Journal of Mathematics in Industry
a SpringerOpen Journal

RESEARCH Open Access

How deep is your model? Network topology selection from a model validation perspective

Nikolai Nowaczyk[1], Jörg Kienitz[2,3,4*], Sarp Kaya Acar[5] and Qian Liang[6]

*Correspondence:
jkienitz@uni-wuppertal.de
[2]Fachbereich Mathematik und Naturwissenschaften, Bergische Universität Wuppertal, Wuppertal, Germany
[3]The African Institute for Financial Markets and Risk Management (AIFMRM), University of Cape Town, Cape Town, South Africa
Full list of author information is available at the end of the article

Abstract
Deep learning is a powerful tool, which is becoming increasingly popular in financial modeling. However, model validation requirements such as SR 11-7 pose a significant obstacle to the deployment of neural networks in a bank's production system. Their typically high number of (hyper-)parameters poses a particular challenge to model selection, benchmarking and documentation. We present a simple grid based method together with an open source implementation and show how this pragmatically satisfies model validation requirements. We illustrate the method by learning the option pricing formula in the Black–Scholes and the Heston model.

Keywords: Neural networks; Model validation; SR 11-7; Derivatives; Risk management; Pricing

Recent advances in machine learning have shown how neural networks can learn to solve classical problems in quantitative finance such as pricing (see [19]), calibration, (see [3, 12, 16]) and hedging (see [5]). A key advantage of using this technique is that once the network has been trained, it performs the computations much faster than classical approaches such as Monte Carlo simulation.

While the machine learning theory and practical advances as well as the quantitative finance foundations behind this approach are well recognized, these models are not yet widely used in production. One of the key obstacles to the deployment of any model in a bank is that it has to pass a thorough *model validation* first and – depending on its use case – requires *regulatory approval*, which is not straightforward to obtain.

A key step in model validation is the *model selection* process. The model development function of a bank needs to explain and justify to the model validation function why a certain model has been selected ("conceptual soundness"). Traditionally, this is often formulated by choosing a champion model and one or various challenger models. However, those principles and practices have been formulated with models in mind like Heston vs. Black–Scholes.

For models with parameters, the model selection process includes the choice of these parameters. Consequently, for artificial neural networks (ANN), which typically have many parameters and hyperparameters, this question is a bit more complex than for classical quantitative finance models. First of all, a type of network topology has to be chosen. For example, can the problem be learned best with a multilayer perceptron (MLP) or a long-term-short-term memory network (LSTM)?[1] After a type of topology has been selected, for example MLP, the precise shape of that topology still needs to be determined and justified, e.g. the number of layers and neurons needs to be specified as hyperparameters as well as the activation functions. In a last step, the networks weights have to be chosen. This last step can be performed automatically by stochastic gradient descent methods such as Adam, see [18], but those often require hyperparmeters as well, e.g. the learning rate.

ANNs are already widely used in many areas and the problem of choosing the type of network topology and the number of layers and units occurs everywhere and not just in quantitative finance.

Typical approaches to this problem are:

1. Arbitrary choice: The model developer simply makes a choice and plays around with it until the results are satisfactory. Alternative choices are not documented or systematically evaluated.
2. Automatic Machine Learning: There are attempts to automate the process of finding supervised machine learning solutions for a given problem and data set using unsupervised machine learning techniques – those are dubbed *AutoML*.
3. Functional Analysis: From a mathematical perspective, a neural network is a method to approximate a non-linear function. All possible choices of network topologies and weights constitute a space of approximation functions and thus existing theory from approximation theory can be utilized.

The first method is of course straight-forward and quite often successfully used in practice. However, it is not always successful and it is certainly not compliant with any model governance framework. For example, the SR 11-7 guidelines clearly state that "a sound development process will produce documented evidence in support of all model choices", see [2, Sect. V.1]. This method clearly violates that requirement and thus cannot be used for financial models in production.

While the idea of the second method, i.e. using unsupervised learning to automatically choose the topology for a supervised learning problem, is very appealing, there are two problems to consider: First, automatic machine learning is unsurprisingly a much more difficult problem than just one supervised learning problem and it is not always feasible to apply it in practice. The second issue is that from a model validation perspective, one tries to shed light into a blackbox with another blackbox. AutoML, in particular if used in a proprietary version, can only be used to validate a bank's machine learning solution to a specific problem after the AutoML engine that was used has itself successfully completed a model validation process. That might be possible, but requires a lot of additional resources – potentially much more than to justify the neural network in question directly. Also, it remains to be seen if a regulator would ever sign off such a blank check. One should however remark that open source initiatives such as [17] are currently improving transparency and efficacy of such automatic machine learning solutions.

The third method, has the advantage that a lot of literature and research already exists in the area of function approximation, see for example [9, 22, 24]. Specifically for neural networks, there is the famous universal approximation theorem, see [8, 11, 15]. This literature is very helpful in providing sound methodological justifications and theoretical

[1] See Sect. 1 for detailed definitions.

background. However, in practice, this will often not be enough to make the determination of the concrete (hyper-)parameters of the network straightforward for the problem at hand.

We therefore propose an intermediary solution that improves the first method by some of the techniques used in the second, in particular the use of grids. We formulate this approach in a language that takes the more classical perspective of *degrees of freedom*, which is very common in model validation. By systematically training neural networks on a grid of comparable parameters, we obtain a framework that can be used in practice to simultaneously satisfy multiple model validation requirements.

The rest of this paper is organized as follows: After a review of some popular neural network topologies in Sect. 1, in particular the MLPs and the LSTMs, we discuss the model selection problem in Sect. 2 and present a simple grid based method to select a neural network topology for a quantitative finance problem. We discuss in detail how this method addresses SR 11-7 model validation requirements in Sect. 3. After quickly discussing the implementation in Sect. 4, we apply the method in Sect. 5 to pricing in the context of Black–Scholes model and a Heston model. The conclusions are summarized in Sect. 6.

1 Artificial neural networks (ANN)

In this section, we establish the notation for two of the most common types[2] of artificial neural networks: the multilayer perceptron (MLP) and the long-term-short-term-memory network (LSTM). It should be highlighted that the notation, the mathematical formalization and also the implementation slightly varies throughout the literature, see for example [10, 20, 21]. We have chosen an approach here that is compatible with `keras` and `tensorflow`, see [1, 6], as those frameworks are very common.

1.1 Multilayer perceptron (MLP)

The most common and most basic form of neural networks are multilayer perceptrons.

Definition 1.1 (Multilayer perceptron) A *multilayer perceptron* MLP is a tuple MLP = $(A_l, b_l, \sigma_l)_{1 \le l \le n_L}$ defined by

- a number n_i of *inputs*,
- a number n_o of *outputs*,
- a number n_L of *layers* and
- for each layer $1 \le l \le n_L$
  - a number n_l of *neurons* (or *units*),
  - a matrix $A_l = (A_{l;ij}) \in \mathbb{R}^{n_{l-1} \times n_l}$ and a vector $b_l = (b_{l;i}) \in \mathbb{R}^{n_l}$ (called *bias*) of *weights* such that $n_0 = n_i$, $n_{n_L} = n_o$ and
  - an *activation function* $\sigma_l : \mathbb{R} \to \mathbb{R}$.

For any $1 \le l \le n_L$, the tuple (A_l, b_l, σ_l) is called a *layer*. For $l = n_{n_L}$, the layer is called *output layer* and for $1 \le l < n_L$, the layer is called *hidden layer*.

Neural networks can be visualized as in Fig. 1: This shows a network $(A_l, b_l, \sigma_l)_{1 \le l \le n_L}$ with a total of $n_L = 4$ layers, i.e. 3 layers are hidden. Notice that the input layer is just a visualization of the input and is not part of the actual network topology.

Computing the output from the input is codified in the *feed forward*.

[2] We illustrate our method on MLPs and LSTMs, but it can in general be applied to other topologies as well like CNNs.

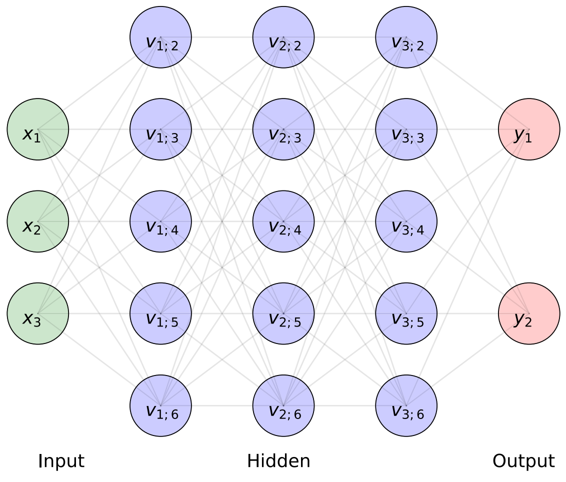


Figure 1 Multilayer Perceptron

Definition 1.2 (Feed forward) Let MLP $= (A_l, b_l, \sigma_l)_{1\le l\le n_L}$ be a multilayer perceptron. Then for each $1 \le l \le n_L$, we define a function

$$F_l : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}, \qquad v \mapsto \sigma_l\big(v^T A_l + b_l\big),$$

where we employ the convention that σ_l is applied in every component. The composition

$$F : \mathbb{R}^{n_i} \to \mathbb{R}^{n_o}, \qquad F := F_L \circ \cdots \circ F_2 \circ F_1$$

is called the *feed forward* of MLP. Any set of inputs $x \in \mathbb{R}^{n_i}$ is called an *input layer*.

The links in Fig. 1 between the nodes visualize the feedforward and the dependence of each output on each input.

We assume here that the activation functions are chosen as the standard *sigmoid*, $\sigma_l(x) := (1 + e^{-x})^{-1}$, for all but the last layer,[3] where we chose the *linear activation* $\sigma_L(x) = x$.

The feed forward $F = F_\Theta$ depends on all the parameters $\Theta = (A_l, b_l)_{1\le l\le n_L}$. In general, the function F_Θ will be unrelated to the problem if the weights Θ are not chosen carefully. This is performed by *training* the neural network with a *training set* $(x_i, y_i)_{1\le i\le N}$. More precisely, for a given cost function, say least squares, this amounts to solving the optimization problem

$$\Theta^* := \operatorname*{argmin}_{\Theta} \sum_{i=1}^{N} \big\| F_\Theta(x_i) - y_i \big\|^2.$$

In practice, this optimization is performed by stochastic gradient descent methods such as *Adam*, see [18], and a clever computation of the gradient, called *backpropagation*, see [23].

[3]The sigmoid function takes values in $[0, 1]$. Thus, choosing a sigmoid activation in the output layer can vastly decrease the accuracy of the network, in particular if the function intended to learn takes unbounded values in $\mathbb{R}$.

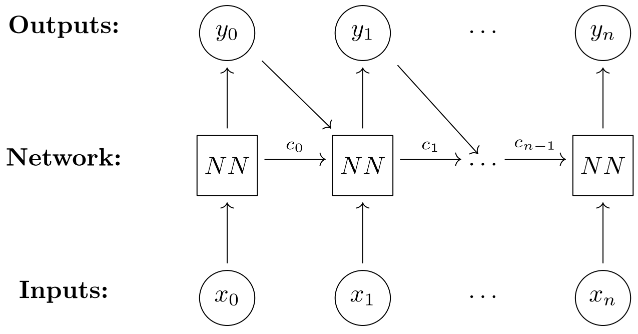


Figure 2 LSTM Concept

1.2 Long-Term-Short-Term-Memory Network (LSTM)

While MLPs work very well in many situations, there are certain applications for which that network topology is sometimes not ideal. MLPs are built to make a prediction y given one input x. But some applications have a canonical time structure and the task is to make time-dependent predictions y_t from time-dependent inputs x_t. In principle, this case can of course be covered by MLPs as well, for example by adding a time grid to the inputs and the outputs, i.e. to learn $(y_{t_1},\ldots,y_{t_n})$ from $(x_{t_1},\ldots,x_{t_n},t_1,\ldots,t_n)$. This has the advantage that it is straight-forward to implement, but the disadvantage that the network has to be potentially quite large. Another option is to train a network separately for each point t_i. That has the disadvantage that there is no flow of information between the networks of different points in time t_i and the prediction of this sequence of networks might suffer from inconsistencies. A key application that has driven the research in that area is Natural Language Processing (NLP), where language is seen as a sequence of words.

A known way out of these technical problems are long-term-short-term memory networks as suggested in [14]. The idea depicted in Fig. 2 is as follows: Only one neural network, i.e. with one fixed set of weights, is trained, but in addition to the input, the network also processes the *cell state*. This additional piece of information is transmitted through the network and serves as a memory of previous predictions.

Formally, an LSTM can be defined by first defining a single LSTM layer, LSTML, and the associated feedforward and then stacking multiple of those together to an LSTM.

Definition 1.3 (LSTML) A *long-term-short-term-memory neural network layer* is a tuple LSTML = LSTML(W,U,b,τ,σ) consisting of

- a number m of units and a number k of features,
- a 4-tuple W of matrices W_i, W_f, W_c, $W_o \in \mathbb{R}^{k\times m}$ called *input*, *forget*, *cell* and *output kernels*,
- a 4-tuple U of matrices $U_i, U_f, U_c, U_o \in \mathbb{R}^{m\times m}$ called *input*, *forget*, *cell* and *output recurrent kernels*,
- a 4-tuple b of vectors $b_i, b_f, b_c, b_o \in \mathbb{R}^m$ called *input*, *forget*, *cell* and *output bias*,
- two functions $\sigma,\tau:\mathbb{R}\to\mathbb{R}$ called *activation* and *recurrent activation.*

This definition needs to be understood in the context of the associated feedforward. The feedforward of an LSTML is more complex than for MLPs. Because of the time-

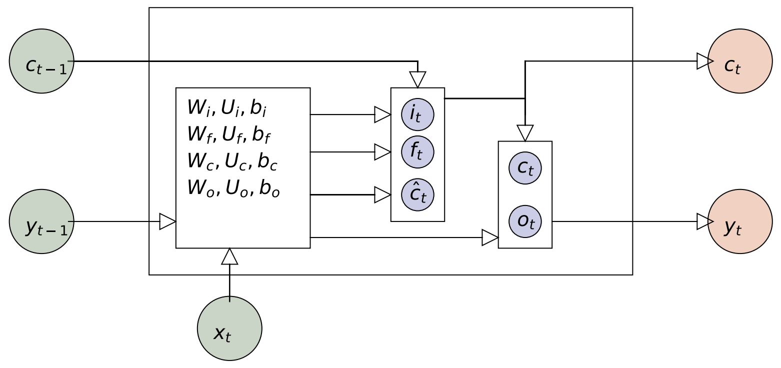


Figure 3 Long-Term-Short-Term-Memory Network Cell

dependence, it needs to keep track not only of the final output, but of all the outputs at the various points in time and in addition it needs to keep track of the cell state.

Definition 1.4 (Feedforward) Let LSTML = LSTML(W, U, b, τ, σ) be as above and let $T \in \mathbb{N}$ be a natural number. Any sequence $x = (x_1, \dots, x_T)$, $x_t \in \mathbb{R}^k$, is called an *input sequence*. The two sequences c_t and y_t, $t = 1, \dots, T$, called *cell state* and *carry state*, are recursively defined as follows:[4]

input: $i_t := \tau(x_t \bullet W_i + y_{t-1} \bullet U_i + b_i) \in \mathbb{R}^m$,
forget: $f_t := \tau(x_t \bullet W_f + y_{t-1} \bullet U_f + b_f) \in \mathbb{R}^m$,
candidate: $\tilde{c}_t := \sigma(x_t \bullet W_c + y_{t-1} \bullet U_c + b_c) \in \mathbb{R}^m$,
cell: $c_t := f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \in \mathbb{R}^m$,
output: $o_t := \tau(x_t \bullet W_o + y_{t-1} \bullet U_o + b_o) \in \mathbb{R}^m$,
carry: $y_t := o_t \tau(c_t) \in \mathbb{R}^m$.

Finally, the function

$$\begin{aligned} F_T &: \mathbb{R}^{k \times T} \to \mathbb{R}^m, \\ x &= (x_1, \dots, x_T) \mapsto (y_1, \dots, y_T) \end{aligned}$$

is called, the *feedforward of* LSTML *of length* T.

The interpretation of this is as follows (see Fig. 3): The input value i_t and forget value f_t represent how much weight is put by the network on the current input x_t and how much weight is put on forgetting the past memory. Then, a candidate cell state $\tilde{c}_t$ and a candidate output o_t are computed on the basis of only the current input x_t and the last prediction y_{t-1}. Thus, the new cell state c_t is computed as a weighted average between the candidate cell state $\tilde{c}_t$ and the previous cell state c_{t-1}, where the weights are given by the input weight i_t and the forget weight f_t. Finally, the carry y_t, i.e. the intermediate output at t, is computed by first computing a candidate output o_t, which is also based only on the current input x_t and the last prediction y_t, and then y_t is computed from o_t by weighing o_t with the new cell

[4]Here, we assume that all vectors are row vectors, all sequences are initialized with zero, $\bullet$ denotes the usual matrix-vector multiplication, $\odot$ denotes the element-wise multiplication of vectors (Hadamard product) and the application of a function $\mathbb{R} \to \mathbb{R}$, e.g. σ and τ, to a vector is performed element-wise.

state c_t. It should be noted that depending on the application, one might either consider the value y_T as the feedforward of the network or the vector $(y_1, \dots, y_T)$.

Practical applications rarely comprise of a single LSTML for various reasons. First, as we can see in Definition 1.4, each step of the computation, i.e. the input, forget, output and candidate cell state are equivalent to a feedforward of an MLP with just a single layer. That means that a single LSTML cannot capture arbitrary non-linearities with these computations. Also, notice that if a network comprises of just an LSTML, then the number k of features is forced to be $k = n_i$, i.e. the number of inputs and the number m is forced to be $m = n_o$, i.e. the number of outputs. That means that no parameters can be changed to adapt the network to the problem. The solution to both of these problems is to chain multiple LSMTLs in sequence, where the number m of units can be chosen at will, followed by a single MLP layer to ensure that the last output is of the same shape as n_o.

Definition 1.5 (LSTM) A *long-term-short-term memory network (LSTM)* with L layers is defined by a sequence $\mathrm{LSTM} = (\mathrm{LSTML}_1, \dots, \mathrm{LSTML}_{L-1}, \mathrm{MLP})$ of $L - 1$ LSTMLs with number of units m_l and number of features k_l such that $k_1 = n_i$, and a single MLP with input dimension k_{L-1} and output dimension n_o.

2 Network topology selection

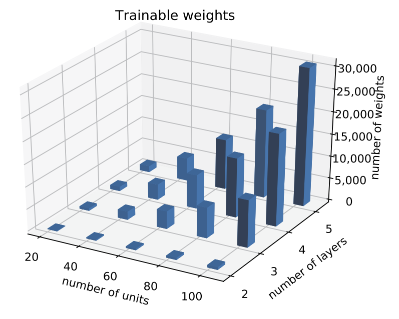
In this section we introduce a method to select a network topology that is consistent with established model validation practices.

Assume we want to train an MLP on a financial problem such as pricing. The key model parameters to choose are the number of layers n_L and the number of units n_u in each layer (we are assuming that we want to choose the same number in each layer). A very simple way to obtain a documented choice for that is to not pick one arbitrary parameter vector (n_u, n_L), but to specify a grid of those parameters and built an MLP for each of them, see Fig. 4 for an example. The idea is to then train all of those models with the same data, analyze their learning curves and the performance of the trained model and then the select the least complex model such that its performance is within thresholds acceptable by the business case. As discussed in Sect. 3, this simple procedure already satisfies many key requirements of model validation, but there is one catch.

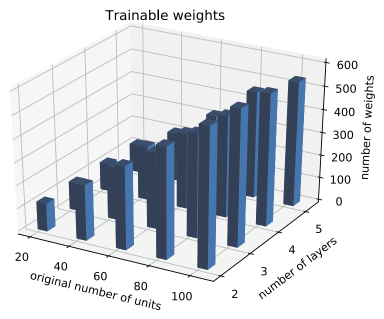
A key question, which is not only interesting from a model validation perspective is: For the given learning problem, is it better to increase model performance by increasing the number of layers or by increasing the number of units? If the grid $\mathcal{G} = (g_{ij})$ of parameters $g_{ij} = (n_{u_i}, n_{L_j})$ is a cartesian product of a vector of possible number units (n_{u_i}) and a vector

$$\begin{pmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \\ g_{51} & g_{52} & g_{53} & g_{54} \end{pmatrix} \mapsto \begin{pmatrix} \mathrm{NN}_{11} & \mathrm{NN}_{12} & \mathrm{NN}_{13} & \mathrm{NN}_{14} \\ \mathrm{NN}_{21} & \mathrm{NN}_{22} & \mathrm{NN}_{23} & \mathrm{NN}_{24} \\ \mathrm{NN}_{31} & \mathrm{NN}_{32} & \mathrm{NN}_{33} & \mathrm{NN}_{34} \\ \mathrm{NN}_{41} & \mathrm{NN}_{42} & \mathrm{NN}_{43} & \mathrm{NN}_{44} \\ \mathrm{NN}_{51} & \mathrm{NN}_{52} & \mathrm{NN}_{53} & \mathrm{NN}_{54} \end{pmatrix}$$

Figure 4 Mapping model parameters to models NN



(a) linear x quadratic



(b) linear x linear

Figure 5 Number of trainable weights

of possible number of layers (n_{L_j}), this amounts to the question of whether one should go down a row or right along a column in the grid, see Fig. 4.

In order to obtain a meaningful answer to this question, one has to consider the *degrees of freedom* of the model. For a neural network, these are exactly the number n_w of trainable weights. It is good model selection practice to increase the degrees of freedom of a model slowly from below to obtain as many as needed to be accurate, but no more than necessary to avoid overfitting. Applied to MLPs, this means that we actually want a grid where along the one (and only one) dimension, we can increase the degrees of freedom and along the other we can change the topology of the network keeping the degrees of freedoms fixed. This is not achieved by a grid $\mathcal{G}$ that is simply a cartesian product of number of units and number of layers, because both increase the degrees of freedom and they do so in a very different way, see Fig. 5(a). An easy way to fix this is to build the grid by first fixing the smallest candidate for the number of layers, say $n_{L_1} = 2$, and then fill the first column of the grid with $g_{i1} = (n_{u_i}, n_{L_1})$, where n_{u_i} is the vector of candidate number of units. In a second step, we then compute the resulting degrees of freedom n_w for each row in the first column. In a third step, we fill the other columns by gradually increasing the number of layers, but simultaneously reducing the number of units to keep the degrees of freedom in each row constant. In this way, the row axis increases the degrees of freedom (via increasing the number of units) and along the column axis we obtain models with the same degree of freedom, but different network topologies. That is exactly what we want. To carry out this program in detail, we need to calculate the degrees of freedom, i.e. the number of trainable weights of the network, see below, and formalize the above in an algorithm, see Algorithm 2.2.

For an MLP NN $= (A_l, b_l, \sigma_l)_{1\leq l\leq n_L}$, calculating the number of trainable weights amounts to the following: Given that any layer has a matrix $A \in \mathbb{R}^{n_{l-1}\times n_l}$ and a bias $b \in \mathbb{R}^{n_l}$, this yields $n_{l-1}n_l + n_l = n_l(n_{l-1} + 1)$ trainable weights per layer. Taking into account that $n_0 = n_i$ and $n_L = n_o$, i.e. the dimensions of the input and the output layers are fixed, the total number n_w of trainable weights is given by

$$n_w = \begin{cases} n_o(n_i + 1), & n_L = 1, \\ n_1(n_i + n_o + 1) + n_o, & n_L = 2, \end{cases} \tag{1}$$

and for $n_L \geq 3$

$$n_w = n_1(n_i + 1) + n_o(n_L + 1) + \sum_{l=2}^{n_L-1} n_l(n_{l-1} + 1). \tag{2}$$

While in theory it is perfectly possible to choose a different number of units for every hidden layer, in practice one often uses the following.

Assumption 2.1 The number of units n_u is the same in each layer.

In that case Eq. (2) simplifies to

$$\begin{aligned} n_w &= n_u(n_i + 1) + n_o(n_u + 1) + (n_L - 2)n_u(n_u + 1) \\ &= n_u^2(n_L - 2) + (n_i + n_o + n_L - 1)n_u + n_o, \end{aligned} \tag{3}$$

which requires two choices, namely n_L, the number of layers, and n_u, the number of units per layer. In Fig. 5(a) we plot this function for an example. We see that – in accordance with Eq. (3) – the number of trainable weights $n_w = n_w(n_L, n_u)$ depends linearly in the number n_L of layers, but quadratically in the number n_u of units. The crucial exception from this is the case of $n_L = 2$ as evident from Eq. (1), where the number of units only enters linearly. This is why we see the huge jump in trainable weights when passing from $n_L = 2$ to $n_L = 3$ layers.

Computing a reduced number n_u of units after increasing the number n_L of layers keeping the total number of weights constant, therefore amounts to rewriting the quadratic equation Eq. (3) as

$$n_u^2 + \underbrace{\frac{n_i + n_o + n_L - 1}{n_L - 2}}_{=:p} n_u + \underbrace{\frac{n_o - n_w}{n_L - 2}}_{=:q} = 0, \tag{4}$$

which is easily solved by setting

$$n_u = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}. \tag{5}$$

Of course in practice one has to take the floor $\lfloor n_u \rfloor$ (or a rounding) to enforce an integer number of units. Using this we can keep the number of weights approximately constant when increasing the number of layers. This is illustrated in an example in Fig. 5(b). Keeping the number of weights constant when increasing the layers allows us to control the degrees of freedom with a single variable resulting in a more meaningful comparison between various network topologies.

This leaves us with the following method of determining a good network topology for any given problem.

Algorithm 2.2 (Network topology selection)

Input:

(i) An artificial neural network NN.

(ii) A labeled data set (x, y) together with a train/test split (e.g. 80%, 20%).
(iii) A range of number of layers $\mathcal{N}_L = (L_1, \ldots, L_r)$.
(iv) A range of number of original units $\mathcal{N}_u = (u_1, \ldots, u_s)$.
(v) A bias threshold t_b and a variance threshold t_v together with metrics for both (e.g. MSE).
(vi) A number $e_{\max}$ of maximal epochs.

Steps:

(i) Create a grid $\mathcal{G} = (g_{ij})_{1\leq i\leq s, 1\leq j\leq r}$ of tuples g_{ij} as follows: For the first number n_{L_1} of layers initialize $g_{i1} := (u_i, n_{L_1})$, i.e. use the original number of units and L_1 layers. For any $j > 1$, set $g_{ij} := (u'_i, n_{L_j})$ where u'_i is the solution of Eq. (5) with n_w set as the same as resulting from g_{i1}. This results in a grid where the degrees of freedom increase by going down a row, but keep constant when going right a column, c.f. Figure 4.
(ii) For each network NN resulting from the parameters in the grid $\mathcal{G}$, train the network with (x, y) until the bias and variance is below the thresholds t_b and t_v (or the maximum number of epochs $e_{\max}$ is reached). This results in a grid of trained models, see Sect. 5.2 for an example.
(iii) Cross out all networks on the grid, for which the bias and the variance are not below the given thresholds.[5]
(iv) Amongst the remaining, find the smallest number n_L of layers, for which there exist a number of units n_u such that the model (n_u, n_L) has not been crossed out. Amongst those, choose the one with the smallest number n_u of units.

Output: A number (n_u, n_L) of units and layers for the network NN such that the bias and the variance of the network on (x, y) are within the threshold and the numbers (n_u, n_L) are optimal within the given range.[6]

Optionally, one can create a second grid to compare how the MLP performs against the LSTM. One only has to determine the number of weights in an LSTM as well and choose the number of units in the original LSTMs such that the resulting degrees of freedom are approximately the same as in the reference MLPs. It follows from Definition 1.3 that the number of weights in a single LSTM layer is given by

$$4m^2 + 4(k+1)m$$

and thus by Definition 1.5, we obtain that the total number n_w of weights in the LSTM is given by

$$n_w = 4(2n_L - 3)n_u^2 + (4n_i + n_o + 4n_L - 4)n_u + n_o,$$

where n_L is the number of layers, n_u is the number of units in each layer and n_i and n_o are the number of inputs and outputs.

[5]In case all models are crossed out, the number of units or layers or the number of training samples or the number of epochs needs to be increased to yield a meaningful result.

[6]They are not optimal in a mathematical sense as a grid obviously only tests this on a finite number of candidate models. However, if the grid is fine enough, this is sufficient for practical model validation purposes.

3 Fulfillment of model validation requirements

The network topology selection as discussed in Algorithm 2.2 serves to fulfill various model validation requirements as mandated by SR 11-7, see [2]. First of all we note that just because neural networks are not classical Monte Carlo simulations, this does not mean that they are in principle ineligible for production. In fact, SR 11-7 explicitly states that the "nature of testing and analysis will depend on the type of model and will be judged by different criteria depending on the context", [2, Sect. IV, p.6]. Thus, it is reasonable and compliant that testing neural networks looks a bit different from testing a Monte Carlo simulation.

Our approach to select a network topology addresses multiple SR 11-7 validation requirements simultaneously:

1. Documented Choice: "A sound development process will produce documented evidence in support of all model choices. Comparison to alternative theories and approaches should be included" ([2, Sect. V.I, p.11]). The learning curves of a grid of trained models resulting from the network topology selection method, see Fig. 9 for an example, serve as documented evidence that a "comparison with alternative theories and approaches" as a "fundamental component of a sound modeling process" ([2, Sect. IV, p.6]) has been conducted. This applies to both, the number of layers and units within a network topology, and comparisons between multiple network topologies.
2. Benchmarking: "Benchmarking is the comparison of a given model's inputs and outputs to estimates from alternative internal or external data or models. It can be incorporated in model development as well as in ongoing monitoring. Whatever the source, benchmark models should be rigorous and benchmark data should be accurate and complete to ensure a reasonable comparison." ([2, Sect. V.2, p.13]) Because the model selection is based on a comprehensive systematic benchmark against alternatives based on exactly the same implementation approach, input data and fitting parameters, this requirement is satisfied automatically.
3. Outcome Analysis: "The third core element of the validation process is outcomes analysis, a comparison of model outputs to corresponding actual outcomes. The precise nature of the comparison depends on the objectives of a model, and might include an assessment of the accuracy of estimates or forecasts, an evaluation of rank-ordering ability, or other appropriate tests." ([2, Sect. V.3, p.13]) Because a model in the grid is only chosen, if its bias is below the specified threshold in a suitable metric, this ensures that the chosen model is accurate.
4. Prevention of Overfitting: "Analysis of in-sample fit and of model performance in holdout samples (data set aside and not used to estimate the original model) are important parts of model development..." ([2, Sect. V.3, p.14]) Because the learning curves include the variance, the requirement to consider both is satisfied. In fact, because the total number of degrees of freedom in the model is increased from below, this method automatically makes the model much less prone to overfitting.

One should highlight that the last point is particularly delicate for financial applications. Models in the financial domain that have hundreds or even thousands of parameters are typically met with great scepticism as avoiding overfitting and instabilities in those models is no easy task. Because the machine learning community routinely deals with models that have many parameters, diagnostic methodological frameworks, a strong culture to cross-validate and standardized open source solutions have already made those models tractable.

We conclude by stressing that the proposed method to network topology selection is only one aspect of model selection, which in turn is only one aspect of model validation. Performing the suggested method of network topology selection does not exonerate the user from performing the full set of validation tasks mandated by SR11-7.

4 Technical implementation

An advantage of the network topology selection method (Algorithm 2.2) is that in theory, the implementation is straightforward. Any framework that can run one neural network can be used to run a grid of neural networks by using a loop. In practice, however, this results in various IT tasks such as managing the IDs of the models, loading and saving them, ensuring that their training parameters are consistent etc. The numerical simulation in Sect. 5 has been performed using the popular `keras` models, see [6]. We have isolated the part of the code that wraps the models into `keras_grid`, an open source module, which conveniently solves these problems.[7]

5 Application to quant finance models

In this section we apply the network topology selection method (Algorithm 2.2) to the problem of learning the option pricing function of the Black–Scholes model and the Heston model. In both cases we compare the topologies resulting from the MLP and the LSTM. We find that even though the prices $C(T,K)$ of call options clearly have a time dependence T, the MLP is actually much better suited to learn them than the LSTM. This is plausible as the problem of managing complex long-term-short-term memory does not really occur for Markovian paths generated by the Black–Scholes or Heston model. The results are described below and can also be explored interactively in a jupyter notebook.[8]

5.1 Black–Scholes & Heston model

The Black–Scholes model, see [4], assumes that the stock price S_t is a stochastic process on a probability space $(\Omega, \mathcal{F}, \mathbb{Q})$ (where we think of $\mathbb{Q}$ as the risk-neutral measure) satisfying

$$dS_t = rS_t\,dt + \sigma S_t\,dW_t, \tag{6}$$

where $r \in \mathbb{R}$ is a fixed *risk-free rate*, $\sigma > 0$ is the *volatility* and the process W_t is a Brownian motion. We denote by $\mathbb{F} = (\mathcal{F}_t)_{t\geq 0}$ the augmented filtration generated by W_t. Under these assumptions, a European call option with expiry at T and strike K, i.e. a derivative with payoff $(S_T - K)^+$ can be priced analytically with the famous Black–Scholes formula:

$$\begin{aligned} C_t(T,K) &= \mathbb{E}\big[e^{-r(T-t)}(S_T - K)^+ \mid \mathcal{F}_t\big] \\ &= S_t\Phi(d_1) - Ke^{-r(T-t)}\Phi(d_2), \end{aligned} \tag{7}$$

[7] See https://github.com/niknow/keras-grid.

[8] See https://github.com/niknow/machine-learning-examples/tree/master/network_topology_selection.

where Φ denotes the cdf of the standard normal distribution and

$$d_1 := \frac{1}{\sqrt{T-t}}\left(\log\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(t-T)\right),$$
$$d_2 := d_1 - \sigma\sqrt{T-t}.$$

The Black–Scholes model rests on the assumption that the volatility is constant, which is arguably not realistic. The Heston model, see [13], belongs to the class of stochastic volatility models, which assume a stochastic dynamic not just for the stock price, but also for the volatility. It is defined by

$$dS_t = rS_t\,dt + \sqrt{\nu_t}S_t\,dW_t^S, \tag{8}$$
$$d\nu_t = \kappa(\theta - \nu_t)\,dt + \xi\sqrt{\nu_t}\,dW_t^\nu, \tag{9}$$
$$dW_t^S\,dW_t^\nu = \rho\,dt, \tag{10}$$

where $r \in \mathbb{R}$ is the risk-free rate, $\kappa \in \mathbb{R}$ is the rate at which the stochastic variance ν_t reverts to the long-term mean $\theta > 0$, $\xi > 0$ is the volatility of the volatility and $\rho \in [0,1]$ is the correlation between the Brownian motions W_t^S, W_t^ν.

The option price in a Heston model can be computed via (see [7])

$$C_0(T,K) = S_0\Pi_1 - e^{-rT}K\Pi_2, \tag{11}$$

where Π_1 and Π_2 are given as integrals over the characteristic function $\Psi = \Psi_{\ln(S_T)}$ of $\ln(S_T)$:

$$\Pi_1 = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}\left(\frac{e^{-iw\ln(K)}\Psi(w-i)}{iw\Psi(-i)}\right)dw,$$
$$\Pi_2 = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}\left(\frac{e^{-iw\ln(K)}\Psi(w)}{iw}\right)dw.$$

We learn the Black–Scholes formula Eq. (7) for $t = 0$ as well as the Heston option pricing formula Eq. (11) with a neural network.

To that end we generate a data set as follows: We first define an evenly spaced grid of 60 maturities T between 3M and 5Y. Second, for the other model parameters, we generate 10,000 samples uniformly distributed in the hypercube with the bounds specified in Fig. 6. Third, we take the cartesian product between the maturities and the samples and obtain a data set with 600,000 samples. The special treatment of the maturity as a parameter is required here to adhere to the input format of the LSTM. Notice that in a productive environment, the specification of the bounds for the traning set has to be in line with business requirements and careful input checking against these bounds has to be performed after training when predictions are made. An example of a price surface for both models is shown in Fig. 7.

5.2 Performing network topology selection

For the grid of MLPs we choose the range of number of layers as $\mathcal{N}_L = (2,3,4)$ and the original number of units for $L = 2$ layers as $\mathcal{N}_u = (64,128,256,512,1024)$. For the grid of

Parameter	Range
$K/100$	$(80\%, 120\%)$
$S_0/100$	$(80\%, 120\%)$
r	$(1\%, 2\%)$
σ	$(5\%, 30\%)$

(a) Black-Scholes

Parameter	Range
$K/100$	$(80\%, 120\%)$
$S_0/100$	$(80\%, 120\%)$
r	$(1\%, 3\%)$
v_0	$(3\%, 5\%)$
ϑ	$(3\%, 5\%)$
κ	$(2\%, 4\%)$
ξ	$(3\%, 5\%)$
ρ	$(-70\%, -50\%)$

(b) Heston

Figure 6 Parameter Ranges Training Set

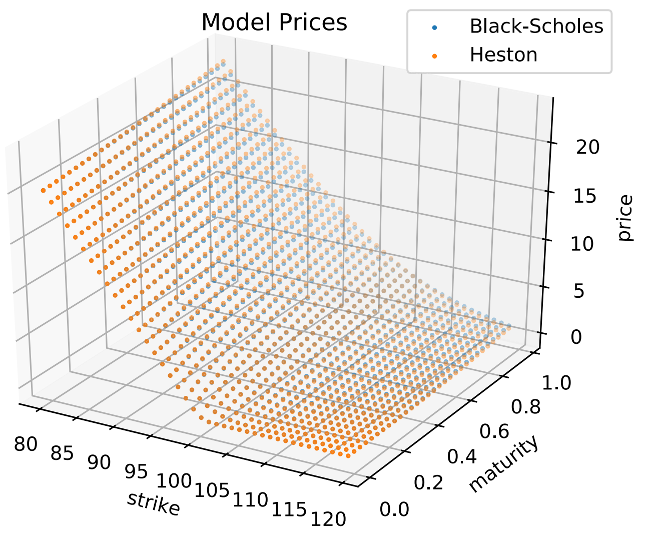


Figure 7 Data Set

LSTMs we choose the original number of units such that the total number of trainable weights in the first column of the LSTM grid is the same as in the MLP grid. This ensures the degrees of freedom of the LSTMs are comparable to the MLPs. The resulting graph of weights is shown in Fig. 8. The key insight here is that while their shape looks the same as expected, the order of magnitude of original number of units is much lower for the LSTM as for the MLP. That is because all these additional complexities of the LSTM, recall Definition 1.3, mean that for the same number of units, the LSTM has much more trainable weights than the MLP. Thus, in order to achieve the same number of trainable weights as the MLP, the LSTM has to be instantiated with a much lower number of units.

We train both models on the above data set with an 80%/20%-split into train/test data with random shuffling. The thresholds are set to $t_b := t_v := 0.25$ and the maximal number of epochs is $e_{\max} := 50$. We choose the mean squared error (MSE) as a loss function and the mean absolute error (MAE) as our metric. The resulting learning curves with the bias and the variance are shown in Fig. 9 for the MLP. In this grid the number of layers increases in each column from left to right and the number of original units increases in each row from top to bottom. We find that the first column does not have enough layers to capture the

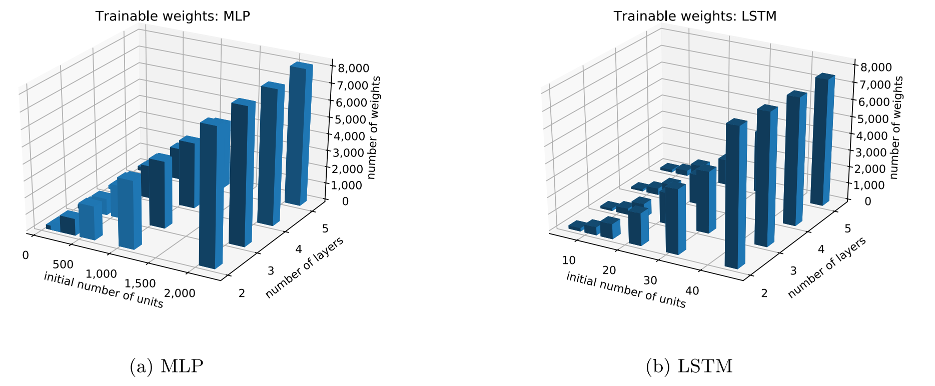


Figure 8 Number of trainable weights

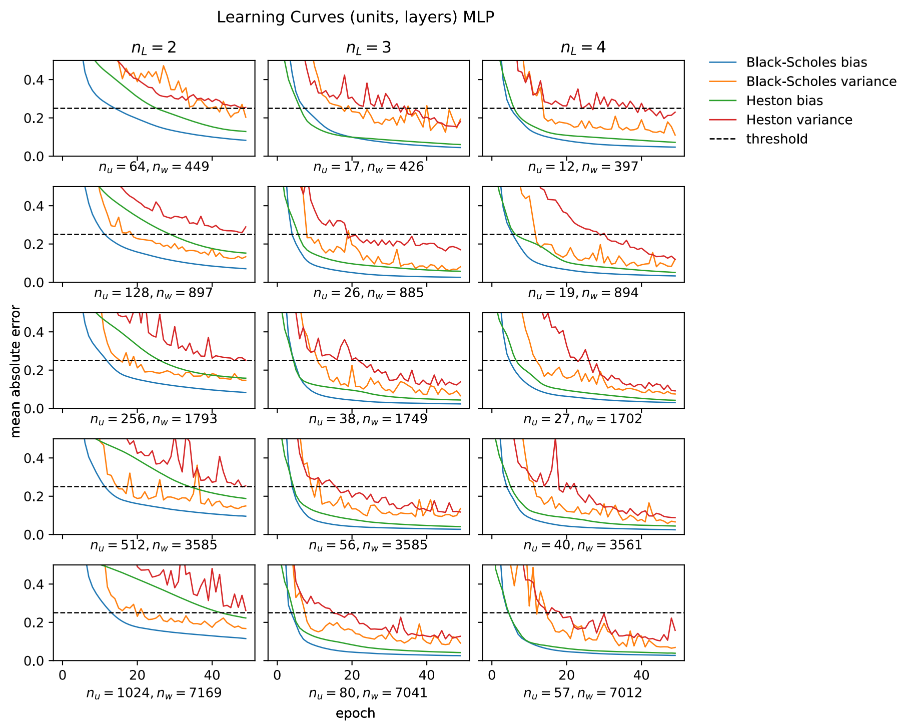


Figure 9 Learning Curves MLP

non-linearity in the pricing function of the Black–Scholes or the Heston model. However, in the second column, the first row is already below the threshold, but only barely and not yet very stably after the 50 epochs, so we select the model below in the second row. This model has $L = 3$ layers and just $n_u = 26$ units in the hidden layers (reduced from the original number of 128). This means that this model has learned the Black–Scholes and Heston pricing function with only $n_w = 885$ trained weights after just 50 epochs.

The learning curves for the corresponding LSTMs are shown in Fig. 10. We find that they are significantly worse than the MLP. Only for the very last model for $L = 3$ layers and $n_u = 17$ units per hidden layer (reduced from an original number of 1024), the learning curves are just at the threshold, so we select this one. Despite having $n_w = 6274$ trained

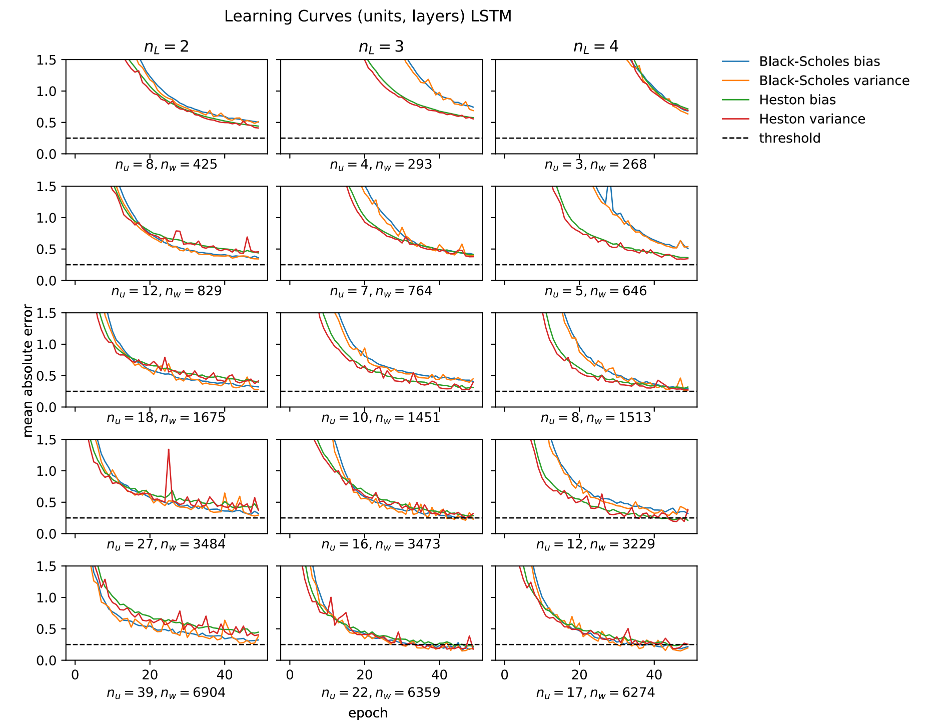


Figure 10 Learning Curves LSTM

weights in total, it still performs worse than the MLP we selected above (which has less than 7x the number of trained weights). The conclusion from this is not that LSTMs cannot learn the Black–Scholes or Heston pricing function, but rather that the simplicity of the MLP network topology is better adapted to this specific problem. The complexity of the LSTM network topology is adapted to a situation, which doesn't occur here, and thus it cannot achieve the same performance as the MLP when constrained by the same number of trainable weights.

5.3 Error distribution

While the mean absolute error (MAE) is a good metric for the learning curve, it is often not sufficient for financial applications. Therefore, we study the whole error distribution for the MLP and the LSTM selected above, see Fig. 11. Unsurprisingly, the LSTM has a higher and a wider error distribution indicating it has learned the pricing functions less well than the MLP. For the MLP it is interesting to note that while the learning curves in Fig. 9 suggest that in the mean error for the Black–Scholes and the Heston model is similar, we see in Fig. 11, that the Heston error distribution is wider than Black–Scholes. This shows that it is 'harder' for the network to learn this, which given the higher dimensionality of the data set is not surprising.

Both impressions are also confirmed Fig. 12, where we compute some statistics of the error distribution. While for MLP, high percentiles or even the max of the error is still <1, meaning in cases where the MLP error is worse than the mean, this network fails 'gracefully', the max error for the LSTM is much worse. For the MLP, the 95th and 99th quantile of the error as well as the max error are slightly higher for Heston than for Black–Scholes.

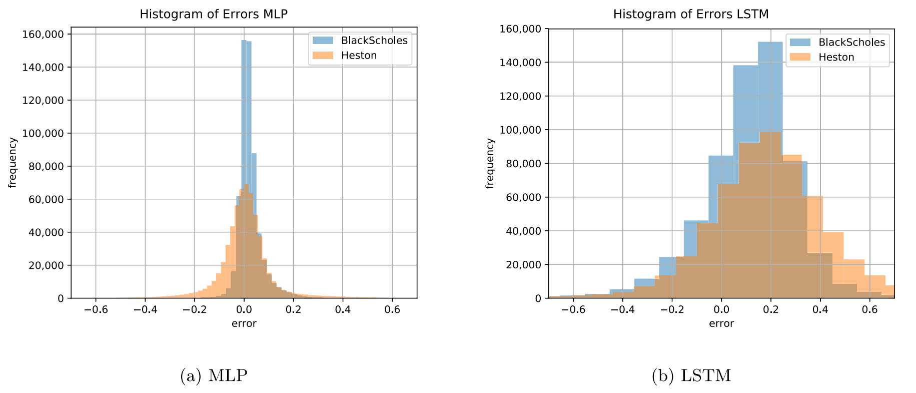


Figure 11 Distribution of Error

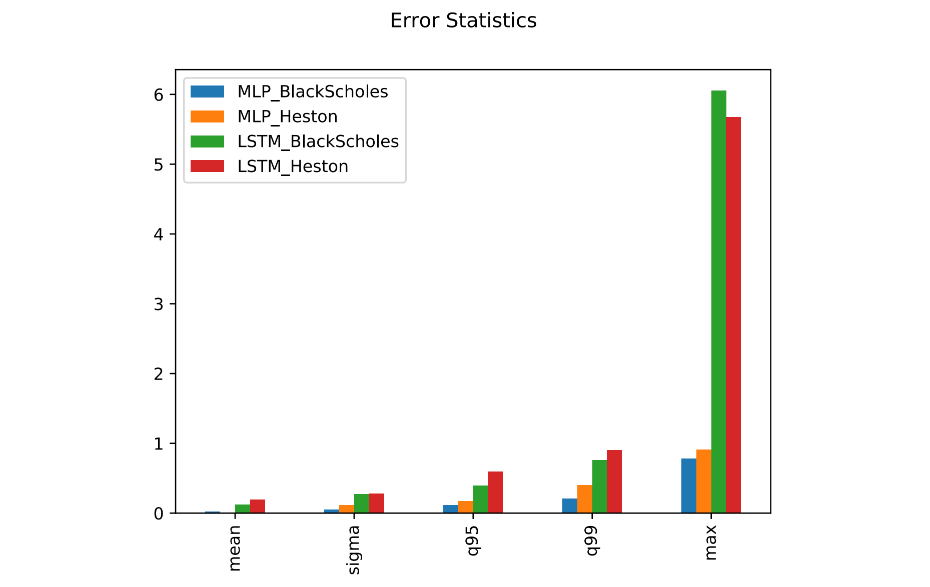


Figure 12 Error Statistics

5.4 Generalization to other models

We have illustrated the network topology selection method for parametric models trained on synthetic data in a given range as this flavour is currently the most popular. It is well known that neural networks usually do not perform very well in regions outside the training set, i.e. when they extrapolate. Therefore, we suggest to take that into account when generating the training set. The fact that this is possible is one of the big advantages of working with synthetic data (rather than real world or historic data). In practice, we recommend an automated bound checking of the new inputs supplied to the network that is consistent with the bounds used in training. Even more care has to be taken when training non-parametric models on historic data, e.g. a VaR model on historic market data shifts. The high dimensionality of the input and non-linearity of the PnL might require a topology so big that given limited availability of historic data, it might simply not be trainiable to get the bias below an acceptable threshold. This will also make it impossible to reduce the variance to satisfactory level. If the network is trained on a decade where markets are calm, it will also easily breach the ranges in which it was trainined when used in a time of market stress causing extrapolation failures.

6 Conclusions

We conclude that the SR 11-7 requirement to conduct model validation, in particular a thorough model selection process, can be satisfied for neural network models as well. The simple grid based network topology selection method is a pragmatic way of producing a good and documented choice of hyperparameters for a given financial application.

As a byproduct we obtain interesting insights into how neural networks learn financial models. Given that evaluating a network is very fast, this makes the use of pricing models feasible, which are too computationally expensive otherwise. We also find that while option prices (or implied volatilities) clearly have a time dependence, LSTMs are overly complicated for this application and the much simpler MLPs perform better in a fair comparison – a textbook case of Occam's razor.

Acknowledgements
We would like to thank Gordon Lee for interesting discussions and feedback. We would also like to thank the reviewer for constructive comments.

Funding
Open Access funding enabled and organized by Projekt DEAL.

Availability of data and materials
Data and an illustrating notebook is available on github:
https://github.com/niknow/machine-learning-examples/blob/master/network_topology_selection

Declarations

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
All authors contributed equally to the paper. All authors read and approved the final manuscript.

Author details
[1]Acadia, London, UK. [2]Fachbereich Mathematik und Naturwissenschaften, Bergische Universität Wuppertal, Wuppertal, Germany. [3]The African Institute for Financial Markets and Risk Management (AIFMRM), University of Cape Town, Cape Town, South Africa. [4]Acadia, Bonn, Germany. [5]Acadia, Frankfurt, Germany. [6]Airplus, Frankfurt, Germany.

Publisher's Note

Received: 26 January 2021 Accepted: 16 December 2021 Published online: 03 January 2022

References
1. Abadi M, Agarwal A, Barham P et al. TensorFlow: large-scale machine learning on heterogeneous systems. 2015. https://www.tensorflow.org/.
2. B. of Governors of the Federal Reserver System/Office of the Comptroller of the Currency. Governors of the Federal Reserver System/Office of the Comptroller of the Currency. Supervisory Guidance on Model Risk Management. 2010. https://www.federalreserve.gov/supervisionreg/srletters/sr1107a1.pdf.
3. Bayer C, Stemper B. Deep calibration of rough stochastic volatility models. 2018. https://arxiv.org/abs/1810.03399.
4. Black F, Scholes M. The pricing of options and corporate liabilities. J Polit Econ. 1973;81(3):637–54. https://doi.org/10.1086/260062.
5. Buhler H, Gonon L, Teichmann J, Wood B. Deep Hedging. 2019. https://arxiv.org/abs/1802.03042.
6. Chollet F, et al. Keras. 2015. https://keras.io.
7. Crisostomo R. An analysis of the heston stochastic volatility model: implementation and calibration using Matlab. 2014. https://ssrn.com/abstract=2527818.
8. Cybenko G. Approximation by superpositions of a sigmoidal function. Math Control Signals Syst. 1989;2(4):303–14. https://doi.org/10.1007/BF02551274. issn: 0932-4194.

9. Ern A, Guermond J. Theory and practice of finite elements. Applied mathematical sciences. New York: Springer; 2004. ISBN 9780387205748. https://books.google.co.uk/books?id=CCjm79FbJbcC.
10. Hastie T, Tibshirani R, Friedman J. The elements of statistical learning. Springer series in statistics. New York: Springer; 2001.
11. Haykin S. Neural networks: a comprehensive foundation. 2nd ed. New York: Prentice Hall; 1998. ISBN 0132733501.
12. Hernandez A. Model Calibration with Neural Networks. 2016. https://ssrn.com/abstract=2812140.
13. Heston SL. A closed-form solution for options with stochastic volatility with applications to bond and currency options. Rev Financ Stud. 2015;6(2):327–43. https://doi.org/10.1093/rfs/6.2.327. issn: 0893-9454. https://academic.oup.com/rfs/article-pdf/6/2/327/24417457/060327.pdf.
14. Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput. 1997;9:1735–80.
15. Hornik K. Approximation capabilities of multilayer feedforward networks. Neural Netw. 1991;4(2):251–7. http://www.sciencedirect.com/science/article/pii/089360809190009T.
16. Horvath B, Muguruza A, Tomas M. Deep Learning Volatility. 2019. https://arxiv.org/abs/1901.09647.
17. Jin H, Song Q, Hu X. Efficient neural architecture search with network morphism. In: CoRR. 2018. http://arxiv.org/abs/1806.10282.
18. Kingma DP, Ba J. Adam: a method for stochastic optimization. 2014. 1412.6980 [cs.LG].
19. Liu S, Oosterlee CW, Bohte SM. Pricing options and computing implied volatilities using neural networks. 2019. https://arxiv.org/abs/1901.08943.
20. Muller A, Guido S. Introduction to machine learning with python: a guide for data scientists. Sebastopol: O'Reilly Media; 2018. ISBN 9789352134571. https://books.google.co.uk/books?id=jGdXswEACAAJ.
21. Murphy KP. Machine learning: a probabilistic perspective. Cambridge: MIT Press; 2013. ISBN 9780262018029. https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020/ref=sr_1_2?ie=UTF8&qid=1336857747&sr=8-2.
22. Rudin W. Functional analysis. International series in pure and applied mathematics. New York: McGraw-Hill; 1991. ISBN 9780070542365. https://books.google.co.uk/books?id=Sh/_vAAAAMAAJ.
23. Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Nature. 1986;323(6088):533–6. http://www.nature.com/articles/323533a0.
24. Timan A. Theory of approximation of functions of a real variable. Dover books on advanced mathematics. New York: Dover; 1994. ISBN 9780486678306. https://books.google.co.uk/books?id=JlJvzgQf/_2IC.