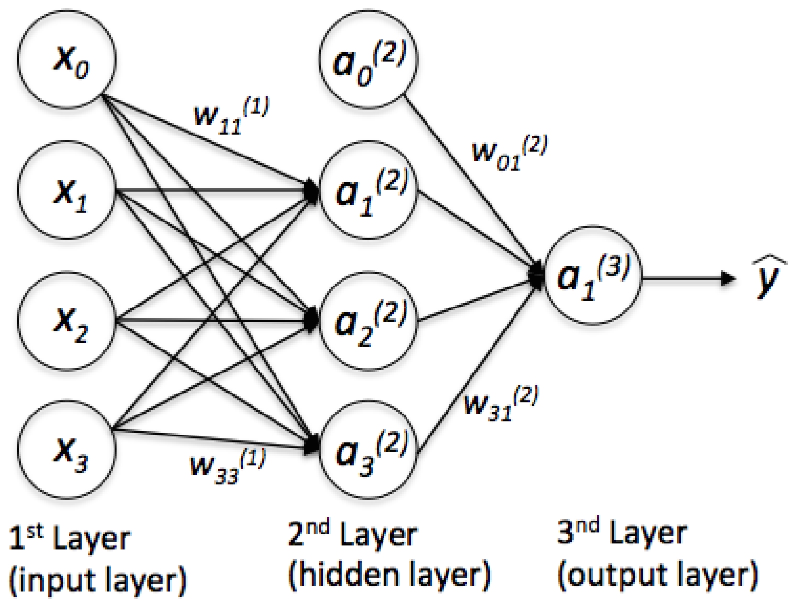

Figure 3: Neural Network with 4 inputs, 1 output and 1 hidden layer

2.4 Gradient Descent

By now, you must be asking yourselves how exactly do we optimize our weights and what do we need derivatives of the cost function with respect to weights and biases for? Well, that's where gradient descent comes to play and theory starts getting more exciting, since we have one more hyperparameter to optimize.

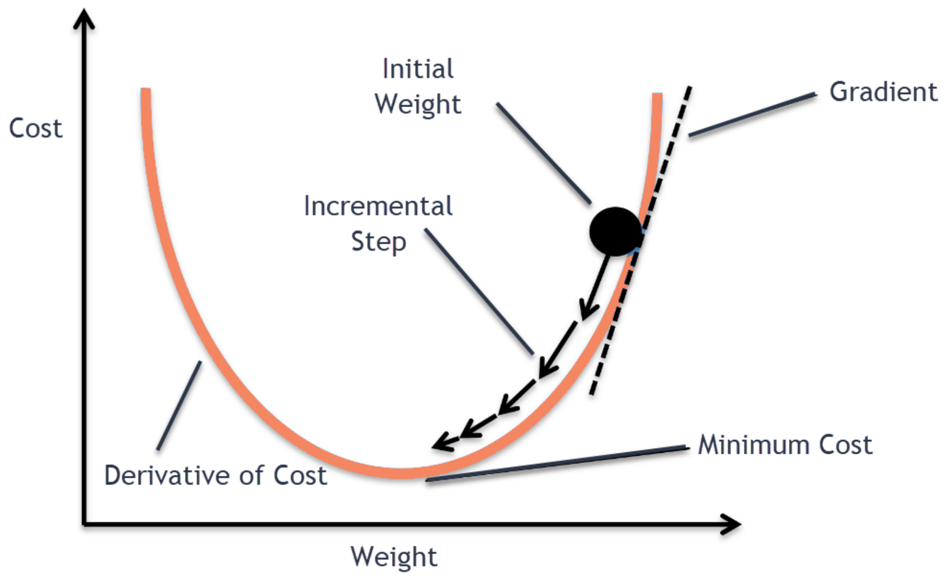

Figure 4: Gradient descent

Figure 4 shows us a graph with weight values on the x-axis and cost values on the y-axis. What does this graph actually present? Well, it tells us dependence between the cost function and weights and that is exactly what gradient is. As we said earlier, we want to minimize our cost function so we take this gradient (derivative of the cost function with respect to weights) and see if it is negative or positive. If it is negative it will be on the left side of our slope, respectively if positive on the right side. Now, we try to get as near as possible to out minimum so we take incremental step (step size), also known as learning rate, in the direction opposite to the gradient. We do this trick many times until we find ourselves at the minimum. Mathematically, we update our weights like this:

$$W^{[L]} = W^{[L]} - \alpha \frac{\partial C}{\partial W^{[L]}}, \tag{21}$$

where α represents the learning rate. We do the same thing for our biases:

$$b^{[L]} = b^{[L]} - \alpha \frac{\partial C}{\partial b^{[L]}}. \tag{22}$$

Learning rate is the hyperparameter I was talking about. There are two large problems concerning the learning rate. First, if it is too small we converge too slowly to our minimum, hence computational cost gets bigger. Second, if it is too big we will be oscillating quite a lot. We usually set it to lie between 0.05 and 0.10. **Figure 5** shows us nicely how a gradient descent looks in a 3-dimensional space with the x-axis representing weights, the y-axis biases and the z-axis cost function.

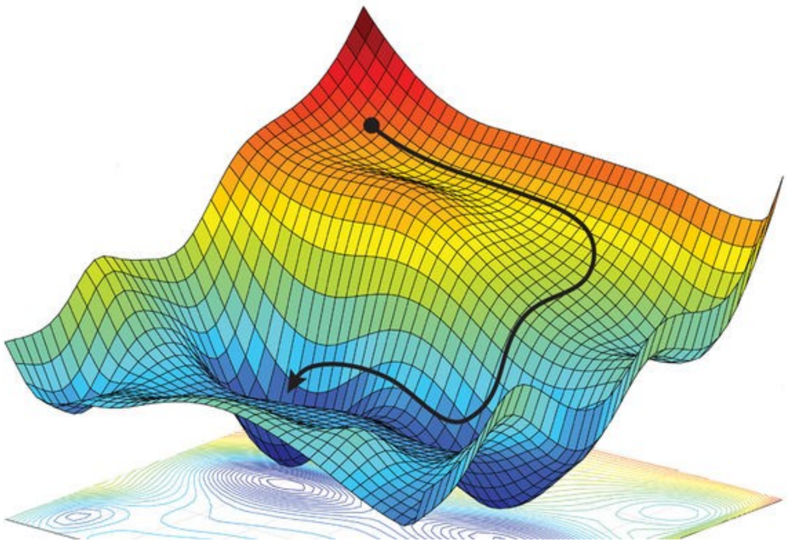

Figure 5: Gradient descent in 3- dimensional space

We say we completed one "epoch" of gradient descent each time we update parameters using gradients. Important to add is that we do not train the model to fully minimize the cost function, as that could result in overfitting, leading to poor performance on new data.

2.5 Stochastic Gradient Descent

I will begin this subsection by defining a very important term we use called ”batch”. Batch is the total number of data samples we use to calculate gradient in a single epoch. Why is that important? Well, if we use whole batch (gradient descent) in a training set with a billions of data samples and a lot of features, we are going to have a hard time training it. The computational cost will be enormous. That is where some other methods, besides gradient descent, become relevant. The most extreme one is stochastic gradient descent or SGD. It uses just one data sample that is randomly picked, hence stochastic in name, to perform each iteration. You can clearly see the difference (**Figure 6**) in performance between gradient descent, that uses whole trainings data set as a batch, and a stochastic gradient descent.

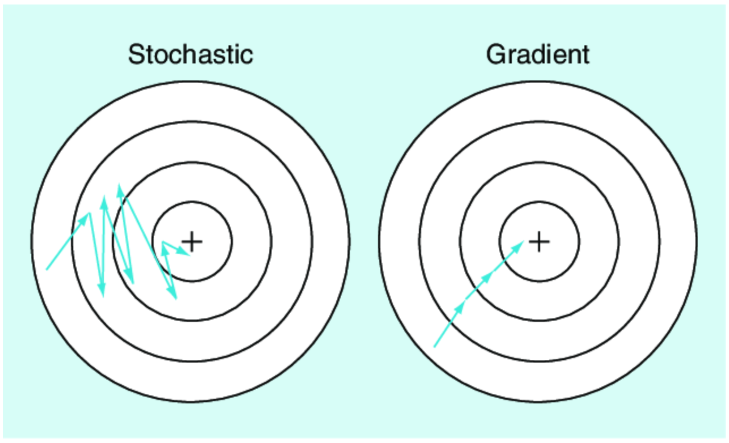

Figure 6: Difference in performance between 2 methods, where center presents minimum of the cost function

Another popular method is mini-batch stochastic gradient descent. It typically uses between 10 and 10 000 randomly chosen examples. Mini-batch SGD reduces the noise obtained by SGD and has a much lower computational cost than full batch (**Figure 7**).

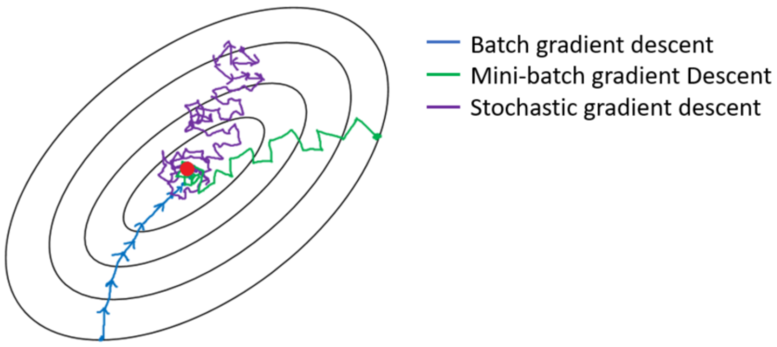

Figure 7: Differences in performance including mini-batch

2.6 Adam Optimizer

In a room of different optimization algorithms there is one that stands out and has taken deep learning community by storm. As you probably guessed it, its name is Adam, shortened from Adaptive moment estimation. Before, I was talking how learning rate is one of the hyperparameters we are trying to optimize. It was a constant value that we defined at the beginning of our learning process and kept it as a constant throughout whole training. Adam optimizer adapts each network parameter (weight, bias) separately as learning unfolds. Some of the most attractive benefits of using Adam are represented by its authors, as follows:

We introduce Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning.

First, I would like to define:

$$\begin{aligned} t &= timestep(iteration), && (23)\\ \mathbf{E}[m_t] &= \mathbf{E}[\nabla C_t], && (24)\\ \mathbf{E}[v_t] &= \mathbf{E}[(\nabla C_t)^2], && (25) \end{aligned}$$

where ∇C represents a gradient of the cost function. Logic behind the method is that it stores exponentially decaying average of previous gradients m_t and an exponentially decaying average of previous gradients squared v_t and calculates the parameter updates in the following way:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla C, && (26)\\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla C)^2, && (27)\\ \hat{m}_t &= \frac{m_t}{1-\beta_1}, && (28)\\ \hat{v}_t &= \frac{v_t}{1-\beta_2}, && (29) \end{aligned}$$

With help of equations (26-29), we obtain final weight and bias update rule:

$$W_t = W_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \tag{30}$$

$$b_t = b_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \tag{31}$$

where β_1 and β_2 are the exponential decay factors with default values equal to 0.9 and 0.99, respectively. ϵ is a small number created to avoid zero division with 10^{-8} as a default value.

Pseudo-code of Adam

```
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
**while** $\theta_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
```

3 Option pricing

3.1 Black-Scholes Formula

People in finance field are constantly trying to figure out new methods of option evaluations and deep learning is just one of them. In this section I will be talking only about option pricing using the Black-Scholes model, since we are going to concentrate on European call options in this paper, but worth mentioning is that you can apply NNs on any derivative. European call option is a financial contract between two parties, the buyer and the seller. The buyer has the right, but not the obligation to buy an agreed quantity of the underlying commodity from the seller at the expiration date (T) for a certain price (Strike price). Why do we want to train the deep learning model on Black-Scholes formula? The answer to that question could be:"Because we want to see if the neural network can start "mimicking" Black-Scholes model without explicitly seeing the formula. Before we start with preprocessing and implementation, I should first present main equations behind the Black-scholes formula. I won't go in depth, since this paper is mostly concerned with logic behind the neural networks and its implementation. European call option using the Black-Scholes formula is represented, as follows:

$$V(S,t) = S\Phi(d_1) - Ke^{-r(T-t)}\Phi(d_2), \tag{32}$$

$$d_1 = \frac{ln(S/K) + (r + \sigma^2/2)(T-t)}{\sigma\sqrt{T-t}}, \tag{33}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{34}$$

where S is the current stock price, K is the option strike price, T is option maturity,Φ is the distribution function of the standard normal distribution, r is the risk-free rate, σ is the annualized volatility. When calculating maturity we should be careful with computation, because there is approximately 252 Trading days in a year and not 365.

3.2 Greeks

Greeks are quantities that are very popular in quantitative finance because of the information they hold about the stock. They have also been called risk sensitivities, risk measures or hedge parameters. They are represented in the following way:

$$Delta - \Delta = \frac{\partial V}{\partial S} \tag{35}$$

measures the sensitivity of an options theoretical value (obtained in Black-Scholes formula) to a change in the price of the underlying asset,

$$Gamma - \Gamma = \frac{\partial^2 V}{\partial S^2} \tag{36}$$

measures the rate of change in the delta for each one-point increase in the underlying asset,

$$Rho- = \frac{\partial V}{\partial r} \tag{37}$$

the rate at which the price of a derivative changes relative to a change in the risk-free rate of interest,

$$Theta - \Theta = \frac{\partial V}{\partial t} \tag{38}$$

a measure of the time decay of an option, the dollar amount that an option will lose each day due to the passage of time,

$$Vega- = \frac{\partial V}{\partial \sigma} \tag{39}$$

measures the sensitivity of the price of an option to changes in volatility.
I use greeks here as input features, but NNs can also be used to compute the greeks. Correlation plot made in python using the seaborn library on Apple data:

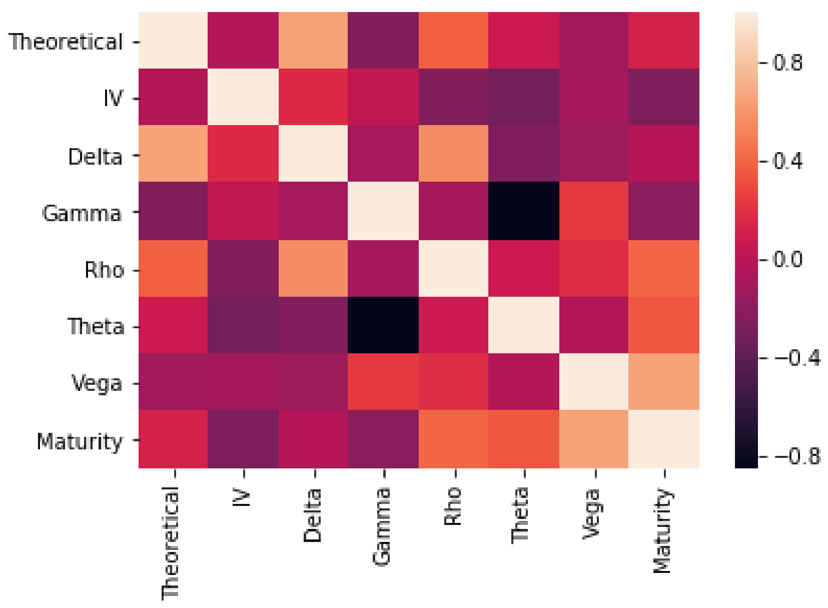

Figure 8: Correlation plot

4 Implementation

4.1 Hyperparameter Tuning

Let's take a step back and try to remember all the hyperparameters we tried to optimize. Those were learning rate, number of samples in a batch and we mentioned the number of layers and a number of nodes in those layers. I won't explicitly talk about how many layers and how many nodes there should be, because every problem will have different optimal values, but you should keep in mind that these hyperparameters, if adjusted properly, can make a huge difference. Code written in python is not optimal code used to get the best prediction of the European call option. It is merely presented here so you could use it in your own implementation and see how easy it is to implement it. Main library used is called Keras, very popular library for neural networks with a lot of amazing tools. Other libraries used are pandas, numpy, seaborn, scipy, sklearn...

4.2 Preprocessing

I downloaded data from the site http://www.barchart.com and started extracting features that I am going to use. Worth mentioning is that European call option can be priced using only four inputs (E.g time to maturity, implied colatility, risk-free rate and spot/strike). We should always normalize our data before training. What I mean by that in this case is we should devide our current stock price with a strike price, hence spot/strike. That is the reason why our output will be V/K (theoretical price of European call option devided by the strike price). When we finished extracting and normalizing data it would be good if we shuffled the data so that the model can learn from a wide variety of examples without any order.

4.3 Python Code

4.3.1 Splitting the Data Set

```
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

predictors=df_all[['Spot/Strike','IV','Maturity','Delta','Gamma',
'Rho','Theta','Vega']]
target=df_all[['Theoretical/Strike']]
X_train,X_test,y_train,y_test=sklearn.model_selection.
train_test_split(predictors,target,test_size=0.3,
random_state=42)
```

Splitting the data set on training and test data in proportion 70:30. Input data are spot/strike, implied volatility,time to maturity and greeks. Output data is theoretical value divided by the strike price.

4.3.2 Model with Adam optimizer

```
n_cols=predictors.shape[1]

model = Sequential()

model.add(Dense(256,activation='relu',input_shape=(n_cols,)))
model.add(Dense(512,activation='relu'))
model.add(Dense(256,activation='relu'))
model.add(Dense(1))

model.compile(optimizer='adam',loss='mean_squared_error')

#early_stopping_monitor=EarlyStopping(patience=3)
#validation_split=0.2, callbacks=[early_stopping_monitor],
verbose=True
model_1_training=model.fit(X_train,y_train,epochs=100,
verbose=False)

prediction = model.predict(X_test)
MSE = mean_squared_error(y_test, prediction)

print('MEAN_SQUARED_ERROR_=',MSE)
```

Sequential model is model that connects nodes from the previous layer only with nodes in the next layer. Dense, as mentioned before means each node is connected with the node in the next layer. Numbers 256,512,256 are numbers of nodes in hidden layers. ReLu activation function. EarlyStopping function can be used if we want that our training stops when model doesn't improve in "patience many" epochs.

4.3.3 Model with SGD

```
from keras.optimizers import SGD

def get_new_model(input_shape=(n_cols,)):

    model = Sequential()

    model.add(Dense(128,activation='relu',input_shape=(n_cols,)))
    model.add(Dense(256,activation='relu'))
    model.add(Dense(128,activation='relu'))
    model.add(Dense(1))
    return(model)

lr_to_test = [0.01,0.1,1]

for lr in lr_to_test:

    model2=get_new_model()
    my_optimizer=SGD(lr=lr)

    model2.compile(optimizer=my_optimizer, loss='mean_squared_error')
    model_2_training=model2.fit(X_train, y_train, epochs=100,
    verbose=False)

    prediction2 = model2.predict(X_test)

    MSE = mean_squared_error(y_test, prediction2)
    print('MEAN_SQUARED_ERROR_=',MSE)

    plt.plot(model_1_training.history['loss'], 'r',
    model_2_training.history['loss'], 'b')

    plt.xlabel('Epochs')
    plt.ylabel('loss_score')
    plt.show()
```

Model trained with 3 different learning rates for SGD: 0.01, 0.1, 1. I wanted to plot the the models to see the difference between 2 models, hence plot.

4.4 Possible Problems

4.4.1 Dying Neuron

First possible problem I will talk about is the dying neuron problem. Even if the learning rate is fine tuned we can still run into this problem. Why is it called ”The dying neuron problem”? Problem occurs when a neuron takes a value less than zero for all rows of data in the training set. Since we use ReLu activation function every output coming from this node will be equal to zero and since every output is equal to zero every gradient will be equal to zero, which means that this neuron brings nothing to our neural network and hence the name ”dying neuron”.

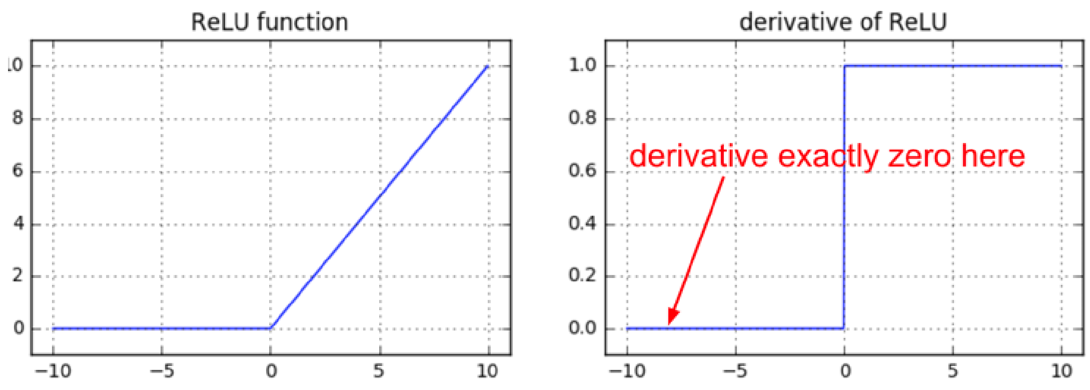

Figure 9: ReLu activation function

4.4.2 Vanishing Gradient

In the second section I mentioned 4 activation function. One of them was tanH and I am going to use this activation function as an example of the problem of vanishing gradient. Let’s take a look at a **Figure 10** to see how tanH function looks like.

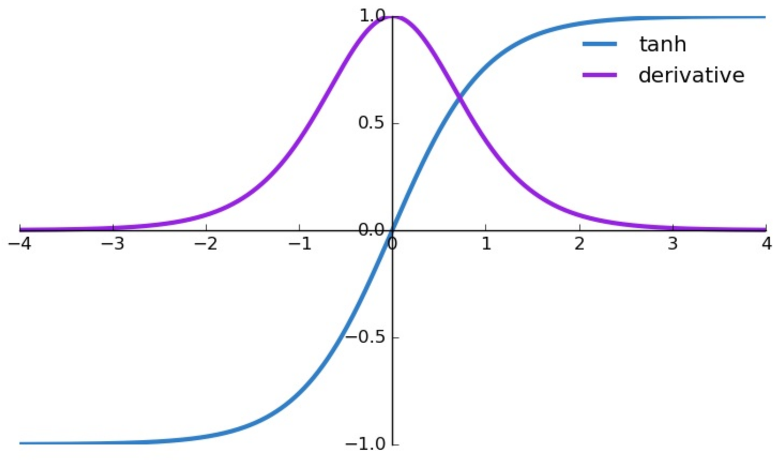

Figure 10: tanH activation function with its derivative

As you can see from the name of the problem gradient starts vanishing, but why? If we take another look at the **Figure 10**, we will see that derivatives are getting smaller if we move further away from the origin. The thing is as we are moving further away from the origin updates become smaller and smaller, resulting in no learning.

References

[1] Ryan Ferguson and Andrew Green: *Deeply Learning Derivatives*, 14/10/2018

[2] Robert Culkin and Sanjin R. Das: *Machine learning in Finance: The case of Deep Learning for Option Pricing*, Santa Clara University August2, 2017

[3] Axel Bromström and Richard Kristiansson: *Exotic Derivatives and Deep Learning*, Stockholm, Sweden 2018.

[4] Sang Il Lee and Seong Joon Yoo: *Multimodal Deep Learning for Finance: Integrating and Forecasting International Stock Markets*,Department of Computer Engineering, Sejong University, Seoul, Republic of Korea

[5] Luyang Chen, Markus Pelgery and Jason Zhuz: *Deep Learning in Asset Pricing*,April 2, 2019

[6] Rosdyana Mangir Irawan Kusuma1, Trang-Thi Ho, Wei-Chun Kao, Yu-Yen Ou and Kai-Lung Hua: *Using Deep Learning Neural Networks and Candlestick Chart Representation to Predict Stock Market*, Department of Computer Science and Engineering, Yuan Ze University, Taiwan Roc Department of Computer Science and Engineering, National Taiwan University of Science and Technology, Taiwan Roc Omniscient Cloud Technology

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

[8] Backpropagation: **https://brilliant.org/wiki/backpropagation/**

[9] Deep Learning in Python, Advanced Deep Learning with Keras in Python: **https://www.datacamp.com/home**

[10] Deep Neural Networks for regression problems: **https://towardsdatascience.com**