UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE MATEMÁTICA

INSTITUTO SUPERIOR DE CIÊNCIAS
DO TRABALHO E DA EMPRESA
DEPARTAMENTO DE FINANÇAS

Estimate European vanilla option prices using artificial neural networks

Diogo Pinto Flórido

Mestrado em Matemática Financeira

Dissertação orientada por:
Diana Mendes

2022

Acknowledgements

I would like to thank two people that played a significant role in this undertaking. My thesis advisor Diana Mendes for her guidance and counseling in what was a personally ambitious project. My girlfriend Mira Pitkänen for her continued support and positivity even in the most despairing moments.

Thank you.

Resumo alargado

A literatura de avaliação de opções é vasta e complexa. De grosso modo pode distinguir-se duas abordagens para a avaliação de opções: analíticas, como o modelo Black-Scholes-Merton (BSM) ou o modelo Heston, e numéricas, como simulações Monte-Carlo ou, mais recentemente, aprendizagem automática (machine learning). Esta tese é sobre a avaliação de de opções, de compra e de venda, Europeias baunilha sobre o índice S&P 500 usando aprendizagem automática. Mais especificamente a tese usa redes neuronais artificiais (artificial neural networks) (ANNs) de diferentes arquiteturas para estimar preços de opções. O uso de redes neuronais artificiais é preferido em relação a outras técnicas de aprendizagem automática pois estas são boas a lidar com grandes bases de dados com relações não lineares entre as variáveis.

Resumindo brevemente a revisão bibliográfica:

- O uso de aprendizagem automática para efeito de avaliação de opções teve início em 1993 com o trabalho de Malliaris and Salchenberger [27];
- Um problema comum neste tipo e abordagem é o facto de ser difícil escolher os chamados hyper-parâmetros. Estes parâmetros não devem ser confundidos com os inputs das redes;
- Exemplos de hyper-parâmetros são: o número de camadas escondidas (hidden layers), o número de neurónios (neurons) nessas camadas, o "batch size", o número de épocas/ciclos, a taxa de aprendizagem (learning rate), etc;
- Verifica-se que muitos autores usam inputs semelhantes aos do modelo BSM para as suas redes;
- O artigo Andrey Itkin (2019) [22] parece ter sido um dos primeiros a tentar criar uma rede que gerasse preços livres de oportunidades de arbitragem;
- O artigo [33] apresenta uma extensa revisão bibliográfica onde explicam que o uso de redes neuronais artificiais não se limita a estimar preços mas também, por exemplo, volatilidades implícitas (implied volatilities) e são úteis para questões de cobertura do risco (hedging);
- No que diz respeito a literatura portuguesa, não parece existir nada relacionado com estimar preços de opções Europeias com recurso a redes neuronais artificiais.

Nesta tese tenta-se replicar o trabalho feito em Ke and Yang (2019) [25]. Isto significa construir 3 tipos de redes neuronais artificiais: duas perceptron multicamada (multi-layer perceptron) e uma rede de memória longa de curto prazo (long short-term memory); usar essas redes para prever preços de opções Europeias baunilha; calcular métricas baseadas na disparidade entre preços previstos e de mercado, com o objetivo de verificar quão próximos estão os primeiros dos segundos; e finalmente comparar essas métricas com o modelo BSM. Deve ficar bem claro que as redes mencionadas não têm qualquer condição que implique preços livres de oportunidades de arbitragem. Esta situação contrasta com a de modelos analíticos como o BSM em que existem pressupostos subjacentes que forçam os preços estimados a estarem livres dessas oportunidades. Por isso, mesmo que as redes neuronais artificiais apresentem melhores métricas, é preciso mais trabalho na área. Preços que permitem arbitragem não são úteis.

Nesta tese tomaram-se escolhas um pouco diferentes das observadas na literatura:

- Usa-se uma base de dados de aproximadamente 13,5 milhões de observações de opções o que contrasta com as milhares da maior parte dos trabalhos na área;
- Não se usa a chamada "homogeneity hint";
- Só se excluem da análise opções que já expiraram e que portanto não precisam de ser avaliadas.

Os dados usados nesta tese são os seguintes:

- Preços diários do ativo subjacente, o S&P 500;
- Taxa nominal diária de obrigações americanas de 3 meses, para servir de "risk-free rate";
- Preços diários de opções SPX e SPXW.

Neste artigo usaram-se 4 modelos (que na verdade são 8 pois tem que haver um modelo para opções de compra e outro para opções de venda). Os 4 modelos são:

- Modelo BSM para servir de referência nas comparações nos modelos de redes neuronais artificiais;
- Modelo MLP1 que toma como inputs os mesmos do BSM e produz um preço de equilíbrio para uma opção. Este preço, ao contrário de um produzido pelo BSM não é necessariamente livre de oportunidades de arbitragem visto que nenhuma restrição é imposta para esse efeito;

- Modelo MLP2 que difere do MLP1 pois produz dois preços para cada opção, o preço de compra (bid) e o preço de venda (ask). Para efeitos de avaliação do modelo usei a média entre esses 2 preços;
- Modelo LSTM que usa os últimos 20 preços do ativo subjacente para calcular uma volatilidade. O objetivo é testar se a volatilidade no momento t depende da volatilidade nos tempos $t-1$, $t-2$, $\ldots$, e $t-20$. Essa volatilidade é depois usada como input num modelo MLP1, juntamente com os outros inputs, para produzir um preço de equilíbrio de uma opção.

De seguida vou apresentar uma breve explicação teórica de cada tipo de rede. Mas primeiro, de um modo geral é importante referir que as redes neuronais artificiais são constituídas por camadas (input, "hidden" e output) que por sua vez são formadas por neurónios. Um modelo MLP é uma ANN que segue 3 passos: inicialização (Init), "feed-forward" (FF) e "back-propagation" (BP). Init é a geração de valores iniciais para os pesos e "biases" da rede. FF significa que a ANN vai dos inputs aos outputs, passando de camada a camada num só sentido. O processo de BP calcula a função de custo (loss function) e vai atualizando os valores dos pesos e "biases" até se chegar a um valor mínimo da função de custo. Um modelo LSTM é mais complexo pois é um exemplo de uma rede neuronal recorrente (RNN). Este tipo de redes serve para detetar padrões ao longo do tempo. Ou seja, pode ser usado para prever a próxima palavra numa frase ou, no caso desta tese, para prever o "próximo" preço de uma opção dada informação passada e não só presente. Curiosamente os modelos de avaliação de opções mais conhecidos assumem que o preço de uma opção não depende de variáveis do passado. Contudo, há literatura que mostra a existência de dependência da volatilidade presente em relação à volatilidade passada. Os modelos LSTM foram pensados para resolver um problema chamado "vanishing gradient problem" e têm mostrado, ao longo do tempo, que o conseguem efetivamente fazer. Também são eficazes a resolver o problema denominado por "exploding gradients problem".

Para avaliar os resultados empíricos usei várias métricas como por exemplo a média do quadrado dos erros (mean-square error) ou a mediana do erro percentual (bias). O modelo MLP1 é o único que supera o BSM em todas as métricas. Contudo é importante lembrar que, os modelos de aprendizagem automática desta tese não impõe qualquer tipo de restrição ao nível de oportunidades de arbitragem e por isso não se pode automaticamente concluir que o modelo MLP1 é superior ao BSM.

Comparando os meus resultados com os de [25], não os consegui replicar na sua grande maioria. Por exemplo os MSEs do MLP1 são substancialmente superiores. No que diz respeito ao modelo MLP2, este é inferior ao BSM em quase todas as métricas, contrariando o

que foi reportado em [25]. Em relação ao LSTM os números deles são ligeiramente superiores aos meus. Todas estas diferenças podem ter que ver com as bases de dados usadas, mas suspeito que a diferença está no código. Apesar dos autores terem publicado o seu código online em formato de livre acesso, foram precisos muitos ajustes do meu lado para conseguir gerar resultados.

Ao nível de convergência e ajuste (fit) pode dizer-se que: O MLP1 não apresenta "overfitting", o MLP2 não apresenta "overfitting" mas no que diz respeito às opções de venda os erros de treino e erros de validação não convergiram, o LSTM apresenta overfitting e não converge. Para este talvez a resposta seria usar "early stopping".

Keywords: Avaliação de Opções, Aprendizagem Automática, Redes Neuronais Artificiais, Perceptron Multicamada, Memória Longa de Curto Prazo

Abstract

In this thesis I attempt to replicate three machine learning (ML) option pricing models thought of in Ke and Yang (2019) [25] and compare them to the Black-Scholes-Merton (BSM) model. Two of them are multi-layer perceptron (MLP) with the same inputs as the BSM. The third is a long short-term memory (LSTM) model, which takes the underlying prices for the past 20 days as an input instead of some pre-calculated volatility. Although my results aren't as good as theirs in terms of error metrics, the models still beat the BSM. However a caveat must be made that the prices generated by these models are not necessarily arbitrage-free as no condition is set in order for that to happen. Future work should focus on that but also on perfecting the models to increase their accuracy, estimate implied volatility (IV) [46] and the greeks.

Keywords: Option Pricing, Machine Learning, Artificial Neural Networks, Multi-Layer Perceptron, Long Short-Term Memory

JEL Codes: C00, G12, G13, G17

Contents

List of Figures

1 Introduction

The literature on pricing financial options is vast and complex. One can distinguish two broad approaches to option pricing methods: analytical ones, like the Black-Scholes-Merton (BSM) model or the Heston Model; or using numerical methods, such as Monte Carlo simulations or, more recently, using Machine Learning (ML). This thesis is about pricing vanilla European call and put options on the S&P 500 using ML. More specifically, the paper will use artificial neural networks (ANNs).

In this thesis I attempt to replicate the work done in Ke and Yang (2019) [25]. This means building three types of neural networks, two multi-layer perceptron (MLP) and one long short long-term memory (LSTM); using those networks to estimate option prices; calculating metrics based on both the predictions and the market prices, in order to ascertain how close the first are to the latter; and finally compare those metrics to the BSM model. It should be noted that these networks are not constrained in any way as to produce arbitrage-free prices. This is unlike the BSM model where the underlying assumptions enforce arbitrage-free price estimates. So, even if these networks have better metrics, more work needs to be done. Prices that allow for arbitrage opportunities are not useful.

To close this introduction I would like to point out that all the code used in this thesis is publicly available on my GitHub page.

2 Literature Review

This section will focus almost exclusively on literature using artificial neural networks (ANN) techniques to price European options. Malliaris and Salchenberger (1993) [27], and not Hutchinson, Lo and Poggio (1994) [21] as is often referenced, was the first entry in said literature. In that first paper, the authors created a 3-layer MLP network where the hidden (also called middle) layer had 4 neurons/nodes. The inputs to the network were those used in the BSM model: price of the underlying (S), strike price (K), time to maturity (T), volatility (σ) (more specifically they used the IV of at-the-money options) and the risk-free rate (r). Moreover, they also used as inputs the previous day prices of both the underlying and the option. And, in order to calibrate the ANN, they also needed the price of the option for the day in question. 6 months of data related to the S&P 100 were used. The results were mixed: "(...) for both in-and out-of-the-money prices, the neural network out-performs the Black-Scholes model in about 50% of the cases examined, as measured by the mean squared error." The authors also faced a number of problems that are common throughout the literature. "There are several limitations that may restrict the use of neural network models for estimation. There is no formal theory for determining optimal network topology, and therefore, decisions like the appropriate number of layers and middle-layer nodes must be determined using experimentation. The development and interpretation of neural network models requires more expertise from the user than traditional analytical models.. Training a neural network can be computationally intensive, and the results are sensitive to the selection of learning parameters, activation function, topology of the network, and the composition of the data set." [27]. Today, computers are more powerful, which makes the computational intensity less of a concern. Also, there are techniques that try to find the optimal hyper-parameters of a network such as gradient search.

Jumping to more recent literature, Hahn (2013) [16] uses ANNs to model either American equity options' prices or volatility (because it relates to option pricing) in the Australian market. So although its focus is not European options, it does contain some broad insights. It explains the prevalence of ANNs for option pricing: "As will be discussed in subsequent chapters, the focus of research in this area has been on artificial neural networks. While alternative machine learning techniques are available to researchers and practitioners, artificial neural networks are particularly well-suited to the type of data found in financial markets, such as large data sets of noisy data with non-linear relationships between the variables." The paper also does a comprehensive review of the literature up to the time of publication, from which some general notes can be drawn. Most papers using ML have had very small datasets (from tens of thousands to little over 100 thousand observations) compared to what we will use in this thesis (almost 14 million) or what was used in Ke and Yang (2019) (over 12,2 million) [25]. Several factors contribute to this, such as computational concerns or data availability (most of it is proprietary and is behind a paywall). Another trend is splitting the data between training, validation, and test sets. These are, respectively, the set that is used for training the model; the set that helps one know if there is overfitting of the training data or not; and the set used to test the fit/accuracy of the model, after it was fully trained with the training set (site 6). Many authors use bigger validation and test sets when compared to Ke and Yang (2019). Finally, in the earlier literature it is common to use a 3-month Treasury rate (TR) as a proxy for the risk-free rate. As a side remark, my original idea was to use the TRs whose maturity matched that of the option. However, after running into computational problems, I ended up using the 3-month rates.

Mezofi and Szabo (2018) [29] echo the problem of specifying an ANN. Although the use of ML to price options has become easier "thanks to the availability of several software packages for neural networks.", "One problem remains: designing the architecture of the neural network and avoiding overfitting the model." The authors continues: "Models that have more parameters can be overfitted more easily, so the number of the perceptrons and layers should be balanced between learning the important features and losing some precision because of overfitting. The learning rate determines how much to modify the parameters in each iteration; it is an important setting and must be set manually. Sometimes these metaparameters are decided based on validation errors; choosing them is more art than science. Picking the 'best' parameters can yield better results, but the accuracy gained during fine tuning usually diminishes, so the trained model is good enough to use after just a few trials." One can also filter out less representative options, for example those with lower liquidity, to improve the accuracy of the pricing model. The paper also explains why there is a need to move beyond the BSM model: "One of the biggest flaws of Black-Scholes is the mismatch between the model's volatility of the underlying option and the observed volatility from the market (the so-called implied volatility surface)." There are other empirical facts that the BSM model doesn't reproduce. However, one could use the BSM model as a good approximation and use ANNs to predict the volatilities to be used in the first. Finally, Malliaris and Salchenberger (1993) [27] argue that one should split the dataset between in-the-money and out-of-the-money options. But again, Mitra (2012) [30] argues that this increases the likelihood of overfitting, especially if the dataset is small.

Daniel Stafford (2018) [36] provides "an overview of the relevant previous literature regarding neural networks in option pricing." Moreover, it uses a convolutional neural network (CNN) instead of a MLP-styled NN. It compares the CNN with the BSM model regarding pricing and hedging performance of European call options on the S&P 500 stock index. The results show that the CNN is superior to both the BSM and MLP-styled models in terms of pricing performance. More precisely, the CNN outperforms the BSM model for all levels of "moneyness" and for mid-to-long-term options. For short-term options, the performance is on par with the BSM model.

Xugan Chen (2019) [8], like [25], builds LSTM models to price options. A distinct feature of the first is that Chen experimented with inputs related to option liquidity, macroeconomic factors and investor sentiment. The results pointed in the direction that these inputs don't add predictive power to the models. He builds an LSTM network with the following architecture:

- Input layer with 21 neurons, n feature and 12 timesteps, where n "(. . .) is the number of features, depending on how many categories of variables I use.";
- 2 LSTM layers (that are the hidden layers in this case) with 10 neurons each;
- Output layer with one neuron and tanh [49] activation function.

To avoid overfitting, the author also uses the dropout regularization method. Simply put, this method "approximates training a large number of neural networks with different architectures in parallel." [6].

Julie Johanne Uv (2019) [39] compares 5 Lévy models and one MLP model to the BSM model. A Lévy model incorporates a Lévy process. Contrary to the Geometric Brownian Motion present in the BSM, Lévy processes allow for jumps. In theory this should be important for financial applications as jumps are a feature of empirical time-series. However, this paper finds that the BSM model generally outperforms the Lévy models in terms of option pricing. Regarding the MLP model: "The MLP had similar error measures to the other models, which (I) consider interesting because of the small amount of data it was able to train on. Another interesting result, was the ability it had to generalize to an unobserved maturity where it actually performed better than all the other models." Julie Johanne Uv (2019) [39] scales the input and output data to a range between 0 and 1. The MLP is constituted by:

- Input layer;
- 5 hidden layers with different number of neurons. The first 4 use the ELU activation function and the last one uses Softplus [43];

- Output layer.

Daniel Bloch (2019) [5] covers (among many other things) the point of using the so called homogeneity hint when creating models. This means using the fact that a function is homogeneous with regards to all or some of it's parameters to "normalize" (or scale) those inputs and even remove some from the equation. A function is homogenous of degree k with regards to an argument if, when that argument is multiplied by a factor, the output of the function is multiplied by a k power of that factor. The BSM function is homogeneous of degree 1 with regards to the strike price and price of the underlying. The approximated function from an ANN may not be homogeneous, so we probably shouldn't assume so. However, according to Hahn (2013), if your dataset is small, having less variables/inputs can reduce the likelihood of overfitting. Also from [5]: "In practice, most of the trading occurs in short-term NTM (near-the-money) options and there is not sufficient data to train the network on ITM options with longer maturities." And most papers train their ANNs using very narrow intervals of parameters.

Andrey Itkin (2019) [22] used ANNs to tackle 2 problems: arbitrage-free option pricing and estimation of IVs to be used in the BSM model. The first one addresses a gap in the literature. Most ANN option pricing models don't consider the possibility (or lack thereof) of arbitrage opportunities.

For another comprehensive but more recent literature review, see Ruf and Wang (2020)[33]. It is important to highlight that the use of ANNs in finance is not restricted to option pricing. From that paper: "(...) an ANN can be used for many applications related to option pricing and hedging. In the most common form, an ANN learns the price of an option as a function of the underlying price, strike price, and possibly other relevant option characteristics. Similarly, ANNs might also be trained to learn implied volatility surfaces or optimal hedging ratios." Ruf and Wang (2020) also also mention lesser-known uses of ANNs: calibration of models for which that process is very time-consuming, solve partial differential equations related to option pricing, "approximate value functions that appear in dynamic programming, for example arising in the American option pricing problem", etc. The calibration use is especially interesting as it can make viable previously-proposed models once that calibration can be done in timely fashion. Examples of these models are non-parametric ones such as the Cont and Tankov model, and the Belomestny and Reiß method [20].

With regards to literature originating from Portugal, I couldn't find any paper that used ML, or more specifically, ANNs, to price European options. I did find 8 papers that applied ML techniques to other financial problems. For example: Inês Marquês (2019) [28] predicts stock prices, Sequeira et al. (2020) [14] prices American options and Tiago Dias (2015) [12] estimates (IV). I list all 8 papers in the bibliography: [28], [14], [12], [3], [32], [35], [41], [4].

To summarize, this thesis does a few things differently than most literature:

- Uses a huge dataset of European options;
- Doesn't use the homogeneity hint;
- Only excludes options whose price is 0 or whose time to maturity is 0. It doesn't exclude or segregate options depending on time to maturity or any type of "moneyness".

3 Theory and Data

3.1 Artificial Neural Networks

In the following section I will use the bold version of a letter to represent either a matrix (in uppercase) or a vector (in lowercase). The explanations were derived mainly from Liu et al. (2019) [26], sections 3.1 to 3.3; H. A. Trønnes (2018) [38], section 2.3; "Understanding Neural Networks" [42]; "Artificial neural network" [44] and Elouerkhaoui, Y. (2019) [13].

3.1.1 Artificial Neural Network (ANN)

At a macro level, ANNs are constituted by layers made up of artificial neurons. There are three types of layers: input, hidden and output. An ANN is composed by 1 input layer, 1 output layer and a variable (0, 1, 2 and so on. . .) number of hidden layers. Figure 3.1 illustrates a simple example of an ANN.

The number of neurons in the input and output layers are respectively, the number of input variables (also called features) and the number of output variables. Hidden layers are the layers between those other two and can have 1 or more neurons each. The exact number of neurons in the hidden layer needs to be thought in order to balance pros and cons ("The Hidden Layer" [34]). On one hand more nodes might lead to a better learning by the network. On the other, the more neurons the layer has, the longer the network might take to train and the risk of overfitting increases. The connections between the neurons are called edges. "Neurons and edges typically have a weight that adjusts as learning proceeds" ("Artificial neural network" [44]). This means that it can change as the network is trained on the data. So we will have a weight matrix $\mathbf{W^{(l)}}$ with $l \in [1, L]$, with L being the number of layers in the ANN, which can be a minimum of 2, in which case $l \in [1, 2]$. To be clear, each neuron or edge will have a weight for each connection they have with another neuron. The neurons in the hidden and output layers perform the calculation in equation (3.1). before transferring, or not, the resulting value to other neuron(s), depending on the type of neural network (NN) (e.g. ANN, recurrent neural network, CNN).

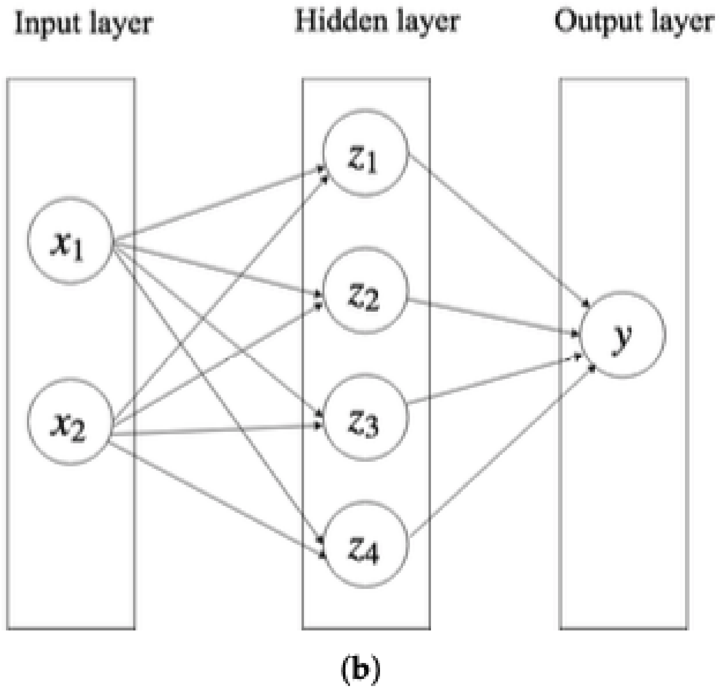
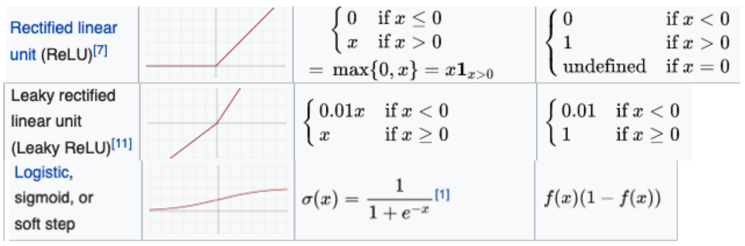


Figure 3.1: Example of an ANN, more precisely a multilayer perceptron.

$$z_j^{(l)} = \varphi^{(l)} \left(\sum_i w_{ij}^{(l)} z_i^{(l-1)} + b_j^{(l)} \right) \tag{3.1}$$

Where $z_j^{(l)}$ represents the output of the j-th neuron in the l-th layer; $j \in [1, N(l)]$, where N(l) is the number of neurons of layer l; and $l \in [1, L]$ as before. This means that the number of neurons varies according to the layer in question and that is quite important to understand. φ(l) represents the activation function of the l-th layer; $w_{i,j}^{(l)}$ represents the learnable weight associated with neurons i, where $i \in [1, N(i)]$, and j, for the (l-1)-th and l-th layers, respectively; and $b_j^{(l)}$ is the bias of the j-th neuron in the l-th layer. An activation functions adds non-linearity to the network which is useful if we are solving a problem which is not linear, like pricing an option ("Understanding Neural Networks" [42]). One can think of weights and biases (check "What is the role of the bias in neural networks?" [2] to better understand biases) as the slope and intercept of a linear function, respectively. However, $z^{(l)}{}_j$ will be the outputs of non-linear activation functions [2].

Note that, when l = 1, the formula above doesn't apply as the neurons in the input layer do not perform such calculation. $z_k^{(1)}$ is simply the value of the k-th input (transformed or not by an activation function), with $k \in [1, K]$, K being the number of inputs. And, when l = L, $z_m^{(L)}$ is the value of the m-th predicted output, with $m \in [1, M]$, M being the number of outputs for the problem at hand. There are many ways of going from l = 1 to l = L, depending on the type of NN. Also, as hinted before, input layers do not have a weight matrix or a bias vector. They may have activation functions though.



Rectified linear unit (ReLU)[7]		$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} = \max\{0, x\} = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$
Leaky rectified linear unit (Leaky ReLU)[11]		$\begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$
Logistic, sigmoid, or soft step		$\sigma(x) = \frac{1}{1+e^{-x}}$ [1]	$f(x)(1 - f(x))$

Figure 3.2: Examples of activation functions. Their graphs, formula and first derivative, respectively.

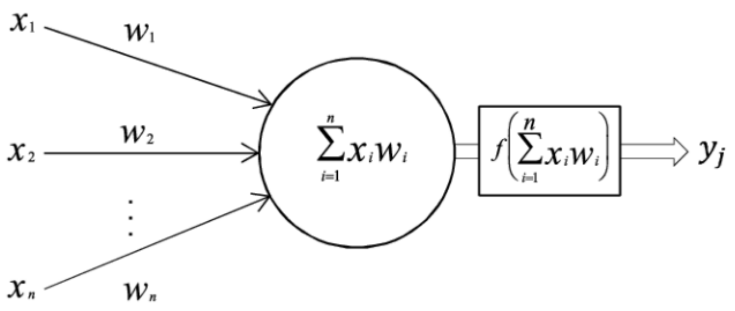


Figure 3.3: Visualization of the calculations explained above for a network with n inputs, one hidden neuron, one output neuron and no bias terms.

The objective of an ANN is to approximate a function, which is not available in closed form but only through sample pairs of given input data **X** and output data **Y** = f(**X**) [19]. We know that such networks can represent even highly nonlinear multi-dimensional functions, such as an option pricing function for example ([16]). Therefore, they "can be used as powerful universal function approximators without assuming any mathematical form for the functional relationship between the input variables and the output." ([26]). It is important to note that there are machine learning algorithms that work only with input data. Those are called unsupervised methods [38]. In this thesis I only deal with supervised methods. In the latter we supply K inputs and M outputs and expect the ANN to approximate the function that transforms $\mathrm{x_k}$ into $\mathrm{y_m}$. This is accomplished by finding the parameters (weights and biases) that make the distance between the actual outputs of the function we want to approximate, $\mathrm{f(x_k)}$, and the outputs generated by the network, $\mathrm{F(x_k}|\theta)$, as small as possible. Mathematically this can be formulated as the following optimization problem:

$$\arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta} \mid (\boldsymbol{x}, \boldsymbol{y})) \tag{3.2}$$

which means to find the parameters $\boldsymbol{\theta}$ that minimize

$$L(\boldsymbol{\theta} \mid (\boldsymbol{x}, \boldsymbol{y})) \tag{3.3}$$

and where

$$\boldsymbol{x} = \left(\mathrm{x}^{(1)}, \mathrm{x}^{(2)}, \ldots, \mathrm{x}^{(\mathrm{K})}\right) \tag{3.4}$$

$$\boldsymbol{y} = \left(\mathrm{y}^{(1)}, \mathrm{y}^{(2)}, \ldots, \mathrm{y}^{(\mathrm{M})}\right) = f(\boldsymbol{x}) \tag{3.5}$$

$$\boldsymbol{\theta} = \left(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\right) \tag{3.6}$$

$$L(\boldsymbol{\theta}) := D(f(\boldsymbol{x}), F(\boldsymbol{x} \mid \boldsymbol{\theta})) \tag{3.7}$$

$L(\boldsymbol{\theta})$ is a loss (also called cost or error) function which is defined as the distance (hence the use of the letter D) between $\mathrm{f}(\boldsymbol{x})$ and $\mathrm{F}(\boldsymbol{x}|\boldsymbol{\theta})$. This distance can be measured in different ways, for example using the p-norm [48]. Let $\mathrm{p}\geq 1$ be a real number and n be a positive integer, representing the total number of observations in one's dataset. The p-norm (also called ℓ_{p}-norm) of vector $\boldsymbol{v} = (v^{(1)}, \ldots, v^{(n)})$ is:

$$\|\mathbf{v}\|_p := \left(\sum_{i=1}^{n} |v_i|^p\right)^{1/p} \tag{3.8}$$

In the case of an ANN, $\mathrm{L}(\theta)$ will be the p-norm of $f(\boldsymbol{x}) - F(\boldsymbol{x} \mid \boldsymbol{\theta})$. More specifically, we will use the average of the squared 2-norm of $f(\boldsymbol{x}) - F(\boldsymbol{x} \mid \boldsymbol{\theta})$. This is called the mean squared error (MSE), and is defined as:

$$\mathrm{MSE}(\mathrm{f}(\boldsymbol{x}) - \mathrm{F}(\boldsymbol{x} \mid \boldsymbol{\theta})) = \frac{\left[\sum_{\mathrm{i}=1}^{\mathrm{n}} \left\{\mathrm{f}\left(\mathrm{x_i}\right) - \mathrm{F}\left(\mathrm{x_i} \mid \boldsymbol{\theta}\right)\right\}^2\right]^{2/2}}{n} = \frac{\sum_{\mathrm{i}=1}^{\mathrm{n}} \left\{\mathrm{f}\left(\mathrm{x_i}\right) - \mathrm{F}\left(\mathrm{x_i} \mid \boldsymbol{\theta}\right)\right\}^2}{n} \quad (3.9)$$

The parameters of an ANN ,$\boldsymbol{\theta}$, are updated so that the output(s) of the network approaches the desired value(s). ANNs are an example of supervised learning in ML because we want the network to map given values [f(x_k)].

To complete the explanation of ANNs it is important to talk about hyper-parameters (not to be confused with the network's parameters $\boldsymbol{\theta}$). Examples of such parameters are: the number of hidden layers; the number of neurons in those layers; the batch size, which is the number of observations in the input set that is used for network training; the number of epochs; the activation functions; the learning rate(s) (again, learning rate doesn't have to be constant); etc. A note on the learning rate: "A large learning rate leads to fluctuations around a local minimum, and sometimes even to divergence. Small learning rates may cause an inefficiently slow training stage. It is common practice to start with a large learning rate and then gradually decrease it until a well-trained model has resulted." [26].

As mentioned in the literature review, finding the optimal hyper-parameters is not straightforward. There are techniques to do it such as grid search and randomized grid search. The first method consists of establishing a range of values that each hyper-parameter can take and then testing the network's loss for each combination of those values. This can be quite time-consuming. The second technique is similar, but it selects randomly from the range in each iteration and tests the network with the selected values.

3.1.2 Multilayer Perceptron (MLP)

An MLP is a type of ANN that follows 3 steps: initialization (Init), feed-forward (FF) and back-propagation (BP) (note that Init and BP are common features of all types of ANNs). There are multiple optimization algorithms that do BP. In this thesis we will use the Adam optimizer (explained below). These optimization algorithms start with initial values for the weights and biases"(. . .) and move in the direction in which the loss function decreases" [26]. The initial values for weights and biases can be determined in numerous ways. For example, from "Layer weight initializers" [1]: extracting random values from a normal distribution, or from a uniform distribution; set as 1s or 0s; using the Glorot normal initializer, also called Xavier normal initializer, etc. This is the Init step.

Returning to the formula in equation (3.1), the process of going from $\boldsymbol{X}$ (inputs) to $\boldsymbol{Y}$ (outputs) traveling one way from a layer to the next is called feedforward. It is also exemplified by figure 3.1.

The following explanation follows Elouerkhaoui, Y. (2019) [13]. There are two types of gradient descent (GD) methods: batch GD and stochastic GD (SGD). The first uses the whole dataset to find the optimal $\boldsymbol{\theta}$, while the second uses a portion/batch of that dataset. Adam is a type of SGD optimizer. The latter computes the gradient of the loss function and then updates $\boldsymbol{\theta}$, using a different batch for each epoch, trying to make that loss smaller each time. An epoch is a complete training cycle of the network (Init, FF and BP steps for the first epoch; FF and BP steps for the following epochs). The SGD method “divides the training set into non-overlapping, roughly equal subsets called batches.” [39]. So in each epoch a different batch is used to train the network. The computation of the gradient is done using the chain rule to compute the derivatives of the loss function “backwards”. Meaning we start with the derivatives of the loss function with respect to both $\boldsymbol{W}^{(L)}$ and, $\boldsymbol{b}^{(L)}$ and move towards the derivatives regarding the first hidden layer, hence the name back-propagation.

Define $\boldsymbol{a}^{(l)}$ as the vector of values calculated in the previous layer, before the application of the activation function. We call this vector pre-activation:

$$\mathbf{a}^{(\mathrm{l})} = \mathbf{W}^{(\mathrm{l})}\mathbf{z}^{(\mathrm{l}-1)} + \mathbf{b}^{(\mathrm{l})} \tag{3.10}$$

where $\boldsymbol{z}^{(l)}$ is the vector, called activation, that results from applying an activation function to that pre-activation and then aggregating (in a vector) the resulting $z_j^{(l)}$ for all j € [1, N(l)] neurons of layer l. Mathematically:

$$\boldsymbol{z}^{(\mathrm{l})} = \varphi^{(\mathrm{l})}\left(\mathbf{a}^{(\mathrm{l})}\right) \tag{3.11}$$

Let’s also define the error term as

$$\delta_j^{(l)} = \partial \mathrm{L}/\partial \mathrm{a}_j^{(l)} \tag{3.12}$$

which is related to the derivatives with respect to the weights and biases through the chain rule as:

$$\frac{\partial L}{\partial w_{i,j}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}}\frac{\partial a_j^{(l)}}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)} \tag{3.13}$$

and

$$\frac{\partial \text{Ł}}{\partial b_j^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}}\frac{\partial a_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{3.14}$$

which should be clear from equation 3.11.

Starting from the last layer L, we compute $\delta_j^{(l)} = \partial \mathrm{L}/\partial \mathrm{a}_j^{(l)}$, for $j \in [1, N(l)]$:

$$\delta_j^{(L)} = \frac{\partial L}{\partial a_j^{(L)}} = \frac{\partial L}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial a_j^{(L)}} = \frac{\partial L}{\partial z_j^{(L)}} f_L' \left(a_j^{(L)} \right) = \left(z_j^{(L)} - y_j \right) f_L' \left(a_j^{(L)} \right) \tag{3.15}$$

where f_L' is the derivative of the activation function, called output gradient, for layer L; y_j are the values we want to approximate; $z_j^{(L)}$ are the network approximations of y_j; $y_j - z_j^{(L)}$ is called the output error. Then we backpropagate via the chain rule for $j \in [1, N(l-1)]$, $l \in [1, L]$:

$$\delta_j^{(l-1)} = \frac{\partial L}{\partial a_j^{(l-1)}} = \sum_{i=1}^{N^{(l)}} \frac{\partial L}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial a_j^{(l-1)}} = \sum_{i=1}^{N^{(l)}} \delta_i^{(l)} \frac{\partial a_i^{(l)}}{\partial a_j^{(l-1)}} \tag{3.16}$$

The summations in 3.16 are a result of layer L having $N^{(L)}$ neurons, each possessing a corresponding input or signal each, and their respective weights. The term $\partial \mathrm{a}_j^{(l)}/\partial \mathrm{a}_j^{(l-1)}$ is given by:

$$\begin{aligned} \frac{\partial a_j^{(l)}}{\partial a_j^{(l-1)}} &= \frac{\partial}{\partial a_j^{(l-1)}} \left(\sum_{k=1}^{N^{(l-1)}} w_{i,k}^{(l)} Z_k^{(l-1)} + b_i^{(l)} \right) \\ &= \frac{\partial}{\partial a_j^{(l-1)}} \left(\sum_{k=1}^{N^{(l-1)}} w_{i,k}^{(l)} f_{l-1} \left(a_k^{(l-1)} \right) + b_i^{(l)} \right) \\ &= \frac{\partial}{\partial a_j^{(l-1)}} \left(w_{i,j}^{(l)} f_{l-1} \left(a_j^{(l-1)} \right) \right) = w_{i,j}^{(l)} f_{l-1}' \left(a_j^{(l-1)} \right) \end{aligned} \tag{3.17}$$

Thus the delta at the (l-1)-th layer is calculated as

$$\delta_j^{(l-1)} = \sum_{i=1}^{N^{(l)}} \delta_i^{(l)} w_{i,j}^{(l)} f_{l-1}' \left(a_j^{(l-1)} \right) \tag{3.18}$$

Notice that $\delta_j^{(L)}$ (error term for the output layer) and $\delta_j^{(l-1)}$ (error term of the hidden layers) have different formulas. Now, following the explanation in "Understanding Neural Networks" [42], weights and biases are updated using following the formulas, respectively:

$$\Delta \mathbf{W}^{(l)} = -\eta \delta^{(l)} \mathbf{z}^{(l-1)} \tag{3.19}$$

and

$$\Delta \mathbf{b}^{(l)} = -\eta \delta^{(l)} \tag{3.20}$$

$\mathbf{W}^{(l)}$ are the weights of layer l, so $\Delta\mathbf{W}^{(l)}$ is the change in those weights; η is a learning rate, which is a positive number that can be constant or variable during the iterations; $\mathbf{z}^{(l-1)}$ is the output of the previous layer (layer l-1) and the input to layer l. This process is repeated E-1 (because we already did one backpropagation) times, where E equals the total number of epochs set.

Next is a numerical example of the feedforward and backpropagation steps, adapted from "Understanding Neural Networks" [42]:

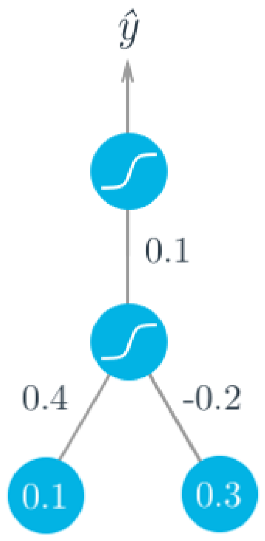


Figure 3.4: Depiction of a three-layer ANN

Let's assume an MLP (as depicted in figure 3.4) with 1 input layer, 2 input neurons (because there are 2 inputs), 1 hidden layer, 1 output layer; there are no bias terms, for the sake of simplicity; y = 1; learning rate = 0,5; and only one activation function is used, the logistic function, described in "Introduction to Deep Learning Networks" [47]. First we calculate the input of the hidden layer h as

$$a^{(h)} = z^{(0)} = \left(\sum_{j=1}^{2} w_j^{(h)} z_j^{(0)}\right) = 0,1 \times 0,4 - 0,2 \times 0,3 = -0,02 \tag{3.21}$$

then we calculate the output of that same layer as

$$f\left(a^{(h)}\right) = z^{(h)} = \text{logistic}(-0,02) = 0,495 \tag{3.22}$$

and then we calculate the output of the output layer (y) (the output of the network) as

$$z^{(y)} = f\left(w^{(y)} * z^{(h)}\right) = \text{logistic}(0,1 \times 0,495) = 0,512 \tag{3.23}$$

Notice that here we don't include a summation for each neuron because there is only 1 in the output layer.

With these calculations we can start the backpropagation process and update the weights for the hidden and output layers. The derivative of the logistic function, applied to arbitrary value c, is:

$$f'(c) = f(c)[1 - f(c)] \tag{3.24}$$

So, we can calculate the error term of the output unit as

$$\delta^{(y)} = \left(z^{(y)} - y\right) f'\left(w^{(y)} \times a\right) = (0,512 - 1) \times (0,512) \times (1 - 0,512) = -0,122 \tag{3.25}$$

We also calculate the error term of the hidden layer as

$$\delta^{(\mathrm{h})} = \mathrm{w}^{(\mathrm{y})} \times \delta^{(\mathrm{y})} \times \mathrm{f}'\left(\mathrm{a}^{(\mathrm{h})}\right) = 0,1 \times (-0,122) \times 0,495 \times (1 - 0,495) = -0,003 \tag{3.26}$$

Now we can calculate the change in the weights. First the change in the weight of the output layer as

$$\Delta \mathrm{w}^{(\mathrm{y})} = -\eta \delta^{(y)} \mathrm{z}^{(\mathrm{h})} = 0,5 \times (-0,122) \times 0,495 = 0,0302 \tag{3.27}$$

Finally, we calculate the change in the weights of the hidden layer as

$$\Delta \mathbf{w}^{(\mathrm{h})} = -\eta \delta^{(\mathrm{h})} \mathrm{z}^{0} = [(-0,5)\times(-0,003)\times 0,1; (-0,5)\times(-0,003)\times 0,3)] = (0,00015; 0,00045) \tag{3.28}$$

This process of updating weights is then repeated as explained in the theory.

3.1.3 Long Short-Term Memory (LSTM)

The following explanation is not exhaustive and the reader is referred to Staudemeyer and Morris (2019) [37] (and many it's references) for a complete understanding of RNNs and LSTMs.

Recurrent Neural Network (RNN)

A LSTM is a type of recurrent neural network (RNN). "RNNs, are DLN systems that are designed to detect patterns present in sequences. This makes them well suited to solve "prediction" problems, as opposed to regular DLNs that are oriented to solving classification problems. An example of a prediction problem is predicting the next word in a sentence(...)" [40]. DLNs are deep-learning networks, in contrast with "shallow" networks. Examples of DLNs are ANNs, RNNs, CNNs and so on [31]. "As a result the RNN is able to detect patterns that are spread over time(...)" [40]. In our case we are interested in predicting the "next" price of an option. More information about RNNs can be found in chapter 13 of [40].

Given what I explained so far, one must wonder if a LSTM model really is a good fit for the problem at hand. It might not be because option values are typically assumed to not depend on information from the past by most common option pricing models. However, since it is used in Ke and Yang (2019), we will test it anyway. Moreover, there is evidence of dependence of current volatility on past volatility [9].

"LSTMs were designed with the objective of solving the Vanishing Gradients Problem (VGP) that plagues RNNs. They have been very successful in this regard, indeed almost all of the successful applications of RNNs in recent years have been with LSTMs rather than with plain RNNs." [40]. The VGP occurs when "(...) the gradients become progressively smaller and get close to zero as the number of RNN stages increases." [40]. There is also the inverse problem called exploding gradients problem (EGP) which LSTMs also address [37]. Basically, in the EGP the gradients became bigger and bigger as the model iterates.

To understand how a LSTM works we have to first understand how RNNs do. The traditional DLN / ANN performs:

$$\boldsymbol{Y} = f(\boldsymbol{X}) \tag{3.29}$$

where $\boldsymbol{Y}$ is the output matrix and $\boldsymbol{X}$ is the input matrix. In the RNN world (and from here on out I will stop differenciating between matrices, vectors and scalars because the sources don't do it), we have:

$$Y_i = f_i\left(X_1, \ldots, X_i\right) \tag{3.30}$$

where i is the index for a discrete moment in time. So 0 $\leq$ i $\leq$ n, n being the index of the last moment in time for the data we have available. “Furthermore the output vector Y_i depends not just on the input vector X_i, but on past values of X as well.” [40]. RNNs use a hidden variable / hidden state / state variable Z_i such that:

$$Z_{i+1} = w_{i+1}\left(Z_i, X_i\right) \tag{3.31}$$

where w_{i+1} are the weights associated with time i+1 and variables Z_i and X_i. Assuming that the vectors Z_i and X_i can be combined linearly, that the weights do not change with the time step [40] and applying activation functions f and h, we get the following equations:

$$Z_{i+1} = f\left(WZ_i + UX_i\right) \tag{3.32}$$

$$Y_{i+1} = h\left(VZ_{i+1}\right) \tag{3.33}$$

where U and V are weight matrices as will be discussed below.

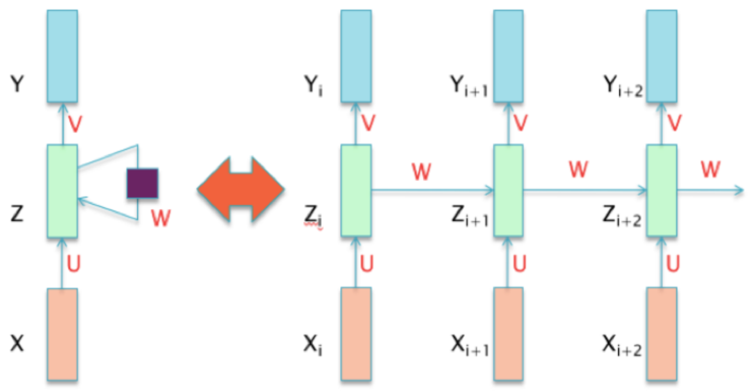


Figure 3.5

Figure 3.5 is taken from [40] (which I will cite multiple times in this paragraph) and provides an intuitive idea of a 3-layer RNN with feedforward, which allows the use of back-propagation. At each point in time, the network will be affected by the hidden variable from the previous point in time. To be clear, “The weight matrix U connects the nodes in the

Input Layer with those in the Hidden Layer”, “The weight matrix W connects the nodes in the Hidden Layer with the same set of nodes, but at the previous time instant.” The square block on the on the left-hand side of the figure “represents a time delay of one unit, and represents the fact that at time n, the nodes in that layer have as one of their inputs, the values that they had at time n - 1.” Finally, “The weight matrix V connects the nodes in the Hidden Layer with those in the Output Layer.”

Diving deeper, with RNNs we need to modify both the feedforward and the backpropagation processes used to train the network. I will present the most popular method, the so called backpropagation through time (BPTT), assuming a 3-layer network (input, hidden and output layers) for the sake of simplicity. The explanation follows that of [40].

- Step 1: "The input vector $X(1) = (x_1(1), \ldots, x_N(1))$ is applied to the system, resulting in the hidden state activation vector $Z(1) = (z_1(1), \ldots, z_P(1))$ and the output vector $Y(1) = (y_1(1), \ldots, y_K(1))$ as per the formulae":

  $$a_i^Z(1) = \sum_{j=1}^{N} u_{ij} x_j(1) + b_i^Z, \quad 1 \leq i \leq P \tag{3.34}$$

  $$z_i(1) = f\left(a_i^Z(1)\right), \quad 1 \leq i \leq P \tag{3.35}$$

  $$a_k^Y(1) = \sum_{j=1}^{P} v_{kj} z_j(1) + b_k^Y, \quad 1 \leq k \leq K \tag{3.36}$$

  where u_{ij} is the weight associated with neuron i for the input j (vector index j) in the hidden layer and v_{kj} is the weight associated with hidden layer output j for the output k. P represents the number of neurons in the hidden layer and K the number of outputs of the network. The loss function $\mathcal{L}(1)$ is computed using the output vector Y(1) and the target vector T(1).

- Step 2: "The input vector X(2) and the hidden state vector Z(1) are applied applied to the system, resulting in the hidden state vector Z(2), the output vector Y(2) and the loss L(2)."

  $$a_i^Z(2) = \sum_{j=1}^{N} u_{ij} x_j(2) + \sum_{j=1}^{P} w_{ij} z_j(1) + b_i^Z, \quad 1 \leq i \leq P \tag{3.37}$$

$$z_i(2) = f\left(a_i^Z(2)\right), \quad 1 \leq i \leq P \tag{3.38}$$

$$a_k^Y(2) = \sum_{j=1}^{P} v_{kj} z_j(2) + b_k^Y, \quad 1 \leq k \leq K \tag{3.39}$$

$$y_k(2) = \frac{a_k^Y(2)}{\sum_j a_j^Y(2)}, \quad 1 \leq k \leq K \tag{3.40}$$

where w_{ij} is the weight associated with the neuron i for the hidden state j for the previous period and u_{ij} is the weight associated with the neuron i of network input j for the current period.

- "Step 2 is repeated M - 2 times to compute the vectors Z(j), $3 \leq j \leq M$ and Y(j), $3 \leq j \leq M$ until the remaining input vectors (X(3), ..., X(M)) have been processed". This constitutes the forward pass of the BPTT algorithm.

Before explaining the backpropagation part we need to define a few concepts, the gradients and error terms for the output and hidden layers, respectively:

$$\Delta^Y(m) = \left(\delta_1^Y(m), \ldots, \delta_K^Y(m)\right), \quad 1 \leq m \leq M \tag{3.41}$$

where

$$\delta_k^Y(m) = \frac{\partial L}{\partial a_k^Y(m)}, \quad 1 \leq k \leq K \tag{3.42}$$

and

$$\Delta^Z(m) = \left(\delta_1^Z(m), \ldots, \delta_P^Z(m)\right), \quad 1 \leq m \leq M \tag{3.43}$$

where

$$\delta_i^Z(m) = \frac{\partial L}{\partial a_i^Z(m)}, \quad 1 \leq i \leq P \tag{3.44}$$

So the backpropagation goes as following:

- Compute the gradients $\Delta^Y(m), 1 \leq m \leq M$ using the formula

$$\delta_k^Y(m) = y_k(m) - t_k(m), \quad 1 \leq k \leq K, \quad 1 \leq m \leq M \tag{3.45}$$

In order to derive this equation, recall that $L = \sum_{m=1}^{M} \mathcal{L}(m)$, so that

$$\frac{\partial L}{\partial a_k^Y(m)} = \sum_{j=1}^{M} \frac{\partial L}{\partial \mathcal{L}(j)} \frac{\partial \mathcal{L}(j)}{\partial a_k^Y(m)} = \frac{\partial L}{\partial \mathcal{L}(m)} \frac{\partial \mathcal{L}(m)}{\partial a_k^Y(m)} = \frac{\partial \mathcal{L}(m)}{\partial a_k^Y(m)} = y_k(m) - t_k(m) \quad (3.46)$$

We used the fact that $\frac{\partial \mathcal{L}(j)}{\partial a_k^Y(m)}$ except when j = m when deriving the equation above, because we are differentiating the loss function at time j by the pre-activation at time m.

- Similarly it is possible to show that

$$\frac{\partial L}{\partial a_i^Z(m)} = \frac{\partial \mathcal{L}(m)}{\partial a_i^Z(m)}, \quad 1 \leq i \leq P, 1 \leq m \leq M \quad (3.47)$$

- "The gradient of the Loss Function with respect to weight matrices U, V, W is then given by the sum of the gradients across each of the stages, as shown below":

$$\frac{\partial L}{\partial u_{ij}} = \sum_{m=1}^{M} x_j(m) \delta_i^Z(m) \quad 1 \leq j \leq N, \quad 1 \leq i \leq P \quad (3.48)$$

$$\frac{\partial L}{\partial u_{ij}} = \sum_{m=1}^{M} \frac{\partial L}{\partial a_i^Z(m)} \frac{\partial a_i^Z(m)}{\partial u_{ij}} = \sum_{m=1}^{M} x_j(m) \delta_i^Z(m) \quad (3.49)$$

"A similar set of equations can be derived for the gradients of the Loss Function with respect to the bias parameters, and is left to the reader".

Finally, with those results we can change the respective weights using the formula:

$$\Delta \mathrm{S}_{(j)} = -\eta \times \mathrm{dO/ds_{ij}} \times \theta\,(\mathrm{a_{j-1}}) \quad (3.50)$$

for each of the 3 weight vectors. S(j) represents the weights of neuron j; η is a learning rate; and $\theta(a_(j-1))$ is the output of neuron j from the previous layer.

Figure 3.7 is an example of an RNN cell or hidden layer. Instead of the tanh activation function we can have a generic one θ. W are the weights related to the state variables from different time steps and U are the weights related to the input variable and the state variable of the same time step.

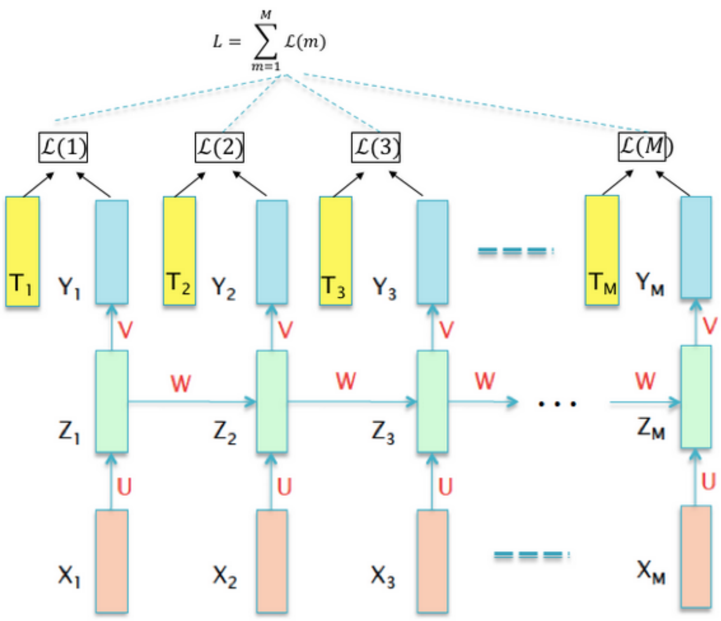


Figure 3.6: Illustration of BPTT [40]

Long Short-Term Memory (LSTM)

Figure 3.8 shows us the different gates used in an LSTM:

- Input gate composed by both the sigmoid of i and tanh of g;
- Forget gate composed of the sigmoid of f;
- Output gate composed of the sigmoid of o.

In the left diagram of figure 3.9 we can see how in a RNN the signals of past periods lose strength as we move further away in time. The lighter the shading, the less sensitive the network is to inputs of previous time periods. In a LSTM we can keep that signal strength.

Comparing RNNs to LSTMs, the hidden layers are more complicated as there is now a cell memory C_t. "The LSTM Cell operates on the Cell Memory $C_{(t-1)}$ from the previous stage, and generates the next Cell Memory C_t." Another difference is the addition of gates. "Indeed the vectors f_t, i_t and o_t are all used to generate gating values that regulate the flow of data in the LSTM Cell. In turn, f_t, i_t and o_t are computed using three new pairs of weight parameter matrices (W^f, U^f),(W^i, U^i) and (W^o, U^o) respectively."

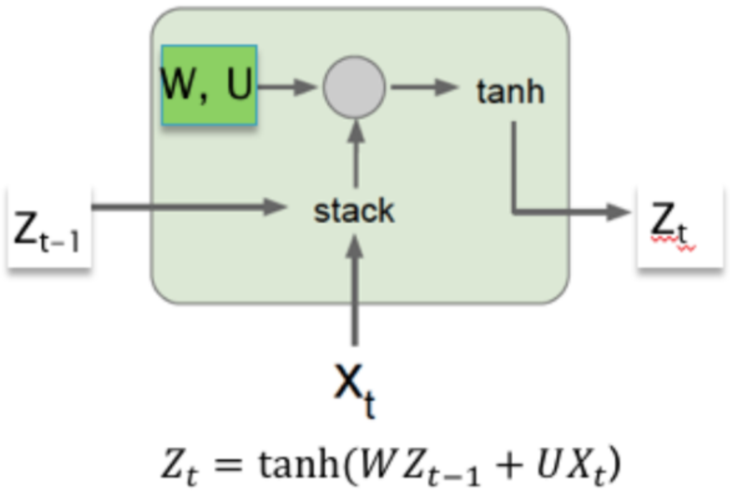


Figure 3.7: Example of an RNN cell

The symbol $\odot$ denotes element-wise multiplication (or Hadamard product) [45] as opposed to the typical matrix multiplication. So, in order to compute the state variable Z_t we do the following [40].

- Compute the gates:
  - Input gate

$$i_t = \sigma\left(W^i X_t + U^i Z_{t-1}\right) \tag{3.51}$$

  - Forget gate

$$f_t = \sigma\left(W^f X_t + U^f Z_{t-1}\right) \tag{3.52}$$

  - Output/Exposure gate

$$o_t = \sigma\left(W^o X_t + U^o Z_{t-1}\right) \tag{3.53}$$

"Note that in all these formulae, the function σ is applied on a componentwise basis."

- Update cell memory value:

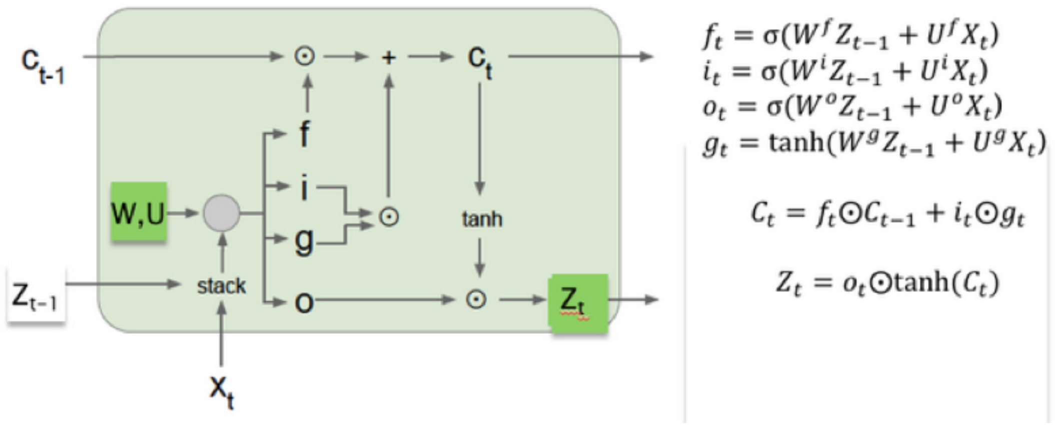

C_{t-1}
C_t
W,U
stack
Z_{t-1}
X_t
f
i
g
o
tanh
Z_t
$f_t = \sigma(W^f Z_{t-1} + U^f X_t)$
$i_t = \sigma(W^i Z_{t-1} + U^i X_t)$
$o_t = \sigma(W^o Z_{t-1} + U^o X_t)$
$g_t = \tanh(W^g Z_{t-1} + U^g X_t)$
$C_t = f_t \odot C_{t-1} + i_t \odot g_t$
$Z_t = o_t \odot \tanh(C_t)$

Figure 3.8: Example of a LSTM cell

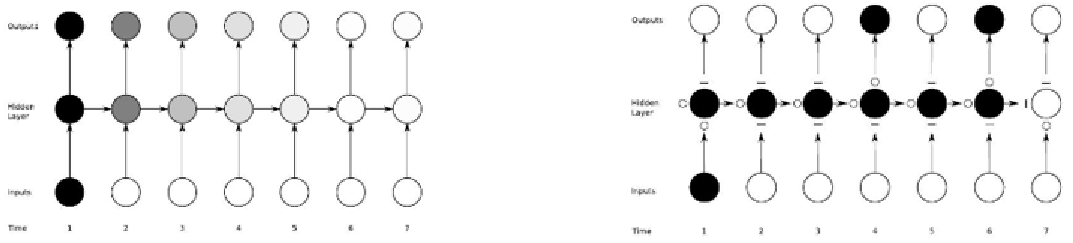


Figure 3.9: Illustration of the VGP and LSTM

– "Compute contribution to Cell Memory from current state X_t while taking the contribution from the previous Hidden State Z_{t-1} into account:"

$$\tilde{C}_t = g_t = \tanh\left(W^g X_t + U^g Z_{t-1}\right) \tag{3.54}$$

– "Combine contribution from current state $\tilde{C}_t$ with old memory C_{t-1} to get new memory C_t using the gating functions f_t and i_t:"

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{3.55}$$

– "Compute the new Hidden State vector using the new memory C_t and the gating function o_t :"

$$Z_t = o_t \odot \tanh(C_t) \tag{3.56}$$

Equation (3.55) is very important as it allows us to deepen our knowledge about how an LSTM behaves. Consider these 4 cases [40]:

1. C_t changes in value in response to new inputs X_t : This happens when $i_t = f_t = 1$, and the cell state increments or decrements at each time step.
2. C_t holds the old value from the previous stage: This happens when $i_t = 0, f_t = 1$, and results in $C_t = C_{t-1}$.
3. C_t resets to the value $\tilde{C_t}$: This happens when $i_t = 1, f_t = 0$ and results in $C_t = \tilde{C_t} = \tanh(W^g X_t + U^g Z_{t-1})$. Note that this is not a complete reset, since the prior history is still be propagated through Z_{t-1}.
4. C_t resets to zero: This happens when $i_t = 0,\ f_t = 0$ and results in a complete reset.

But remember that what we care about is Z_t so we have to look at equation (3.56) [40]:

1. Changes in value in response to new inputs X_t : This happens when $i_t = f_t = o_t = 1$.
2. Holds the old value from the previous stage: This happens when $i_t = 0, f_t = 1, o_t = 1$, and results in $Z_t = Z_{t-1}$.
3. Resets to the (tanh) value of the new cell state: This happens when $i_t = 1, f_t = 0, o_t = 1$ and results in $Z_t = \tanh\left(\tilde{C_t}\right)\big) = \tanh(\tanh(W^g X_t + U^g Z_{t-1}))$. Note that Z_t still retains memory of the prior state, hence it is not a complete reset.
4. Resets to zero: This happens when $o_t = 0$ or when $i_t = 0, f_t = 0$ and results in $Z_t = 0$, resulting in a full reset.

Finally, Z_t is "transferred" to the next layer. This is the feedforward process.

Now for the backpropagation process I follow the explanation in [10]. We want to find the gradients for the weights at each time step. Let the input at time t in the LSTM cell be x_t, the cell state from time $t-1$ and t be c_{t-1} and c_t and the output for time $t-1$ and t be h_{t-1} and h_t. The initial value of c_t and h_t at $t = 0$ will be zero.

- Step 1: Initialization of the weights. The weights for the different gates are:
  - Input gate: $w_{xi}, w_{xg}, b_i, w_{hj}, w_g, b_g$;
  - Forget gate: w_{xf}, b_f, w_{hf};

  - Output gate: w_{x_0}, b_0, w_{h_0}

- Step 2: Passing through different gates.
  - Passing through input gate:

$$\begin{aligned} & z_g = w_{xg} \times x + w_{hg} \times h_{t-1} + b_g \\ & g = \tanh(z_g) \\ & z_j = w_{xi} \times x + w_{hi} \times h_{t-1} + b_i \\ & i = \text{sigmoid}(z_i) \end{aligned} \tag{3.57}$$

    Input gate out = igo = $g \times i$

  - Passing through forget gate:

$$\begin{aligned} & Z_f = w_{xf} \times x + w_{hf} \times h_{t-1} + b_f \\ & f = \text{sigmoid}(Z_f) \end{aligned} \tag{3.58}$$

    Forget gate out = fgo = f

  - Passing through the output gate:

$$\begin{aligned} & z_0 = w_{x0} itemx + w_{ho} \times h_{t-1} + b_0 \\ & o = \text{sigmoid}(z_0) \end{aligned} \tag{3.59}$$

    Output gate out = ogo = o

- Step 3 : Calculating the output of the hidden layer h_t and current cell state c_t.

$$c_t = (c_{t-1} \text{ fgo }) + \text{ igo} \tag{3.60}$$

$$h_t = \text{ ogo } \times \tanh(c_t) \tag{3.61}$$

- Step 4 : Calculating the gradient through back propagation through time at time stamp t using the chain rule.

  Let the gradient that passes through the LSTM cell be:

$$\delta LSTM = \mathrm{dE/dh_t} \tag{3.62}$$

If we are using MSE for error then, $\delta LSTM = (y - h(x))$. y is the true value, h(x) is the predicted value.

– Gradient with respect to output gate

$$\begin{aligned} &\mathrm{dE/do} = (\mathrm{dE/dh_t}) \times (\mathrm{dh_t/do}) = \delta LSTM \times (\mathrm{dh_t/do}) = \\ &= \delta LSTM \times \tanh(\mathrm{c_t}) \end{aligned} \tag{3.63}$$

– Gradient with respect to c_t

$$\begin{aligned} \mathrm{dE/dc_t} &= (\mathrm{dE/dh_t}) \times (\mathrm{dh_t/dc_t}) = E_{-}\,\mathrm{delta} * (\mathrm{dh_t/dc_t}) \\ &= \delta LSTM \times o \times \left(1 - \tanh^2(\mathrm{c_t})\right) \end{aligned} \tag{3.64}$$

– Gradient with respect to input gate $\mathrm{dE/di}, \mathrm{dE/dg}$

$$\begin{aligned} \mathrm{dE/di} =& (\mathrm{dE/di}) \times (\mathrm{dc_t/di}) = \\ &= \delta LSTM \times o \times \left(1 - \tanh^2(\mathrm{c_t})\right) \times\ \mathrm{g} \end{aligned} \tag{3.65}$$

and, similarly

$$\mathrm{dE/dg} = \delta LSTM \times \mathrm{o} \times \left(1 - \tanh^2(\mathrm{c_t})\right) \times \mathrm{i} \tag{3.66}$$

– Gradient with respect to forget gate

$$\begin{aligned} &\mathrm{dE/df} = \delta LSTM \times (\mathrm{dE/dc_t}) \times (\mathrm{dc_t/dt})\,\mathrm{t} = \\ &= \delta LSTM \times o * \left(1 - \tanh^2(\mathrm{c_t})\right) \times \mathrm{c_{t-1}} \end{aligned} \tag{3.67}$$

– Gradient with respect to c_{t-1}

$$\begin{aligned} &\mathrm{dE/dc_t} = \delta LSTM \times (\mathrm{dE/dc_t}) \times (\mathrm{dc_t/dc_{t-1}}) = \\ &\qquad = \delta LSTM \times o \times \left(1 - \tanh^2(c_t)\right) \times f \end{aligned} \tag{3.68}$$

– Then, the gradients associated with the weights are 12 in total and their formulas are in [10]

– To finish the training process we need the change all those weights. This is done in a similar fashion to what I explained for the MLPs:

$$W^{new} = W^{old} - \lambda * \delta W^{old} \tag{3.69}$$

Figure 3.10 is an example from "The Unreasonable Effectiveness of Recurrent Neural Networks" [24] of an RNN / LSTM with 4 neurons (corresponding to the 4 different characters in “helo”) in both the input and output layers and 3 neurons in the hidden layer. The idea is that we feed the network one of those 4 characters, and the output will give us a probability distribution for the character that should follow that input character. Ideally this network is calibrated (weights changed) until the input sequence of characters “hell” gives us the output sequence ello” to form the word “hello”. In the input layer, neuron i, a 1 means we are inputting the i-th character of the 4 character sequence “helo”, i ∈ [1, 2, 3, 4]

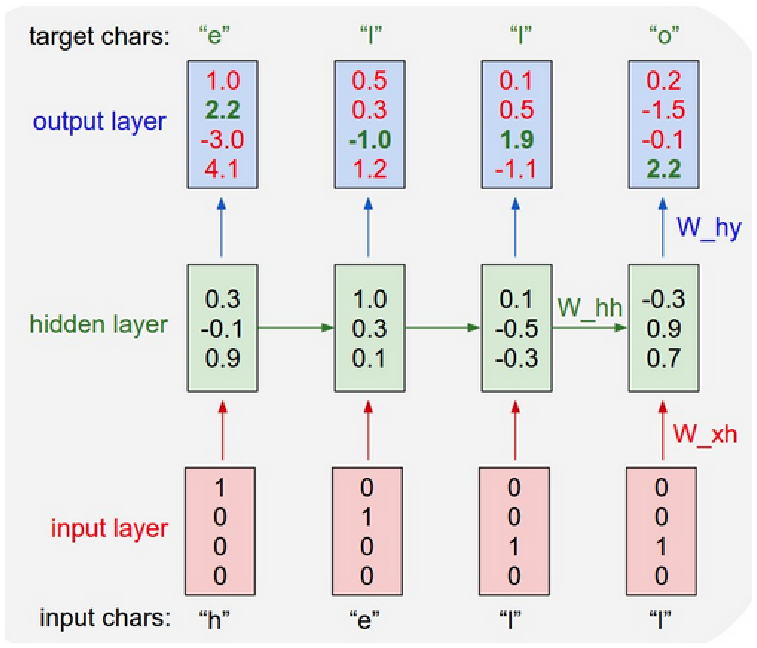


Figure 3.10: Example of RNN/LSTM

So we would like the network to provide high green numbers (the probability corresponding to the right character to follow) and low red numbers, through training. Of notice are the 3rd and 4th inputs which are the same “l”, but the network should have past inputs in consideration and differentiate one from the other, providing different outputs (in the word “hello”, the first “l” is followed by another “l” but the second “l” is followed by “o”).

Another numerical example, more detailed, can be found in "Backpropogating an LSTM:

A Numerical Example" [15].

3.2 Data

The data used in this paper are the following:

- Daily prices of the underlying asset, the S&P 500 [23]. These prices were used both in raw form and to calculate an annualized volatility of the past 20 days' prices. Ke and Yang (2019) [25] did not annualize the volatility, which I think is a mistake. 3.879 data points were obtained, according to the available options' data (see below).
- "Risk-free" (RF) interest rates which are the treasury rates from the United States of America (USA) [11]. I used only 3-month interest rates due to computational power limitations. There were some dates for which there was no RF data so I used the data from adjacent days. Also there were a few some options with dates that didn't exist in the [11] database, so we removed them. [25] use the interest rate closest to the time to maturity of the option, which I also wanted to replicate but couldn't due to limitations in computational power.
- SPX and SPXW options (on the S&P 500, differing on which day of the week they expire) data from a CBOE (a proprietary, paid source), from 02/01/2004 until 30/04/2019. In total, we have 13.575.866 observations of the:
  - bid and ask price;
  - strike price;
  - type of option (put or call);
  - time to maturity (TTM) (in years). Ke and Yang (2019) used the TTM in days as an input to the BSM model, which I think is another mistake.

In some of the literature, the authors decided to remove options with a small open interest or that are very deep, in or out of the money. Since we have millions of observations in our case, I didn't do it. I left out 1% of the data points for testing purposes.

If someone wants to play with the models presented in this paper, there are some limited free options' data available:

- September 2018 [18]
- August 2019 [17]

4 Models used and their specifications

In this thesis, four option pricing models are used (in truth, there are eight models because we need two separate models, one for calls and another for put options. They were created using Python and many packages such as Tensorflow and Keras. My source code can be found on my GitHub page. The models are:

1. The standard BSM model because I needed a model to use as the standard of comparison for the ML models. As we all know, this model takes as inputs the price of the underlying asset, the strike price of the option, the time to maturity of said option, the RF rate and a measure of volatility of the underlying asset. For the latter, we used the variable described in section 3.2. The model outputs an arbitrage-free price of an option.

2. The MLP1 takes in the same inputs as the BSM model. It's hyper-parameters are 3 hidden layers, 300 neurons per hidden layer, a batch size of 4.096 and 30 epochs. I chose to use the same parameters as in [25] because I could not efficiently optimize the hyper-parameters using the limited computing power at hand after a lot of trial and error. LeakyReLU activation functions are used for all layers except the output layer, which uses the ReLU function.Like the BSM model, the MLP1 also outputs only one price, an equilibrium price between the bid and the ask. No constraints are imposed so that the model produces arbitrage-free prices so that shouldn't be expected.

3. The MLP2 model is similar to MLP1, but it outputs both a bid and ask price. Its hyper-parameters are 4 hidden layers, 400 neurons per hidden layer, a batch size of 4.096 and 50 epochs.

4. The LSTM model is composed of 4 hidden LSTM layers, each with 8 neurons, that take the last 20 underlying prices for a certain time period and estimate a measure of volatility. That volatility estimate, together with the strike price, TTM, RF rate, and the underlying price, are then fed into an MLP1 to compute an equilibrium option price. We are testing if volatility at time t depends on volatility at times $t-1$, $t-2$,

…, and $t-20$. The model's other hyper-parameters are 3 hidden layers like those in MLP1, 100 neurons in each of those, a batch size of 4.096 and 20 epochs.

A note on the number of hidden layers for all these models: "multiple findings indicate that adding hidden layers beyond four hidden layers does not significantly improve network performance." [19].

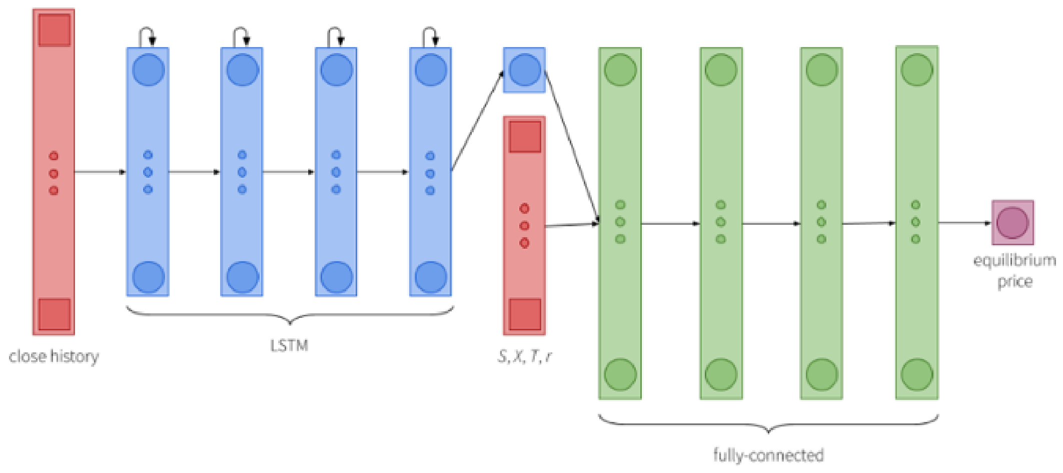


Figure 4.1: LSTM architecture [25]

5 Empirical Results

5.1 Overall results

For all models, MSE is the objective function that we want to minimize. Next follows a table that compares various metrics for the four models. Those metrics are:

- train-MSE, the MSE of the training set. In the case of the BSM, this metric was not calculated because the model is not trained like a ML one;
- MSE;
- Bias, which is the median percent error;
- Average absolute percent error (AAPE);
- Median absolute percent error (MAPE);
- PEX%, which is the percentage of observations within ±X% of the actual price.

Option type	Model	train-MSE	MSE	Bias	AAPE	MAPE	PE5	PE10	PE20
Call	BSM	–	$379,78$	$-0,43$	$75,82$	$2,25$	$60,82$	$67,34$	$73,16$
	MLP1	$41,04$	$40,55$	$0,41$	$48,96$	$1,36$	$65,14$	$70,75$	$75,49$
	MLP2	$138,5$	$138,71$	$-2,5$	$62,11$	$3,42$	$58,81$	$69,67$	$76,82$
	LSTM	$71,02$	$71,06$	$1,74$	$24,7$	2	$60,38$	$66,37$	$71,79$
Put	BSM	–	$450,98$	$96,5$	$67,36$	$96,86$	$16,92$	$21,96$	$26,96$
	MLP1	$77,66$	$78,65$	$45,32$	$67,3$	$53,17$	$16,85$	$27,89$	$38,31$
	MLP2	$25,55$	$26,18$	$7,04$	$55,83$	$21,19$	$29,85$	$39,35$	$49,18$
	LSTM	$118,25$	$118,28$	$72,18$	$60,49$	81	$13,52$	$24,44$	$33,7$

It should be noted that the metrics for the MLP2 were calculated after averaging the bid and ask price outputs. The MLP1 was the only model to beat the BSM model in all metrics.

But again, assuming a model like this is better than the BSM is premature as it does not, in theory, produce arbitrage-free prices.

Comparing these results to those of [25], I could not replicate their numbers for the most part. For example, the MSE of the MLP1s is quite different (40,55 vs. 24 for calls and 78,65 vs. 15,66 for puts). But more important are the results regarding the MLP2 and LSTM. Concerning call options, their MLP2 network almost completely outperforms their MLP1 model, but my results show the opposite. When it comes to put options, there is a similarity in that my MLP2 model outperforms MLP1, like in their results. When looking at the LSTM model, for call options, their numbers are, in general, significantly better than mine. For put options their error numbers are much lower, but unexpectedly, the PE% numbers I obtained are better. These differences might in part have to do with the different datasets used, but I suspect that the main difference is at the code level. Although they published their code as open-source, it wasn't just plug and play[1].

5.2 Results per model

5.2.1 MLP1

In figures 5.1 and 5.2, we can see both the training and the validation losses over epochs, for call and put options, respectively. Because the train-MSE and MSE are fairly similar in both cases, I conclude that there is no overfitting.

5.2.2 MLP2

The same can be said about MLP2 (and LSTM below): train-MSE is almost equal to MSE, indicating no overfitting. Figures 5.3 and 5.4 show both the training and the validation losses over epochs, for call and put options, respectively. Regarding puts, there seems to be some difficulty in convergence, so maybe more epochs would be needed.

5.2.3 LSTM

Figures 5.5 and 5.6 show both the training and validation losses over epochs, for call and put options, respectively. For put options the losses are not converging so maybe we should restrict epochs to around 8 or 9 as the best results are obtained around those epochs (the so called early stopping). "At the beginning of the learning process, the error of the validation

[1]There was a lot of work involved in understanding and replicating their models. Plus, I was new to all this technology.

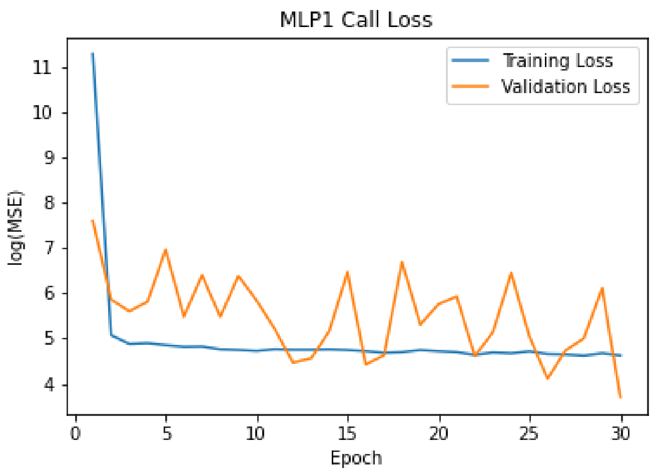


Figure 5.1

sample decreases synchronously with the error of the training sample, but later the training and validation samples start to diverge; the error decreases only in the training sample and increases in the validation sample. This phenomenon signals the overfitting of the parameters, and the process should be stopped at the end of the synchronously decreasing phase." [29].

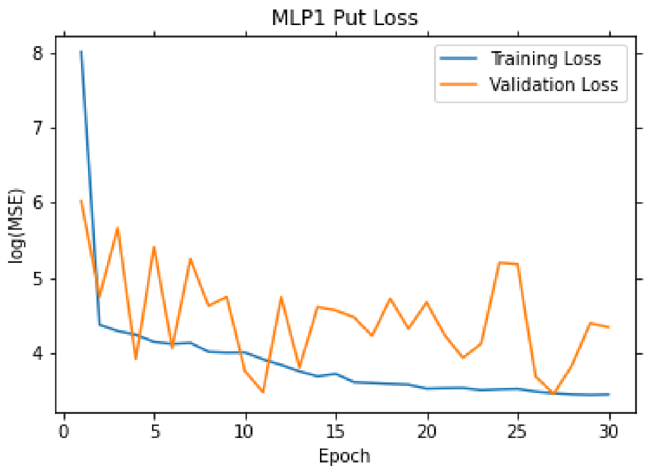


Figure 5.2

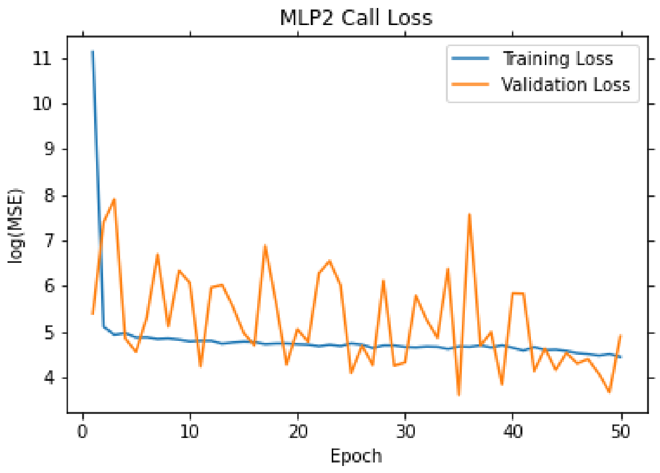


Figure 5.3

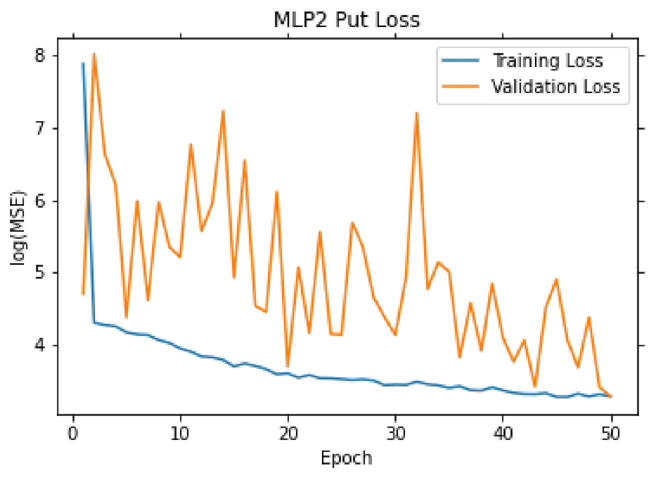


Figure 5.4

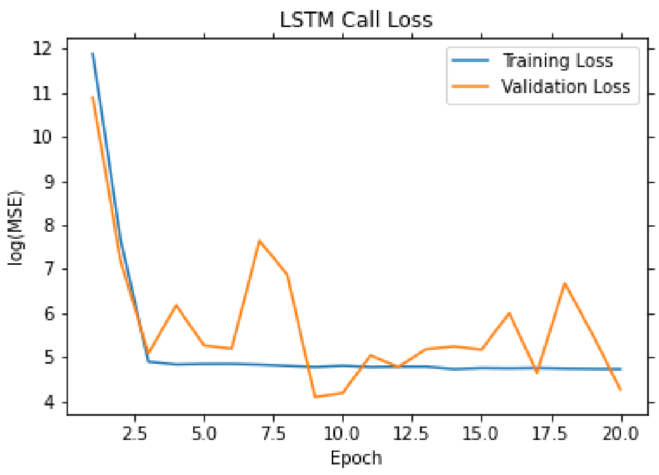


Figure 5.5

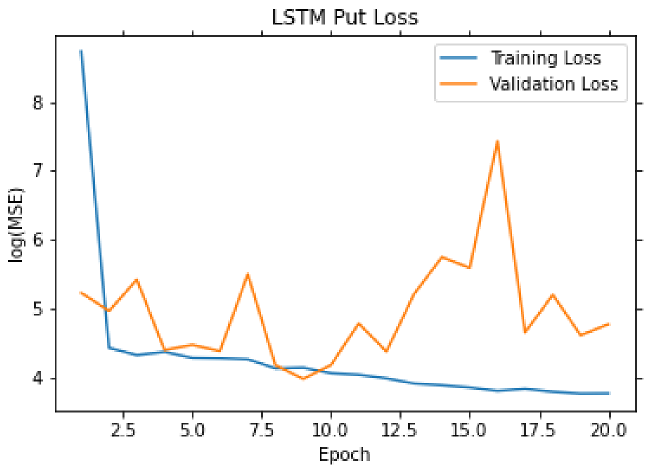

LSTM Put Loss
Training Loss
Validation Loss
log(MSE)
Epoch
4
5
6
7
8
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Figure 5.6

6 Conclusions

This thesis points to significant work to be done in option pricing via ML. The metrics for all models are better than the BSM model, but we must incorporate a way to output arbitrage-free prices. In the later stages of writing this thesis, I found the paper of Cao et al. (2020) [7]. They use ANNs to produce arbitrage-free prices, find analytical formulas for the greeks (with good hedging performance), and estimate IV. In terms of future research, it might be more productive to follow up on this paper rather than continue down this thesis's path.

In case [7] is critically flawed or can't be replicated, for example, we can continue to pursue the path taken in my thesis. In fact I tried, but without success, to create a 3rd MLP model that produced arbitrage-free prices based on the work of Andrey Itkin (2019) [22]. Maybe more seasoned researchers can do it.

Other paths one might take in future research are, for example:

- the study of ANNs for implied volatility estimation and hedging (the latter one requiring the calculation of the greeks);
- as mentioned in [25], "Additionally, we could do a deeper error analysis and examine pricing bias to see if our models perform better or worse given particular features (near the money, time until expiry, etc.)";
- work on hyper-parameter optimization, maybe using as guidance of sections 3.2 and 3.3 of [26] or section 2.4 of [16];

Bibliography

[1] Layer weight initializers. `https://keras.io/api/layers/initializers`.

[2] What is the role of the bias in neural networks? `https://stackoverflow.com/questions/2480650/what-is-the-role-of-the-bias-in-neural-networks`.

[3] C. Almeida. Previsão de séries temporais financeiras: Uma abordagem com long short-term memory deep neural networks. 2019 - Master's Thesis, Faculdade de Ciências da Universidade de Lisboa and Instituto Superior de Ciências do Trabalho e da Empresa.

[4] I. Ayati. Machine learning, investor sentiment and stock market portfolios. 2019 - Master's Thesis, CATÓLICA-LISBON and BI Norwegian Business School.

[5] D. Bloch. Option pricing with machine learning. 2019.

[6] J. Brownlee. A gentle introduction to dropout for regularizing deep neural networks. `https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks`.

[7] Y. Cao, X. Liu, and J. Zhai. Option valuation under no-arbitrage constraints with neural networks. 2020, working paper.

[8] X. Chen. Option pricing and deep learning: Based on long short-term memory. 2019 - course work.

[9] B. Christensen and M. Nielsen. The Effect of Long Memory in Volatility on Stock Market Fluctuations. *The Review of Economics and Statistics*, 89(4):684–700, November 2007.

[10] Contributors to GeeksforGeeks. Lstm – derivation of back propagation through time. `https://www.geeksforgeeks.org/lstm-derivation-of-back-propagation-through-time`.

[11] Department of the Treasury. Daily treasury par yield curve rates. `https://www.treasury.gov/resource-center/data-chart-center/interest-rates/pages/TextView.aspx?data=yieldYear&year=2019`.

[12] T. Dias. Volatility forecasting : the role of financial news in forecasting stock market volatility. 2015 - Master's Thesis, ESCP Europe and CATÓLICA-LISBON.

[13] Y. Elouerkhaoui. Derivatives pricing with a deep learning approach. *QuantMinds International Conference*, 2019.

[14] R. Gaspar, S. Lopes, and B. Sequeira. Neural network pricing of american put options. *REM Working paper nº 0122 – 2020*, 2020, Instituto Superior de Economia e Gestão.

[15] A. Gomez. Backpropogating an lstm: A numerical example. `https://medium.com/@aidangomez/let-s-do-this-f9b699de31d9`.

[16] T. Hahn. Option pricing using artificial neural networks : an australian perspective. 2013.

[17] Historical Option Data contributors. August 2019 s&p500 option data. `https://www.historicaloptiondata.com/files/Sample_L2_2019_August.zip`.

[18] Historical Option Data contributors. September 2018 s&p500 option data. `https://www.historicaloptiondata.com/files/Batch_PRO_Sample_PHC.zip`.

[19] Y. Horvath, A. Muguruza, and M. Tomas. Deep learning volatility: A deep neural network perspective on pricing and calibration in (rough) volatility models. 2019 - working paper.

[20] J. Huh. Pricing options with exponential lévy neural network. *Expert Systems With Applications*, 127:128–140, 2019.

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

[22] A. Itkin. Deep learning calibration of op-tion pricing models: some pitfalls and solutions. 2019.

[23] Wall Street Journal. S&p 500 index historical prices. `https://www.wsj.com/market-data/quotes/index/SPX/historical-prices`.

[24] A. Karpathy. The unreasonable effectiveness of recurrent neural networks. `https://karpathy.github.io/2015/05/21/rnn-effectiveness`.

[25] A. Ke and A. Yang. Option pricing with deep learning. `https://cs230.stanford.edu/projects_fall_2019/reports/26260984.pdf`, 2019 - Course Report.

[26] S. Liu, C. Oosterlee, and S. Bohte. Pricing options and computing implied volatilities using neural networks. *Risks*, 7(1), 2019.

[27] M. Malliaris and L. Salchenberger. A neural network model for estimating option prices. *Journal of Applied Intelligencee*, 3:193–206, 1993.

[28] I. Marques. Machine learning in finance : stock market prediction. 2018 - Master's Thesis, Universidade de Lisboa. Instituto Superior de Economia e Gestão.

[29] B. Mefozi and K. Szabo. Beyond black-scholes: A new option for options pricing. 2019, presented at WorldQuant Perspectives.

[30] S. Mitra. An option pricing model that combines neural network approach and black scholes formula. 2012.

[31] P. Pedamkar. Introduction to deep learning networks. `https://www.educba.com/deep-learning-networks`.

[32] D. Rodrigues. Os efeitos do algorithmic trading na previsibilidade dos mercados financeiros. 2019 - Master's Thesis, Faculdade de Ciências da Universidade de Lisboa and Instituto Superior de Ciências do Trabalho e da Empresa.

[33] J. Ruf and W. Wang. Neural networks for option pricing and hedging: a literature review. 2020.

[34] Scan2CAD. The hidden layer. `https://www.scan2cad.com/user-manual/the-hidden-layer`.

[35] B. Sequeira. American put option pricing : a comparison between neural networks and least-square monte carlo method. 2019 - Master's Thesis, Universidade de Lisboa. Instituto Superior de Economia e Gestão.

[36] D. Stafford. Machine learning in option pricing. 2018 - Master's Thesis, Oulun Yliopisto, University of Oulu. Oulu Business School.

[37] R. Staudemeyer and E. Morris. Understanding lstm – a tutorial into long short-term memory recurrent neural networks. 2019.

[38] H. Trønnes. Pricing options with an artificial neural network: A reinforcement learning approach. 2018.

[39] J. Uv. Pricing european options with lévy market models and deep learning. 2019 - Master's Thesis, Norwegian University of Science and Technology. Faculty of Information Technology and Electrical Engineering.

[40] S. Varma and S. Das. Deep learning. `https://srdas.github.io/DLBook`.

[41] S. Vaz. How efficient is the portuguese stock market? 2012 - Master's Thesis, Universidade de Lisboa. Instituto Superior de Economia e Gestão.

[42] R. Vijay. Understanding neural networks. `https://towardsdatascience.com/understanding-neural-networks-677a1b01e371`.

[43] Wikipedia contributors. Activation function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Activation_function#Table_of_activation_functions`.

[44] Wikipedia contributors. Artificial neural network — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Artificial_neural_network`.

[45] Wikipedia contributors. Hadamard product (matrices) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Hadamard_product_(matrices)`.

[46] Wikipedia contributors. Implied volatility — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Implied_volatility`.

[47] Wikipedia contributors. Logistic function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Logistic_function`.

[48] Wikipedia contributors. Norm (mathematics). `https://en.wikipedia.org/wiki/Norm_(mathematics)#p-norm`.

[49] Wolfram Research. Introduction to the hyperbolic tangent function. `https://functions.wolfram.com/ElementaryFunctions/Tanh/introductions/Tanh/ShowAll.html`.