

Internet Appendix for “Predicting Option Prices From Their Price History Via Machine Learning”

This Internet Appendix contains theoretical foundations as well as several additional tables and figures that complement the results presented in the main paper.

IA.1 Theoretical foundations on neural networks

This section describes the neural network models used in the empirical analyses described in the main part of the paper. After providing some general information on neural networks we introduce feed forward neural networks.

IA.1.1 Introductory information

Work on neural networks has been motivated by the fact that the human brain solves problems in an entirely different way than conventional digital computers (see [Haykin, 2009](#)). In fact, some of the earliest machine learning algorithms have been specifically designed as models of biological learning, that is, to replicate how learning happens in the brain (see [Goodfellow et al., 2016](#)). A remarkable feature of neural networks is that they are able to use abstraction to model the mapping between input and output values by building complex functions out of many simpler (often linear) functions (cf. [Goodfellow et al., 2016](#)).

Neural network models are most often employed in supervised machine learning where the aim is to model the input-output relationship of a potentially complex system based on labeled data examples (pairs of inputs and outputs). Once the network has been trained on the data, it can subsequently be used to provide estimates of the labels. This can for example be used in a classification setting where the aim is to assign a data example to one out of a finite set of discrete values (the inputs' class labels). However, when the output data are continuous like the IVs in our empirical study, we are in a regression context. In the following, we will focus on the latter.

IA.1.2 Artificial neural networks

The last decades have seen an increasing amount of research on neural networks such that the term “neural network” or “neural net” has evolved to cover a large class of different models and associated learning algorithms. In this section we introduce the most widely used “vanilla” neural networks (single hidden layer feed forward networks) sometimes called the *single layer perceptron* (see [Hastie et al., 2017](#)) before turning to deeper networks (*multilayer perceptron*). In the following, we will denote these models as *artificial neural networks* (ANN) regardless of the number of hidden layers. Our introduction of the single layer perceptron is based on [Hastie et al. \(2017\)](#).

The network architecture

A single layer perceptron can be interpreted as a two stage regression model. We provide an illustration of the model in form of a network diagram in Figure [IA.1](#). The model there consists of three layers, the input layer, the hidden layer, and the output layer and features $P \in \mathbb{N}$ input nodes (also called neurons) and one output node.¹ This corresponds to a mathematical function mapping P arguments to a scalar value. More formally, the ANN models the functional relationship²

$$f : \mathbb{R}^P \rightarrow \mathbb{R}, \quad x = (x_1, \dots, x_P) \mapsto y. \quad (4)$$

In addition to the input and output layer the model consists of a hidden layer with an arbitrary number of neurons (M neurons in Figure [IA.1](#)). In the hidden layer the model achieves abstraction by transforming the input values non-linearly into the feature space (the values $z_1, \dots, z_M$). In this new space, dependencies between inputs and outputs might be picked up more easily (see [Haykin, 2009](#)). Finally, the output $y = f(x)$ is derived from these unobserved (*hidden*) features as a linear combination to which an ultimate (potentially) non-linear transformation g is applied. More formally, we have

$$\begin{aligned} z_m &= \sigma(a_{0m} + a_m^T x), \quad m = 1, \dots, M, \\ o &= \beta_0 + \beta^T z, \\ f(x) &:= g(o), \end{aligned}$$

where $z := (z_1, \dots, z_M)$.

The non-linearity in the mapping from the vector x to z comes from the functions σ and g which are often chosen to be the rectified linear unit (ReLU) function³, the sigmoid function⁴, or

¹One output node is most typical in a regression context while for classification tasks one usually relies on $K > 1$ nodes to model the probability of an input vector belonging to one of K classes. In this paper, we focus on the regression case. A generalization to K outputs is, however, straight forward. A presentation for the K output case can be found in [Hastie et al. \(2017\)](#).

²The input data are often required to be in the unit hypercube $[0, 1]^P$. This can be achieved by scaling the inputs and will be discussed later.

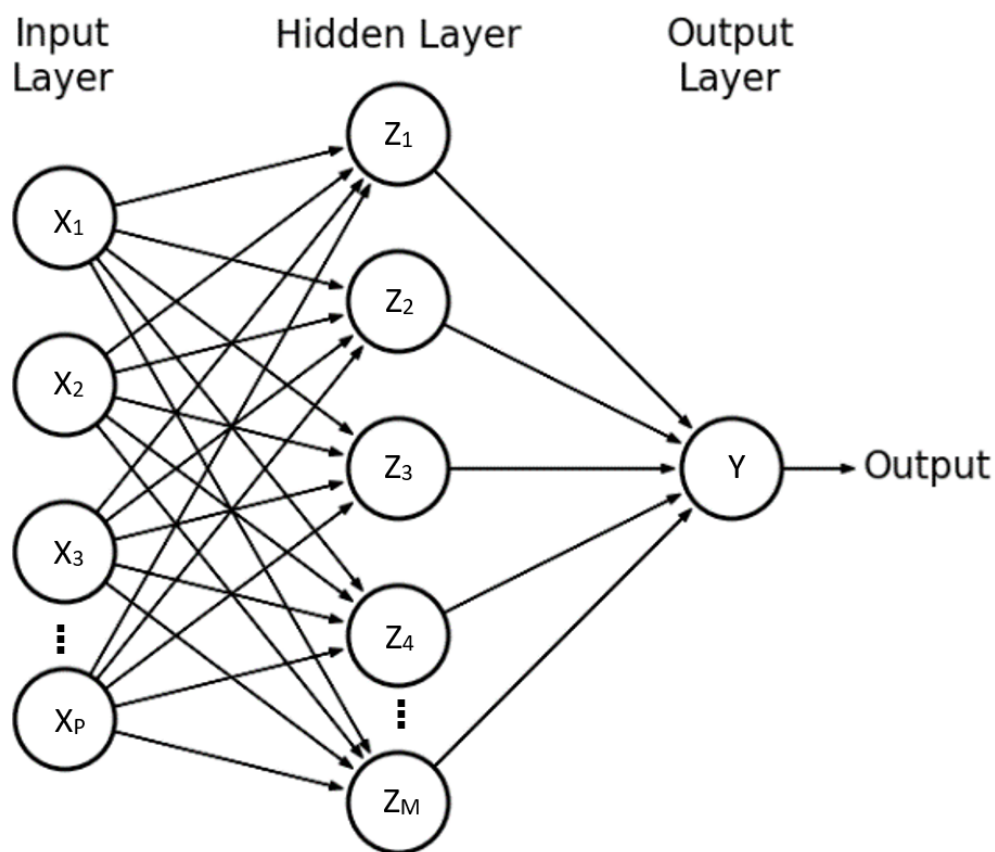
³The ReLU function is defined as $\sigma : \mathbb{R} \rightarrow [0, \infty), x \mapsto \max(0, x)$.

⁴The term *sigmoid function* refers to a broader class of functions. However, in machine learning the term typically denotes the logistic function given by

$$\sigma : \mathbb{R} \rightarrow (0, 1), \quad x \mapsto \frac{e^x}{e^x + 1}.$$

Figure IA.1: Single layer perceptron

This figure provides an illustration of the simplest form of an ANN, the single layer perceptron model, in form of a network diagram. The model consists of three layers, the input, the hidden, and the output layer. The neurons of one layer are connected to all neurons of the directly following layer. As in the paper we focus on regression tasks, the model only has one output neurons. However, a generalization to more than one output neuron is straight forward.



Source: Own representation based on [Hastie et al. \(2017\)](#).

the radial basis function⁵. These non-linear transformations are a great strength of neural networks that allow them to model complex functional dependencies. Additionally, if all transformations in the neural network were linear, there would be no point in including hidden layers at all. In this case one would simply derive the vector z via multiplication of x with the weight matrix formed by the vectors $a_m, m = 1, \dots, M$ and analogously y from z via multiplication with β .⁶ As a consequence, one could instead directly infer y from x by multiplying the vector x with the product of the weight matrix and β making the hidden layer superfluous. Thus, one can think of a neural network as a generalization of a linear model.

After linearly mapping the hidden features $z_1, \dots, z_M$ to o , we employ a final transformation g to obtain the output y . While in a regression context one usually relies on the identity function, in a classification context one often uses the softmax function which is the same function as in the multinomial logistic regression model.

Of course, one can add more than one hidden layer to the ANN, thereby allowing the ANN to extract higher-order statistics from its inputs. Typically, the neurons in each layer only have access to the outputs of the neurons in the preceding layer (see [Haykin, 2009](#)). By adding multiple hidden layers we obtain deep networks which are currently the preferred approach for solving many complex machine learning problems. However, increasing the number of hidden layers adds substantial complexity to the models. Consequently, deep networks rank among the least transparent and most highly parameterized machine learning algorithms (see [Sirignano and Cont, 2019](#)).

Universal approximation theorem

Neural networks can approximate any smooth function on a compact set uniformly with an arbitrarily small error. Therefore, neural networks are often referred to as *universal approximators*. Most interestingly, this requires neither a complex nor a deep network architecture. Actually, a

⁵Radial basis functions are symmetric kernels that are employed to achieve localization around a so-called centroid μ . A radial basis function can be represented as

$$\sigma(x) = \sum_{j=1}^M K_{\lambda_j}(\mu_j, x) \beta_j,$$

where the β_j 's are weights and a popular choice for the functions K_{λ_j} is the Gaussian kernel given by $K_{\lambda}(\mu, x) = e^{-||x-\mu||^2/2\lambda}$ with λ denoting a so-called bandwidth parameter. For more details we refer to [Hastie et al. \(2017\)](#).

⁶Many illustrations of ANNs add a bias that feeds into each neuron in the hidden and output layer. This can already be included in the weight matrix by adding the constant "1" as an additional input feature.

vanilla feed forward neural network with sigmoidal activation functions and one hidden layer is sufficient. This result is due to [Hornik et al. \(1989\)](#) and [Cybenko \(1989\)](#). We present the theorem in the formulation by [Haykin \(2009\)](#).

Theorem 1 (Universal approximation theorem) *Let $\phi(\cdot)$ be a non-constant, bounded, and monotone-increasing continuous function. Let I_{m_0} denote the m_0 -dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Then, given any function $f \in C(I_{m_0})$ and $\epsilon > 0$, there exists an integer m_1 and sets of real constants a_i, b_i , and ω_{ij} , where $i = 1, \dots, m_1$ and $j = 1, \dots, m_0$ such that the function*

$$F(x_1, \dots, x_{m_0}) := \sum_{i=1}^{m_1} a_i \phi \left(\sum_{j=1}^{m_0} \omega_{ij} x_j + b_i \right) \quad (5)$$

fulfills

$$|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})| < \epsilon$$

for all $x_1, x_2, \dots, x_{m_0}$ in the unit hypercube $[0, 1]^{m_0}$.

More shortly, the theorem states that functions of the form given by Equation (5) are dense in $C(I_{m_0})$.

Equation (5) represents the output of an ANN network with activation function $\phi(\cdot)$. The network has m_0 input neurons with inputs denoted as $x_1, \dots, x_{m_0}$. The network has one hidden layer with m_1 neurons. The weights for hidden neuron i are given by $\omega_{i1}, \omega_{i2}, \dots, \omega_{im_0}$ with bias b_i . The network uses a linear activation function for the (single) output neuron with weights $a_1, \dots, a_{m_1}$. The theorem is an existence theorem and unfortunately is not constructive. This means that the theorem does not provide any information on how to choose the number of neurons or the various parameters. In fact, the theorem does not even state that a single hidden layer is the optimum in terms of computational efficiency, ease of implementation, or generalization to new data.

Training the network

As already mentioned, neural networks were first developed as models of the human brain. The analogy is that the nodes in the models can be thought of as neurons in the brain where the

connections between the nodes represent synapses. Early models employed a step function instead of smooth functions to model neurons that process and pass signals if and only if the sum of all incoming signals exceeds a certain threshold. The strength of particular synapses between neurons is represented by the weight parameters.

Based on these weights, ANNs approximate functional dependencies $y = f^*(x)$ between input (x) and output values (y). Therefore, instead of viewing the network as a function $f = f(x)$ like in Equation (4) a more appropriate view would be to think of it as a function $f = f(x; \theta)$ where θ represents the weights (and biases). The aim of training (or fitting) the network is then to find the (multi-dimensional) parameter θ that yields the best approximation of f^* (see [Goodfellow et al., 2016](#)).

As a consequence of the non-linearity of the ANN, most loss functions become nonconvex. Such a cost function might have many local minima instead of a global minimum. Therefore, neural networks are usually trained based on gradient-based optimizers that iteratively decrease the value of the cost function. However, there is no guarantee of achieving a global or even a local minimum. Additionally, results may depend on the starting values. That is, while in convex optimization convergence to the optimal parameters is guaranteed from any starting point, gradient descent is sensitive to the parameters that are used to initialize the network. Usually, the weights of ANNs are initialized to small random values while the biases are set to zero or small positive values (see [Goodfellow et al., 2016](#)).

In the context of training neural networks, gradient descent is often referred to as *backpropagation*. Thanks to the compositional form of the ANN (one layer put after the other), gradients of the cost function (in regression tasks usually the MSE) can be derived via the chain rule for differentiation. As a consequence, calculation of the gradients can be implemented via a two-pass algorithm. Given N training examples $(x_i, y_i), i = 1, \dots, N$, in the so-called *forward pass* one computes the predicted values $\hat{f}(x_i), i = 1, \dots, N$ while the current weights are assumed to be fixed. Comparing the predicted values to the actual ones (y_i) based on the respective cost function yields errors that can be propagated backwards from the output layer over the hidden layers to the input layer to indicate necessary adjustments to the weights and biases in every layer.⁷ For the corresponding

⁷Backpropagation was first developed in 1970 by the Finnish master student Seppo Linnainmaa and published in [Linnainmaa \(1976\)](#).

backpropagation equations we refer to [Hastie et al. \(2017\)](#). The main advantage of backpropagation is its simple local nature, that is, each neuron only receives and passes information from and to nodes that have a direct connection. As a consequence, calculations can be performed efficiently on GPUs that offer a highly parallel architecture for performing matrix multiplications (cf. [Hastie et al., 2017](#)).

However, as datasets can be quite large, calculating gradients based on the whole training sample (so-called *batch learning*) and backpropagating them through the network can be computationally very expensive. Therefore, instead of the gradient descent algorithm, one usually relies on *stochastic* gradient descent (SGD) for training neural networks thereby following the gradient of a randomly selected subsample downhill. This is also often referred to as mini-batch learning.⁸ The most important advantage of the SGD algorithm over the gradient descent algorithm is that the computation time for each iteration step does not increase with the number of training examples. While asymptotic convergence (in terms of number of iterations) of the SGD is slower than for its non-stochastic counterpart, the ability of the SGD algorithm to achieve rapid initial progress by evaluating only a subset of data examples makes it the most used optimization algorithm for training neural networks (see [Goodfellow et al., 2016](#)).

IA.1.3 Recurrent neural networks

The ANNs discussed in the previous section are feed forward neural networks as the information flows from the input neurons directly through the hidden layers to the output neuron, see Figure [IA.1](#). In particular, there are no feedback connections in which outputs of the model enter as inputs into the model again (cf. [Goodfellow et al., 2016](#)). When extending feed forward networks to contain feedback connections, we obtain recurrent neural networks (RNNs). For example, when connecting the outputs of all neurons to the inputs of all neurons, one yields a fully recurrent neural network. In particular, this network contains self-feedback loops where the output of a neuron is fed into its own input (see [Haykin, 2009](#)). RNNs use an internal state (memory) to process input sequences allowing these networks to exhibit temporal dynamic behavior (see, e.g., [Tealab, 2018](#); [Abiodun et al., 2018](#)). This makes them suitable candidates for modeling sequential data such as

⁸We discuss the size of the mini-batches for the empirical study in Section [IA.1.4](#). An important parameter for the SGD algorithm is the learning rate that determines the extent to which single weights are adjusted depending on the gradient in each iteration. More details are provided *ibid*.

time series. Other applications include speech and handwriting recognition (see, e.g., [Graves et al., 2009](#); [Sak et al., 2014](#); [Li and Wu, 2015](#)).

RNNs are based on [Rumelhart et al. \(1986\)](#) with other early contributions made by [Schmidhuber \(1992\)](#); [El Hihi and Bengio \(1995\)](#), and [Jaeger \(2007\)](#). Computations in RNNs can typically be decomposed into three transformations including the transformation from the input to the hidden state, from the previous hidden state to the next hidden state, and from the hidden state to the output. The network can be unfolded into a feed forward network where each of these transformations corresponds to a shallow transformation.⁹ As a consequence, a one layer RNN can be substantially more complex than a multi hidden layer feed forward neural network. However, there are several ways to make a RNN deep, for example by breaking down hidden recurrent states into multiple layers (see [Pascanu et al., 2014](#), for more information). There is experimental evidence that introducing depths into RNNs can lead to superior performance (see [Graves, 2013](#); [Pascanu et al., 2014](#)).

Training of RNNs is very similar to feed forward neural networks. In fact, one simply applies the backpropagation algorithm outlined in the previous section to the unfolded network. This is called backpropagation through time (for details see [Werbos, 1990](#)). As a consequence, no specialized algorithms are needed and gradients obtained via backpropagation through time can be used with any gradient-based technique that is also applicable to feed forward networks (see [Goodfellow et al., 2016](#)). However, propagating gradients back over many stages in RNNs often poses serious mathematical challenges. This complicates learning long-term dependencies as over many stages gradients tend to either vanish or explode.¹⁰ At its core, the problem is due to the fact that RNNs require the application of the same transformation repeatedly. In the simplest case of linear transformations this corresponds to multiplying the same matrix multiple times with itself. The eigenvalues of the product are powers of the eigenvalues of the original matrix. Consequently, eigenvalues smaller 1 (of the original matrix) converge towards 0 (in the product matrix) while eigenvalues > 1 converge towards infinity. Apart from the vanishing or exploding gradient problem, this also complicates learning long-term dependencies as exponentially smaller weights are given to

⁹A shallow transformation denotes a transformation that can be represented by a single layer of an ANN, see [Goodfellow et al. \(2016\)](#).

¹⁰These problems were independently discovered and described by [Hochreiter \(1991\)](#) and [Bengio et al. \(1993, 1994\)](#).

long-term interactions compared to short-term ones (see [Goodfellow et al., 2016](#)).

A model that was specifically designed to capture long-term dependencies while trying to solve the vanishing gradient problem is the long short-term memory (LSTM) model by [Hochreiter and Schmidhuber \(1997\)](#). It is among the most effective sequence models that are used in practical applications including speech and unconstrained handwriting recognition (cf. [Graves et al., 2009](#); [Graves, 2013](#); [Graves and Jaitly, 2014](#)), machine translation (see [Sutskever et al., 2014](#)), and parsing ([Vinyals et al., 2015](#)). Of course, there are various other RNN architectures. Early examples are the Hopfield network by [Hopfield \(1982\)](#) and the Elman network by [Elman \(1990\)](#) as well as the Neural history compressor by [Schmidhuber \(1992\)](#). More complex models are, among others, the Gated recurrent unit by [Cho et al. \(2014\)](#), the continuous time RNN ([Pearlmutter, 1995](#)), and recurrent multilayer perceptron networks ([Pineda, 1987](#)). However, because of its superior time series modeling ability ([Goodfellow et al., 2016](#)) and broad applicability we will focus on the (plain) LSTM model along with two variants (bidirectional and convolutional LSTM).

(Plain) LSTM model

The basic idea of LSTM models is to create paths through time where gradients neither vanish nor explode. This is achieved via connection weights that can change over time allowing the LSTM model to accumulate information over long periods. On the other hand, the model is also able to forget old states which is particularly useful when modeling a sequence of subsequences where resetting the old state at the beginning of each new subsequence might prove useful.

More detailed, the LSTM model consists of specialized neurons that are called LSTM cells. These LSTM cells introduce self-loops that feed the previous state of the cell together with the new input back into the cell. This allows the LSTM cell to store previous experiences and memorize dependencies between the data. The idea of self-loops is, however, not new to LSTM models but is a regular feature of RNNs. What distinguishes the LSTM model from other earlier models is that instead of a regular neuron the LSTM cell contains several layers and gates that control the memorizing process by regulating how new information enter into the state (the memory of the cell), when the memory should be reset, or which fraction of the information should be forwarded.

The following more formal description of the LSTM model builds on [Goodfellow et al. \(2016\)](#). For each time step t and LSTM cell i two major transformations are performed. First, the internal

cell state $c_i^{(t)}$ is updated according to the formula

$$c_i^{(t)} := f_i^{(t)} c_i^{(t-1)} + \iota_i^{(t)} \eta_i^{(t)}, \quad (6)$$

where the conditional self-loop weight $f_i^{(t)}$ (the *forget gate* unit) and the *external input gate* unit $\iota_i^{(t)}$ assume values between 0 and 1 and govern the extent to which the cell state is adjusted depending on the current input vector $x^{(t)}$ and the hidden layer vector $h^{(t-1)}$.¹¹ The value $\eta_i^{(t)}$ is a scalar that represents the new (preprocessed) information provided in the current time step and is also derived from $x^{(t)}$ and $h^{(t-1)}$.

Based on the updated cell state $c_i^{(t)}$, the output $h_i^{(t)}$ of the LSTM cell is calculated via

$$h_i^{(t)} := \tanh(c_i^{(t)}) o_i^{(t)}, \quad (7)$$

where $o_i^{(t)}$ denotes the *output gate* unit that like the forget gate and external input gate units assumes a value between 0 and 1 depending on the current input vector $x^{(t)}$ and the previous outputs $h^{(t-1)}$ of the LSTM cells. We now provide some details on the calculation of the three gate units and the scalar $\eta_i^{(t)}$.

The forget gate unit from Equation (6) is defined as

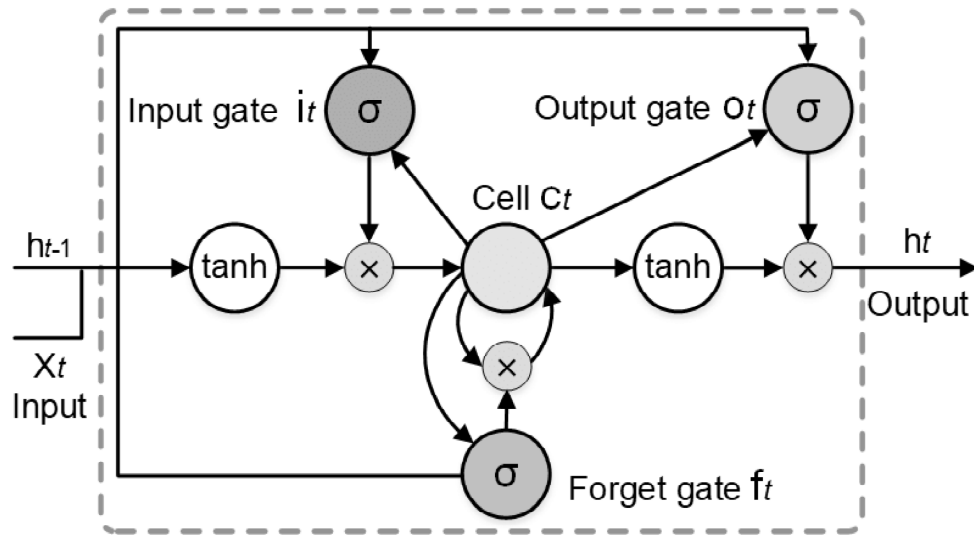
$$f_i^{(t)} := \sigma \left(b_i^f + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right),$$

where b^f, U, W are the biases, input weights, and recurrent weights for the forget gates, respectively, and $x^{(t)}$ and $h^{(t-1)}$ are as above. The result from this linear transformation is gated by the sigmoid function σ . The external input gate unit $\iota_i^{(t)}$ and the output gate unit $o_i^{(t)}$ as well as $\eta_i^{(t)}$ are defined completely analogously. The only difference is that the calculations rely on other weight matrices and bias vectors and in case of $\eta_i^{(t)}$ the hyperbolic tangent function is used instead of the sigmoid function. Note, however, that although the gates and $\eta_i^{(t)}$ are calculated similarly, they fulfill very different functions. While the gates control how much of the memory is forgotten, how much new information enters into the memory, and how much of the new memory is outputted, $\eta_i^{(t)}$ represents the new (preprocessed) information at a particular timepoint itself. This interaction

¹¹The vector $h^{(t-1)}$ contains the outputs of *all* LSTM cells of the respective layer from the previous time step.

Figure IA.2: Structure of a LSTM cell

A standard LSTM cell consists of an input, an output, and a forget gate. The information flow is depicted by arrows. The cell can remember values over arbitrary time periods in the cell state c_t while the flow of information into the cell (new inputs, previous outputs) and out of the cell is controlled by the three gates. Sigmoidal activation functions are symbolized by σ while $\tanh$ refers to the hyperbolic tangent function. The symbol $\otimes$ represents multiplication with a weight matrix.



Source: Long short-term memory structure by [Xue et al. \(2019\)](#) licensed under CC BY 4.0

between different gates and layers is illustrated in Figure [IA.2](#).

Bidirectional and convolutional LSTM models

Bidirectional RNNs were proposed by [Schuster and Paliwal \(1997\)](#) to increase the amount of information that is available to the network. The bidirectional network is allowed to reach future input information from its current state as opposed to standard RNNs like the LSTM model described above. Bidirectional LSTM models connect two layers of LSTM cells in opposite directions to the same output. The outputs of one layer are not connected to the inputs of the other layer. As a consequence, the model can be trained with algorithms similar to the plain LSTM model (backpropagation through time).¹² Bidirectional LSTM models are especially useful in applications where context is needed like for example in speech or handwriting recognition ([Goodfellow et al., 2016](#)). In our empirical analysis, however, we apply the model to option prices. While subsequent prices can not influence previous ones, we use subsequent observations within the same sample window as a proxy for market expectations. Importantly, the model is trained and evaluated in a way that prevents any look-ahead bias when making predictions. The rationale for including a bidirectional LSTM is that by processing information from both temporal directions within the available sample, the model may better capture the market's implicit expectations and temporal dependencies.

Another variant of the LSTM model is the convolutional LSTM model due to [Shi et al. \(2015\)](#). The convolution operator gives a network the possibility to share parameters over time by using the same convolution kernel at each time step. The output of a convolution is again a sequence of values where each value is a function of a small number of neighboring observations of the input ([Goodfellow et al., 2016](#)). More detailed, the convolutional LSTM is obtained by extending the fully connected LSTM by [Srivastava et al. \(2015\)](#) to obtain a network in which the gates (input to state as well as state to state transitions) perform convolution operations. Thus, the future state of a particular cell is determined based on the inputs and past states of its local neighbors. For more details we refer to the original paper. The convolutional LSTM model is especially suited for the forecasting of spatiotemporal sequences. In our empirical study, however, we do not have spatial

¹²For further details on the training of the model as well as the model itself we refer to [Schuster and Paliwal \(1997\)](#).

information. Instead, we hope that the additional level of abstraction from deriving features from the inputs and states of neighboring cells helps in detecting hidden patterns in the price series.

IA.1.4 Further remarks on training neural networks

In this section we discuss further aspects concerning the specification and training of the models. For brevity, we will from now on subsume the ANN as well as the LSTM, bidirectional, and convolutional LSTM under the term *neural networks*.

One often normalizes the input data before feeding them into the input layer of the network. Although in theory scaling is rarely strictly necessary as any rescaling of an input vector can be replicated by scaling the corresponding weights and biases, there is a variety of practical reasons why standardizing the input data can lead to faster convergence and thus speed up learning. A popular choice is to linearly transform the input vectors to the interval $[-1, 1]$ (Gu et al., 2020).¹³

As discussed previously, all network models are fitted based on some form of stochastic gradient descent where at each time only a fraction of the available data is used. This type of optimization is often referred to as *mini-batch learning* and methods that only rely on one data example at a time are also sometimes called online methods (Goodfellow et al., 2016). The size of the mini-batches fed into the gradient descent algorithm serves as a hyperparameter where the choice of a specific batch size has to take several factors into account.¹⁴ First of all, gradients derived from larger batches are more accurate as they are averaged over more data examples and thus less prone to noise in the data. However, in practice often all data examples in a batch are processed in parallel. As a consequence, its size is limited by the available memory. On the other hand, smaller batch sizes can have a regularizing effect leading to models that are less prone to overfitting (cf. Wilson and Martinez, 2003). This comes at the cost of an increased runtime as more steps are necessary to process the entire training sample (the errors have to be backpropagated through the network more often). Finally, depending on the hardware one sometimes can achieve better runtimes with power of 2 batch sizes. This is especially true when using GPUs for training as in the analysis for time series models and the trading strategy implementation in the main part of the paper. The batch size also depends on the algorithm that is used. Gradient based algorithms are relatively robust

¹³For further details we also refer to Haykin (2009).

¹⁴The following description is based on Goodfellow et al. (2016).

and thus able to make use of smaller batch sizes while methods additionally relying on the Hessian matrix typically require substantially larger batch sizes in order to provide appropriate estimates of the second moments.

Like the gradient descent algorithm, the SGD algorithm is an iterative method for minimizing an error functional. Therefore, the available data have to be processed multiple times in order to reach at least a local minimum. One sweep over the whole dataset is referred to as an epoch. In total, training a network model based on N training examples, a batch-size of m , and n_e epochs requires $\lceil \frac{N}{m} \rceil \cdot n_e$ iterations. While generally errors are decreasing over epochs, one has to be aware of the fact that too many runs through the same dataset might lead to overfitting.¹⁵ This is discussed later in this section. Another limitation to the number of epochs is the computational capacity as the runtime scales linearly with the number of epochs.¹⁶

An important parameter for mini-batch learning is the learning rate that determines the extent to which single weights are adjusted in each iteration depending on the gradient. In practice, one typically relies on a decreasing learning rate over time to reduce the impact of noise from randomly selected mini-batches. This noise does not vanish even when the algorithm approaches a (local) minimum in contrast to the gradient of the cost function which should reach zero in the minimum. As a consequence, one obtains a low signal to noise ratio (cf. [Goodfellow et al., 2016](#)). Convergence in the SGD algorithm is ensured when choosing a learning rate λ_k with k denoting the corresponding iteration step such that

$$\lim_{k \rightarrow \infty} \lambda_k = 0, \quad \sum_{k=1}^{\infty} \lambda_k = \infty, \quad \text{and} \quad \sum_{k=1}^{\infty} \lambda_k^2 < \infty.$$

A possible choice fulfilling these conditions is $\lambda_k := \frac{1}{k}$ (see also [Hastie et al., 2017](#)). However, the cost function is often highly sensitive to some directions in the parameter space while it is insensitive to others. As a consequence, such a simplistic specification of the learning rate might not lead to optimal results. Instead, it might be favorable to use a separate learning rate for each parameter that is automatically adapted throughout the learning process, that is, to use algorithms

¹⁵That is, while the MSE decreases in the *training set* it might start to increase at some point in the *test set*.

¹⁶For large datasets some machine learning applications only use each training example once or even use the whole sample incompletely (see [Goodfellow et al., 2016](#)).

with adaptive learning rates.¹⁷

An early example of this kind of algorithms is the delta-bar-delta algorithm by [Jacobs \(1988\)](#). This heuristic approach is based on the simple rule of increasing the learning rate whenever the partial derivative of the loss with respect to the current parameters does not change its sign, while decreasing the learning rate if it does. Unfortunately, this method can only be applied to batch optimization. In recent years, algorithms with adaptive learning rates that are especially suited for mini-batch learning have been introduced. These include among others the AdaGrad algorithm by [Duchi et al. \(2011\)](#) and the RMSProp algorithm due to [Tieleman and Hinton \(2012\)](#). A more recent representative of this class of algorithms is “adaptive moments estimation” by [Kingma and Ba \(2014\)](#) commonly known as *adam*. Adam uses both averages of the gradients and the second moments of the gradients and additionally includes a bias correction to both the first- and second-order moments (see [Goodfellow et al., 2016](#)). It is quite robust to the choice of the hyperparameters and represents a popular choice in financial applications.¹⁸

As mentioned previously, overfitting is a serious problem especially in training large neural networks that can rely on tens or hundreds of thousands of parameters.¹⁹ Loosely speaking, a model is overfitted if it corresponds too closely to the dataset it was trained on and loses the ability to generalize well to similar data it has not seen before. This can happen, for example, when complex models like neural networks end up memorizing the training data and find features that are present in the training data but not in the true underlying data generating process. There are several possible remedies to this problem like early-stopping method where training is stopped as soon as the validation error (on the test set) starts to increase ([Haykin, 2009](#)). Another popular approach is *dropout* by [Srivastava et al. \(2014\)](#). It is a quite powerful regularizing method that can be applied to a broad family of models and is at the same time computationally inexpensive.²⁰ As these approaches are employed in the empirical study, we will present them in more detail.

¹⁷For the following description of algorithms with adaptive learning rate we rely on [Goodfellow et al. \(2016\)](#).

¹⁸See, e.g., the papers by [Sirignano and Cont \(2019\)](#); [Rothfuss et al. \(2019\)](#); [Chen et al. \(2023\)](#); [Elend et al. \(2020\)](#); [Gu et al. \(2020\)](#). However, there is currently no consensus on the optimal choice of the optimization algorithm. For example, [Schaul et al. \(2014\)](#) study a range of adaptive learning rate algorithms and find no single best one leaving the choice of the optimization algorithm largely to the user or researcher and its familiarity with the algorithm (cf. [Goodfellow et al., 2016](#)).

¹⁹In the empirical application of the paper, we train relatively small neural networks with one to three hidden layers. However, the number of parameters quickly reaches five-digit numbers especially for the LSTM model and models derived from it.

²⁰Dropout is frequently used in the recent literature, see, e.g., [Heaton et al. \(2017\)](#); [Fischer and Krauss \(2018\)](#); [Chen et al. \(2023\)](#); [Elend et al. \(2020\)](#); [Bianchi et al. \(2021\)](#).

Dropout can be thought of as a version of bagging that is applicable to neural networks.²¹ More detailed, dropout trains an ensemble of subnetworks of a given neural network that is obtained by randomly removing non-output units from the network. This can typically be achieved by multiplying the outputs of a particular neuron with zero. Therefore, each time a new mini-batch is fed into the network, a fraction of input and hidden units is sampled independently from each other and their outputs are set to zero. The fraction of neurons to be sampled is a hyperparameter that has to be specified at the start of training. Calculating the gradients on the error functional and backpropagating those through the network then continues as usual (Goodfellow et al., 2016).

Another widely used regularization technique is early stopping. The idea is to monitor performance on a separate validation set during training and halt the optimization procedure once improvements cease, even if training error continues to decrease (Bengio, 2012). This prevents the network from overfitting noise in the training data and typically improves out-of-sample generalization. Early stopping is particularly attractive because it is simple to implement, computationally inexpensive, and often complements other approaches like dropout, which can shorten the time needed until the optimal validation performance is reached. In empirical finance, early stopping is frequently used to halt training once validation performance stops improving (Gu et al., 2020; Jiang et al., 2023).

Further difficulties concerning the training of neural networks like ill-conditioning, saddle points, cliffs, exploding gradients, etc. are discussed in Goodfellow et al. (2016) and Hastie et al. (2017) and are not covered here for brevity.

IA.1.5 Hyperparameter optimization

Beyond the choice of the optimization algorithm, the effectiveness of network training strongly depends on the specification of hyperparameters that govern model capacity, regularization, and convergence behavior. In the paper, we therefore conduct a systematic search for suitable hyperparameter configurations, while at the same time deliberately leaving certain structural design choices (e.g., the length of the look-back window or the number of hidden layers) outside the optimization routine, since the effects of changes in these parameters are analyzed explicitly in the paper.

²¹Loosely speaking, bagging is about fitting the same model multiple times to randomly selected subsamples and averaging the model outcomes. This approach is usually not directly applicable to neural networks as training and evaluating neural networks is computationally very expensive.

Hyperparameters refer to model characteristics that are not directly learned during training but need to be specified a priori. They determine, for example, the capacity of the network, the degree of regularization, or the learning dynamics during optimization. In the paper, we systematically optimize various hyperparameters, namely the number of hidden nodes, dropout rate, activation functions of the hidden and final layer, batch size, number of epochs, and the learning rate. Other design choices, such as the number of hidden layers, look-back window length, feature set, or model architecture (ANN vs. LSTM variants), are varied separately in the main analysis to study their effects on the model performance.

Given the considerable computational costs of repeatedly fitting neural networks, we design the hyperparameter search to balance efficiency with robustness. To this end, we employ a universal feedforward neural network with a single hidden layer as the baseline model, since this architecture is fast to train, allows extensive exploration of the hyperparameter space, and has proven robust for comparative purposes. To further accelerate the procedure and thereby increase the number of trials for exploring suitable hyperparameters, the network is trained with early stopping (see Section [IA.1.4](#) for details) and the optimization is conducted on a randomly selected subset comprising one third of the training and validation data. The data are split chronologically into training (up to December 2015), validation (2016-2017), and test set (from 2018 onward). The validation period is used to guide hyperparameter selection, while the out-of-sample test set is then used in the main analyses for the final evaluation of predictive performance. The search space comprises the number of hidden nodes (8-128), dropout rate (0.0-0.5), hidden and output activation functions (ReLU, Sigmoid, Linear), batch size (16-1024), number of epochs (100-500), and learning rate (10^{-5} - 10^{-2}). Optimization is performed with Optuna ([Akiba et al., 2019](#)) using the Tree-structured Parzen Estimator (TPE) sampler ([Bergstra et al., 2011](#)), complemented by early stopping of underperforming trials (MedianPruner). In total, we conducted 100 trials.

The best-performing configuration was characterized by the following set of hyperparameters:

hidden nodes = 64, dropout = 0.299, hidden activation = ReLU, output activation = ReLU,

batch size = 64, epochs = 300, learning rate = 1.78×10^{-4} .

Across trials, performance proves robust, with many nearby parameter combinations yielding almost identical validation errors, suggesting low sensitivity to minor hyperparameter variations. Clear performance differences are nonetheless observed with respect to certain parameters: learning rates outside the range 2.5×10^{-5} - 2×10^{-4} lead to worse convergence, and dropout rates outside 0.20-0.35 either fail to regularize or cause underfitting. ReLU consistently outperformed Sigmoid as hidden activation, while model capacity was optimal around 64 nodes. Other parameters, such as output activation (Linear vs. ReLU), batch size, or training epochs, exhibit only secondary influence. Taken together, these findings confirm that while hyperparameter optimization improves accuracy, results are quite stable across a wider range of parameters, such that the central empirical conclusions of the study are not materially affected by specific tuning choices.

IA.2 Traditional time series models

To provide a more realistic assessment of the forecasting accuracy of the neural networks described above, we also include some “traditional” time series models in the study. However, as the primary focus is on the network models we only include some very simple and parsimonious models, namely the exponentially weighted moving average (EWMA) model along with the autoregressive integrated moving average (ARIMA) and the vector autoregressive moving average (VARMA) model.

IA.2.1 EWMA model

Linear forecasting procedures are methods that apply weights to present and previous observations of a time series in order to derive estimates of unknown future observations. One of the most commonly used representative of this class of models is the EWMA model where the weights are exponentially discounted into the past ([Haywood and Wilson, 1997](#)). The model in its earliest form dates back to Poisson ([Oppenheim and Schafer, 1975](#)) and was first explicitly covered by [Brown \(1956\)](#) and [Holt \(1957\)](#) and later shown to be optimal for simple integrated moving average processes (cf. [Muth, 1960](#)). One of the reasons for its popularity is its very simple structure, especially

when expressing it in terms of the recursive equation

$$s_t := \lambda x_t + (1 - \lambda)s_{t-1},$$

where s_t and x_t , $t = 1, 2, \dots$ denote the EWMA at time t and observations from the time series, respectively, s_1 is defined as x_1 , and $0 < \lambda < 1$ is a weighting factor. In fact, the model is especially popular in estimating and forecasting conditional volatility (McNeil et al., 2015). In particular, it was chosen as the basis for one of the first value at risk (VaR) models in 1994 by J.P. Morgan (the *RiskMetrics* model). In volatility forecasting, the EWMA model can also be understood as a special case of the GARCH(1,1)-model (Bollerslev, 2008).

There is no standard procedure for finding an optimal weighting factor λ . Sometimes λ is chosen such that the sum of squared errors between predicted and actual values is reduced. (Haywood and Wilson, 1997). However, in some applications the exponential weighting factor is specified ex-ante and not calibrated to the data.²² As the most attractive feature of the EWMA model is its simplicity we also specify the weights ex-ante and consider values of 0.95 and 0.99 in the empirical analysis.²³

IA.2.2 ARIMA model

A standard model in classical time series analysis is the ARIMA model.²⁴ While the EWMA model is primarily focused on modeling trends and seasonality, ARIMA models describe the auto-correlations in the data. ARIMA models form a general class of models for forecasting time series. They consist of an autoregressive term formed by lags of the stationarized series and an moving average term based on lags of the forecasting errors. A time series that is not mean stationary can be differenced to become mean stationary. It is then referred to as an integrated version of the original time series.

For a more formal introduction of the ARIMA model we start with the non-integrated

²²This is, e.g., the case in the RiskMetrics model where λ is set to 0.94.

²³The primary focus of this paper is on the neural networks. Therefore, we refrain from engaging in further optimizations of the EWMA model. For the same reason, we restrict ourselves to the univariate case although there are multivariate versions of the EWMA model, in particular for one-step ahead forecasts of conditional covariance matrices, see McNeil et al. (2015).

²⁴A first comprehensive treatment of ARIMA models is due to Box and Jenkins (1970). The book stimulated further research in time series modeling (cf. McNeil et al., 2015).

ARMA(p, q) model where p refers to the number of autoregressive terms and q denotes the number of moving average terms. More explicitly, for a time series (x_t) the ARMA(p, q) model is given by

$$x_t = \sum_{j=1}^p \alpha_j x_{t-j} + \sum_{j=1}^q \beta_j \epsilon_{t-j} + \epsilon_t,$$

where $\alpha_j, j = 1, \dots, p$ and $\beta_j, j = 1, \dots, q$ are parameters and $\epsilon_\tau, \tau = t - q, \dots, t$ are (previous) forecasting errors. Using the lag operator L where $Lx_t := x_{t-1}$ the equation can be transformed into $(1 - \sum_{j=1}^p \alpha_j L^j)x_t = (1 + \sum_{j=1}^q \beta_j L^j)\epsilon_t$.

The ARIMA(p, d, q) model arises naturally from the ARMA model by first differencing the original time series d times before fitting an ARMA model to it. This is done in order to achieve mean stationarity which is a prerequisite for applying an ARMA model to it. The ARIMA(p, d, q) model is thus characterized by the following equation:

$$\left(1 - \sum_{j=1}^p \alpha_j L^j\right)(1 - L)^d x_t = \left(1 + \sum_{j=1}^q \beta_j L^j\right)\epsilon_t.$$

There exist various criteria for choosing the order p, d , and q of the model, some relying on the sample or the partial autocorrelation function while others are based on information criteria such as the AIC or BIC. However, as with the EWMA model, for simplicity we choose a parsimonious specification and rely on an ARIMA(1,1,1) model in our empirical application. The model parameters are then fitted via maximum likelihood estimation. For more details on the model and the estimation process we refer to the book by [Box and Jenkins \(1970\)](#).

IA.2.3 VARMA model

ARIMA models can only be applied to univariate time series. Therefore, we also consider the VARMA model to allow for the inclusion of past stock returns into the forecasting of option prices. We start by introducing the vector autoregressive (VAR) model and subsequently extend it to the VARMA model.²⁵

The VAR model relies linearly on past lags of the time series of interest as well as on other

²⁵The description of the VARMA model is based on [Lütkepohl \(1991\)](#).

variables to derive forecasts. More formally, the VAR(p) model is given by

$$y_t = c + \sum_{j=1}^p A_j y_{t-j} + \epsilon_t,$$

where (y_t) is a vector-valued time series, c is a vector of constants, $A_1, \dots, A_p$ denote time invariant parameter matrices, and $\epsilon_\tau, \tau = t-p, \dots, t$ are vectors of error terms. The error terms are assumed to be white noise. More exactly, one requires

$$E(\epsilon_t) = 0, \quad E(\epsilon_t \epsilon_t^T) = \Omega, \quad \text{and} \quad E(\epsilon_t \epsilon_{t-k}^T) = 0 \text{ for all } k > 0,$$

that is, the errors are assumed to have zero mean, a time invariant covariance matrix, and most noteworthy no serial correlation between individual error terms. By relaxing the last assumption and including a finite order moving average process into the VAR model we obtain the more flexible VARMA model.

The VARMA(p,q) model is given by

$$y_t = c + \sum_{j=1}^p A_j y_{t-j} + \sum_{j=1}^q M_j u_{t-j} + u_t,$$

where $A_j, j = 1, \dots, p$ and $M_j, j = 1, \dots, q$ denote time invariant parameter matrices, $u_\tau, \tau = t-q, \dots, t$ are vectors of error terms that are assumed to be *independent* white noise, and the rest of the notation is as above. Estimation of the parameters can be performed via maximum likelihood.

IA.3 Summary statistics of option samples

Table [IA.1](#) provides the mean, median, standard deviation, minimum, and maximum of the standardized ATM IVs of **719,157** call options between **January 2018 and August 2023**. Summary statistics are first calculated per IV series and than averaged across underlyings. We can thus interpret the values as summary statistics of an average IV series. The mean of the the average IV series is **49.40 %** with an average median of **44.85 %**. There is substantial variation between the minimum of **21.89 %** and the maximum of **147.43 %**. We further provide summary statistics

Table IA.1: Summary statistics of standardized call option ATM IVs

This table provides summary statistics for the standardized ATM call option IVs between **January 2018 - August 2023**. Over this test set, the forecasting accuracy of the time series models is evaluated. We provide the mean, median, standard deviation (sd), minimum (min), and maximum (max) of all observations (*overall*). Additionally, we present summary statistics for eight macroeconomic sectors according to the first digit of the MorningStar industry classification along with the percentage of observations in a particular industry (*percent. of obs.*). The data include **719,157** observations over **295** different timepoints. The summary statistics were first calculated for the IVs of each underlying separately and then averaged over all underlyings. As a consequence, the results can be interpreted as summary statistics of an average IV series.

	percent. of obs.	mean	median	sd	min	max
overall	100.00%	49.40%	44.85%	20.26%	21.89%	147.43%
Basic materials	9.75%	59.75%	53.49%	25.46%	28.56%	183.10%
Consumer goods	7.98%	47.19%	43.49%	17.44%	22.07%	135.72%
Financial	34.04%	37.30%	33.25%	17.34%	14.65%	129.74%
Healthcare	7.74%	68.60%	62.97%	25.49%	29.97%	166.34%
Industrial goods	7.36%	45.43%	41.87%	16.67%	21.94%	132.36%
Services	15.56%	54.68%	50.26%	21.68%	25.36%	161.53%
Technology	13.64%	54.50%	49.81%	21.45%	24.07%	148.58%
Utilities	2.63%	38.70%	33.90%	17.72%	16.34%	123.37%

for eight macroeconomic sectors according to the MorningStar industry classification. There are notable differences between the sectors. In particular, the average standardized IVs range between **37.30 %** in the **financial** and **68.60 %** in the **healthcare sector**. Average IVs above **50 %** can also be observed in the technology (**54.50 %**), services (**54.68 %**), and **basic materials** sector (**59.75 %**). Interestingly, average IV in the financial sector is relatively low (**37.30 %**). When sorting the sectors with regard to the median IV, the ordering is mostly unchanged. This ordering is also roughly reflected by the standard deviation of the standardized IVs within the sectors (between **16.67 %** and **25.49 %**). For all industries we observe substantial variation between the minimum and maximum values.

Table [IA.2](#) provides summary statistics on buy-and-hold raw and delta-hedged returns as well as on IV, delta, gamma, and vega of our set of listed ATM call options within the time period **January 2018 until August 2023**. The statistics are first calculated for each underlying individually and subsequently averaged across all underlyings. The numbers can thus be interpreted as summary statistics of the average option series. The average weekly buy and hold return is **-21.55 %**. The

Table IA.2: Summary statistics of listed options

This table reports summary statistics for the sample of listed ATM call options considered in the trading strategy from **January 2018 to August 2023**. For each underlying security, we compute weekly buy-and-hold raw and delta-hedged option returns as well as the main option characteristics (implied volatility, delta, gamma, and vega). The statistics (mean, median, standard deviation (sd), minimum (min), and maximum (max)) are first calculated at the individual underlying level and then averaged across all underlyings. Hence, the reported numbers can be interpreted as average summary statistics across option series.

	mean	median	sd	min	max
raw returns	-21.55%	-70.49%	118.85%	-92.62%	322.57%
delta hedged returns	-1.24%	-1.45%	3.67%	-7.26%	6.73%
IV	60.95%	56.33%	24.68%	36.80%	123.10%
delta	0.53	0.53	0.10	0.38	0.69
gamma	0.40	0.39	0.15	0.23	0.69
vega	2.11	2.06	0.65	1.16	3.33

returns are highly skewed to the right with a median that is extremely negative (**-70.49 %**). The returns vary between **-92.62 %** and **322.57 %** with a standard deviation of **118.85 %**. These large variations are not surprising as (under the assumption of no arbitrage) options can be seen as leveraged positions in the underlying stocks (see [Buraschi and Jackwerth, 2001](#)). As a consequence, raw option returns are highly dependent on movements in the underlying stock. Therefore, we also provide summary statistics on delta-hedged returns that have the lowest exposure to the underlying stock (see, e.g., [Bakshi and Kapadia, 2003](#); [Goyal and Saretto, 2009](#); [Cao and Han, 2013](#); [Zhan et al., 2022](#); [Bernales and Guidolin, 2014](#); [Hu and Jacobs, 2020](#)). By means of delta-hedging we can reduce variations in the returns substantially. For example, the standard deviation of the average option return series is reduced to **3.67 %**. The average delta-hedged returns are **-1.24 %** with a median of **-1.45 %**. Although standard deviation is largely reduced, there is still substantial variation in the returns with a minimum of **-7.26 %** and a maximum of **6.73 %**.

IA.4 Additional tables

Table IA.3: Summary statistics for the squared errors of the universal ANN model over different training periods per macroeconomic sector

This table complements Table VI by reporting the mean squared errors (MSE) and out-of-sample R^2 for the universal one-layer artificial neural network (ANN) model trained over time periods of increasing length, where the year in parentheses indicates the end of the validation window. For example, the ANN (2009) model is trained on data up to 2007 with validation from 2008–2009, whereas the ANN (2017) model represents the baseline specification used throughout the analysis, trained on data from 1996–2015 with validation in 2016–2017. The ANN is employed here because all universal neural network models deliver very similar forecasting performance, while the ANN can be fitted much faster than LSTM-based architectures, making it particularly suitable for additional robustness analyses. All models are evaluated on the same out-of-sample period from January 2018 to August 2023. The MSE measures the mean squared deviation between forecasted and realized one-week-ahead ATM IVs, while the out-of-sample R^2 is computed using the test-period mean as benchmark. Results are shown for the overall sample and for eight macroeconomic sectors based on the MorningStar industry classification. Forecasting performance remains remarkably stable across training windows, indicating strong temporal robustness of the universal ANN model.

	MSE				R^2			
	ANN (2009)	ANN (2012)	ANN (2014)	ANN (2017)	ANN (2009)	ANN (2012)	ANN (2014)	ANN (2017)
Overall	2.73%	2.72%	2.68%	2.68%	70.55%	70.68%	71.07%	71.07%
Basic materials	3.53%	3.52%	3.49%	3.51%	66.89%	66.97%	67.29%	67.12%
Consumer goods	1.83%	1.83%	1.81%	1.81%	73.05%	73.12%	73.41%	73.42%
Financial	2.54%	2.53%	2.49%	2.50%	65.74%	65.82%	66.35%	66.29%
Healthcare	4.23%	4.20%	4.13%	4.07%	73.12%	73.36%	73.81%	74.14%
Industrial goods	1.70%	1.70%	1.68%	1.69%	69.96%	69.99%	70.42%	70.24%
Services	2.59%	2.58%	2.55%	2.57%	71.40%	71.52%	71.82%	71.58%
Technology	3.21%	3.19%	3.15%	3.12%	65.58%	65.83%	66.27%	66.55%
Utilities	2.45%	2.41%	2.39%	2.37%	63.72%	64.22%	64.51%	64.83%

Table IA.4: Summary statistics of squared errors for the universal ANN model over different training periods

This table reports results similar to Table IA.3 but for the overall sample instead of separate macroeconomic sectors. It provides summary statistics for the universal one-layer artificial neural network (ANN) model trained over time periods of increasing length, where the year in parentheses indicates the end of the validation window. For example, the ANN (2009) model is trained on data from 1996 to 2007 with validation from 2008–2009, whereas the ANN (2017) model represents the baseline specification used throughout the analysis, trained on data from 1996–2015 with validation in 2016–2017. All models are evaluated on the same test set from 2018–2023. The table reports the mean, median, standard deviation (sd), minimum (min), and maximum (max) of the squared forecast errors, as well as the corresponding out-of-sample R^2 . Forecasting performance remains remarkably stable across training windows, confirming the temporal robustness of the universal ANN model.

	mean	median	sd	min	max	R^2
ANN (2009)	2.73%	0.14%	16.05%	0.00%	696.45%	70.55%
ANN (2012)	2.72%	0.14%	15.99%	0.00%	709.46%	70.68%
ANN (2014)	2.68%	0.14%	15.83%	0.00%	699.81%	71.07%
ANN (2017)	2.68%	0.15%	15.91%	0.00%	699.51%	71.07%

Table IA.5: Summary statistics of squared errors for all models when excluding the COVID-19 years 2020-2021

This table complements Table IX by reporting summary statistics on the squared errors between one-week-ahead forecasts of standardized ATM IVs and the actual values for various models in different specifications. All models are trained on data from January 1996 to December 2015 and evaluated out-of-sample over the period from January 2018 to August 2023, but *excluding* the COVID-19 years 2020–2021 from the calculation of forecast errors. We report the mean, median, standard deviation (sd), minimum (min), and maximum (max) of squared errors, complemented by the out-of-sample R^2 . Models are sorted in ascending order by their mean squared error. The first column (group) corresponds to different model specifications, where Baseline refers to Section 3.2, Short Term to Section 3.3, Returns to Section 3.4, Deep to Section 3.5, Universal to Section 3.6, and Traditional to Section 3.7. The relative ranking of models remains largely unchanged compared to the full-sample results in Table IX, although average MSEs across models are somewhat lower when excluding the volatile COVID-19 years from the evaluation.

		mean	median	sd	min	max	R^2
Universal	Conv. LSTM	2.51%	0.11%	15.72%	0.00%	680.36%	70.78%
Universal	LSTM1	2.51%	0.12%	15.62%	0.00%	695.43%	70.70%
Universal	Bidir. LSTM	2.52%	0.11%	15.46%	0.00%	684.64%	70.65%
Universal	LSTM3	2.52%	0.18%	15.52%	0.00%	668.49%	70.57%
Universal	ANN1	2.53%	0.13%	15.48%	0.00%	699.51%	70.56%
Universal	ANN3	2.67%	0.29%	16.36%	0.00%	703.05%	68.86%
Traditional	ARIMA	2.75%	0.12%	16.38%	0.00%	694.79%	67.96%
Traditional	VARMA	2.83%	0.14%	16.68%	0.00%	681.75%	67.35%
Traditional	EWMA (0.95)	3.42%	0.10%	20.44%	0.00%	739.30%	60.13%
Traditional	EWMA (0.99)	3.53%	0.10%	21.13%	0.00%	744.21%	58.87%
Short term	Bidir. LSTM	3.63%	0.44%	19.33%	0.00%	796.72%	57.67%
Short term	LSTM	3.75%	0.45%	19.92%	0.00%	826.42%	56.31%
Baseline	LSTM	3.76%	0.35%	20.37%	0.00%	896.23%	56.10%
Short term	Conv. LSTM	3.76%	0.50%	19.72%	0.00%	880.15%	56.13%
Short term	ANN	3.81%	0.35%	20.96%	0.00%	896.23%	55.63%
Baseline	Conv. LSTM	3.82%	0.34%	20.90%	0.00%	886.15%	55.44%
Baseline	Bidir. LSTM	3.90%	0.39%	20.48%	0.00%	890.09%	54.52%
Deep	LSTM2	4.07%	0.38%	21.90%	0.00%	843.81%	52.58%
Baseline	ANN	4.28%	0.29%	21.46%	0.00%	888.39%	50.05%
Returns	Conv. LSTM	4.32%	0.42%	21.98%	0.00%	849.73%	49.34%
Returns	LSTM	4.38%	0.42%	21.64%	0.00%	885.06%	48.63%
Returns	Bidir. LSTM	4.46%	0.43%	22.40%	0.00%	862.95%	47.69%
Deep	LSTM3	4.49%	0.39%	23.24%	0.00%	836.52%	47.53%
Deep	ANN2	4.63%	0.38%	22.59%	0.00%	895.81%	45.98%
Deep	ANN3	5.85%	0.55%	26.22%	0.00%	883.69%	31.79%
Returns	ANN	12.27%	4.41%	33.12%	0.00%	898.35%	-43.84%

Table IA.6: Weekly returns from a raw option and a delta-hedged trading strategy based on the optimized universal LSTM model (1 hidden layer, 4 weeks lookback window)

This table provides similar results to Table XI but for the optimized universal LSTM model with one hidden layer and a 4-week lookback window instead of the convolutional universal LSTM (1 layer, 50-week lookback) used in the main analysis. The optimized model combines the specification choices that individually improved forecasting performance, namely a shorter lookback horizon, fewer layers, and the universal training design. We report the mean, median, standard deviation (sd), minimum (min), and maximum (max) of weekly returns separately for the long, short, and long-short portfolios. The returns are calculated based on value-weighted (by dollar open interest) portfolios of ATM call contracts, sorted by predicted changes in IVs for both the raw option strategy and the delta-hedged strategy between January 2018 and August 2023. Weekly Sharpe ratios (SR) are reported for the long-short portfolios, with *, **, and *** denoting significance at the 10 %, 5 %, and 1 % levels, respectively.

All models are trained on standardized ATM IVs from January 1996 to December 2015, and the resulting forecasts for the out-of-sample test period from January 2018 to August 2023 form the basis of the trading strategies. Compared to the baseline convolutional LSTM, the optimized universal LSTM achieves higher average returns and Sharpe ratios for both raw and delta-hedged strategies, confirming that shorter lookback horizons and reduced model complexity enhance trading performance.

		mean	median	sd	min	max	SR
raw	long	1.55%	-16.94%	79.21%	-99.75%	452.79%	
	short	-9.01%	-26.94%	73.60%	-98.44%	361.02%	
	long-short	10.57%	10.92%	76.10%	-232.40%	273.14%	0.139***
delta	long	-0.25%	-0.33%	1.14%	-4.93%	6.15%	
	short	-0.89%	-1.14%	3.53%	-26.81%	10.33%	
	long-short	0.65%	0.71%	3.36%	-12.94%	25.19%	0.193***

Table IA.7: Weekly returns when considering transaction costs for the optimized universal LSTM model (1 hidden layer, 4 weeks lookback window)

This table reports results similar to Table XIII but for the optimized universal LSTM model with one hidden layer and a 4-week lookback window instead of the convolutional LSTM (1 layer, 50-week lookback) used in the main analysis. The optimized model combines the specification choices that individually improved forecasting performance, namely a shorter lookback horizon, fewer layers, and the universal training design. The table provides summary statistics on weekly returns from raw option and delta-hedged long-short trading strategies, but after considering transaction costs. The trading strategies are based on IV forecasts for ATM call contracts from the optimized universal LSTM model trained on pooled data from 1996 to 2015, where portfolios are formed by sorting on predicted changes in IVs over the test period from January 2018 until August 2023 value-weighted by dollar open interest. We consider ratios of effective to quoted bid-ask spreads of 25 % and 50 %. *, **, and *** denote weekly Sharpe ratios that are greater than zero at the 10 %, 5 %, and 1 % significance levels, respectively.

			mean	median	sd	min	max	SR
raw	trans.	Cost						
		25%	7.73%	8.03%	76.19%	-242.51%	268.29%	0.101**
		50%	4.73%	5.29%	76.41%	-252.59%	261.80%	0.062
delta		25%	0.46%	0.55%	3.29%	-13.45%	23.20%	0.141***
		50%	0.26%	0.39%	3.26%	-13.91%	21.74%	0.080*

Table IA.8: Equally-weighted weekly returns from the trading strategies based on different models

This table provides results similar to Table XIV, but for equally weighted raw option and delta-hedged portfolios, as opposed to the value-weighted portfolios (by dollar open interest) used in the main analysis. The table summarizes information on the weekly returns from strategies based on IV forecasts by various models. The returns are calculated based on equally-weighted portfolios of ATM call contracts, sorted by predicted changes in IVs for both the raw option strategy as well as the delta-hedged strategy between January 2018 and August 2023. We report the mean, standard deviation (sd), and Sharpe ratio (SR) where *, **, and *** denote Sharpe ratios that are greater than zero at the 10 %, 5 %, and 1 % significance level, respectively. The table also includes the optimized universal LSTM model with one hidden layer and a 4-week lookback window, which combines in a greedy manner the individual specification choices that improved forecasting performance. The models are ordered in descending order based on the Sharpe ratios from the delta-hedged trading strategy.

Compared to the value-weighted portfolios, the overall cross-model ranking remains broadly consistent, with rank correlations of 0.75 for raw option returns and 0.50 for delta-hedged returns, although some individual strategies show notable deviations. However, fewer specifications achieve statistically significant Sharpe ratios for raw returns relative to the value-weighted case.

		Raw returns			Delta hedged returns		
		mean	sd	SR	mean	sd	SR
Universal	LSTM (4 weeks)	3.34%	47.59%	0.070	0.37%	1.37%	0.272***
Universal	ANN3	7.78%	66.43%	0.117**	0.50%	1.90%	0.262***
Short term	ANN	5.16%	51.49%	0.100**	0.32%	1.50%	0.212***
Short term	Conv. LSTM	6.49%	56.77%	0.114**	0.26%	1.53%	0.172***
Short term	Bidir. LSTM	5.49%	56.26%	0.098**	0.23%	1.49%	0.154***
Short term	LSTM	4.85%	58.52%	0.083*	0.23%	1.60%	0.145***
Universal	LSTM3	4.67%	61.88%	0.076*	0.25%	1.77%	0.142***
Deep	LSTM2	5.33%	50.19%	0.106**	0.20%	1.41%	0.141***
Baseline	Bidir. LSTM	4.36%	47.84%	0.091*	0.19%	1.40%	0.138**
Deep	LSTM3	3.24%	53.09%	0.061	0.20%	1.42%	0.138***
Universal	LSTM	1.73%	48.02%	0.036	0.19%	1.41%	0.137**
Baseline	LSTM	4.83%	48.25%	0.100**	0.17%	1.32%	0.130**
Baseline	Conv. LSTM	3.94%	47.41%	0.083*	0.18%	1.44%	0.127**
Universal	ANN	2.36%	50.21%	0.047	0.18%	1.49%	0.119**
Returns	LSTM	4.16%	42.47%	0.098**	0.13%	1.21%	0.111**
Deep	ANN3	1.07%	41.51%	0.026	0.13%	1.22%	0.105**
Returns	Bidir. LSTM	4.92%	39.43%	0.125**	0.12%	1.18%	0.104**
Returns	Conv. LSTM	3.72%	44.49%	0.084*	0.13%	1.38%	0.092*
Traditional	EWMA (0.99)	-0.06%	43.47%	-0.001	0.08%	0.99%	0.084*
Universal	Bidir. LSTM	-0.83%	42.86%	-0.019	0.09%	1.22%	0.076
Deep	ANN2	-0.65%	38.53%	-0.017	0.08%	1.17%	0.072
Traditional	EWMA (0.95)	0.07%	43.68%	0.002	0.07%	0.98%	0.071
Traditional	VARMA	0.80%	39.87%	0.020	0.05%	1.08%	0.049
Traditional	ARIMA	-1.42%	35.15%	-0.040	0.05%	1.13%	0.047
Baseline	ANN	-1.72%	35.75%	-0.048	0.03%	1.02%	0.032
Universal	Conv. LSTM	-0.79%	44.02%	-0.018	0.04%	1.28%	0.032
Returns	ANN	-1.35%	27.07%	-0.050	-0.10%	0.70%	-0.149

Table IA.9: Weekly returns from the trading strategies when including options on dividend paying stocks

This table provides summary statistics on *weekly* returns from a raw option and a delta-hedged trading strategy based on IV forecasts by various models but when *not* excluding options on dividend paying stocks. The returns are calculated based on value-weighted (by dollar open interest) portfolios of ATM call contracts, sorted by predicted changes in IVs for both the option strategy as well as the delta-hedged strategy between January 2018 and August 2023. We report the weekly mean, standard deviation (sd), and Sharpe ratio (SR) where *, **, and *** denote Sharpe ratios that are greater than zero at the 10, 5, and 1 percent significance level, respectively. To provide a more complete picture, we include the returns derived from various other time series models additional to the universal convolutional LSTM from our main analysis. The models are ordered in descending order based on the Sharpe ratios from the delta-hedged trading strategy. The table also includes the optimized universal LSTM model with one hidden layer and a 4-week lookback window, which combines in a greedy manner the individual specification choices that improved forecasting performance in the main analysis. The results are in line with our previous findings.

		Raw returns			Delta hedged returns		
		mean	sd	SR	mean	sd	SR
Universal	LSTM (4 weeks)	10.35%	74.16%	0.140***	0.64%	3.30%	0.195***
Universal	LSTM	13.10%	83.60%	0.157***	0.48%	3.14%	0.153***
Universal	ANN3	20.10%	92.66%	0.217***	0.55%	3.84%	0.143***
Short term	ANN	12.10%	103.83%	0.117**	0.44%	3.46%	0.127**
Universal	Conv. LSTM	8.39%	80.41%	0.104**	0.33%	2.76%	0.120**
Universal	ANN	12.08%	90.35%	0.134**	0.40%	3.45%	0.115**
Traditional	ARIMA	5.01%	88.41%	0.057	0.38%	3.58%	0.107**
Universal	LSTM3	9.55%	96.64%	0.099**	0.41%	3.84%	0.106**
Short term	Conv. LSTM	14.56%	120.60%	0.121**	0.35%	3.63%	0.097**
Deep	LSTM2	17.72%	88.95%	0.199***	0.35%	3.60%	0.097**
Universal	Bidir. LSTM	10.32%	80.51%	0.128**	0.28%	3.00%	0.092*
Traditional	EWMA (0.95)	2.63%	81.97%	0.032	0.24%	2.68%	0.091*
Traditional	EWMA (0.99)	2.62%	82.77%	0.032	0.25%	2.76%	0.091*
Baseline	Conv. LSTM	15.36%	96.78%	0.159***	0.33%	3.74%	0.089*
Short term	Bidir. LSTM	9.68%	119.13%	0.081*	0.27%	3.79%	0.072
Short term	LSTM	9.20%	120.23%	0.077*	0.26%	3.65%	0.071
Returns	Bidir. LSTM	14.78%	87.44%	0.169***	0.17%	3.11%	0.055
Deep	LSTM3	11.19%	102.30%	0.109**	0.17%	3.35%	0.049
Returns	LSTM	9.82%	89.61%	0.110**	0.14%	2.81%	0.049
Traditional	VARMA	8.72%	90.09%	0.097**	0.10%	2.76%	0.035
Baseline	Bidir. LSTM	15.00%	113.42%	0.132**	0.11%	3.38%	0.031
Baseline	LSTM	10.62%	100.79%	0.105**	0.07%	3.47%	0.020
Returns	Conv. LSTM	11.23%	102.15%	0.110**	0.04%	3.00%	0.012
Deep	ANN3	6.51%	111.60%	0.058	0.00%	3.28%	0.000
Deep	ANN2	6.60%	93.74%	0.070	0.00%	3.40%	-0.001
Baseline	ANN	9.79%	91.28%	0.107**	-0.09%	2.91%	-0.031
Returns	ANN	-3.80%	71.72%	-0.053	-0.13%	1.46%	-0.087

Table IA.10: Weekly returns from the trading strategies based on different models when winsorized at the 5 % and 95 % level

This table summarizes information on the weekly returns from strategies based on IV forecasts by various models. The models are denoted similarly to Table IX. The returns are calculated based on a value-weighted (by dollar open interest) raw option strategy as well as a delta-hedged strategy between January 2018 and August 2023.

Compared to our main analysis, the option returns here are winsorized (separately for the long and short legs of the long-short portfolios at the 5 % and 95 % cross-sectional levels). This procedure reduces the impact of extreme returns. As expected, winsorization lowers the volatility of portfolio returns and generally increases the Sharpe ratios, particularly for the delta-hedged strategies. However, the relative ranking of model specifications remains largely unchanged suggesting that our main results are not driven by return outliers.

We also include the optimized universal LSTM model with one hidden layer and a 4-week lookback window instead of the convolutional LSTM (1 layer, 50-week lookback) used in the main analysis. The optimized model combines the specification choices that individually improved forecasting performance, namely a shorter lookback horizon, fewer layers, and the universal training design.

We report the weekly mean, standard deviation (sd), and Sharpe ratio (SR) where *, **, and *** denote Sharpe ratios that are greater than zero at the 10 %, 5 %, and 1 % significance level, respectively. The models are ordered in descending order based on the Sharpe ratios from the delta-hedged trading strategy.

		Raw returns			Delta hedged returns		
		mean	sd	SR	mean	sd	SR
Universal	LSTM (4 weeks)	10.66%	71.18%	0.150***	0.59%	2.45%	0.243***
Universal	ANN3	23.86%	88.31%	0.270***	0.66%	3.14%	0.212***
Universal	LSTM	11.39%	76.84%	0.148***	0.49%	2.41%	0.202***
Short term	ANN	14.41%	91.82%	0.157***	0.50%	2.76%	0.181***
Universal	ANN	13.46%	83.78%	0.161***	0.46%	2.60%	0.179***
Universal	Conv. LSTM	9.39%	73.07%	0.129**	0.37%	2.17%	0.172***
Deep	LSTM2	19.94%	85.78%	0.232***	0.46%	2.88%	0.161***
Baseline	Conv. LSTM	17.74%	89.37%	0.199***	0.48%	2.99%	0.160***
Short term	Conv. LSTM	14.34%	107.89%	0.133**	0.47%	2.97%	0.157***
Short term	LSTM	9.82%	106.27%	0.092*	0.46%	2.94%	0.156***
Universal	LSTM3	11.37%	89.28%	0.127**	0.48%	3.35%	0.143***
Short term	Bidir. LSTM	11.68%	106.27%	0.110**	0.41%	3.11%	0.132**
Universal	Bidir. LSTM	10.96%	74.65%	0.147***	0.29%	2.25%	0.131**
Returns	LSTM	11.03%	81.42%	0.135**	0.29%	2.43%	0.118**
Returns	Bidir. LSTM	15.78%	83.20%	0.190***	0.32%	2.82%	0.114**
Traditional	ARIMA	5.50%	73.38%	0.075*	0.23%	2.10%	0.108**
Traditional	VARMA	8.82%	78.83%	0.112**	0.22%	2.00%	0.108**
Deep	ANN3	8.81%	92.26%	0.096*	0.26%	2.66%	0.099**
Baseline	Bidir. LSTM	14.46%	98.22%	0.147***	0.28%	2.85%	0.098**
Baseline	LSTM	11.07%	90.56%	0.122**	0.26%	2.75%	0.096*
Traditional	EWMA (0.95)	2.76%	78.33%	0.035	0.18%	2.07%	0.089*
Deep	LSTM3	10.51%	94.73%	0.111**	0.25%	2.88%	0.088*
Returns	Conv. LSTM	11.47%	87.47%	0.131**	0.23%	2.65%	0.088*
Traditional	EWMA (0.99)	2.31%	80.44%	0.029	0.16%	2.14%	0.076
Deep	ANN2	7.76%	82.32%	0.094*	0.14%	2.88%	0.050
Baseline	ANN	10.79%	82.50%	0.131**	0.06%	2.23%	0.028
Returns	ANN	-4.65%	61.20%	-0.076	-0.13%	1.28%	-0.103

Table IA.11: Weekly returns in option subsamples

This table provides results similar to Table XI but for subsamples of options grouped by trading volume and relative bid–ask spread to examine whether strategy performance varies across distinct market segments. Subsamples are formed by sorting options each week into terciles based on trading volume as a proxy for investor attention and trading activity and on the relative bid–ask spread, defined as the spread divided by the midquote price, as a measure of liquidity. We report results for the lowest and highest terciles of each measure. The returns are calculated based on value-weighted (by dollar open interest) portfolios of ATM call contracts, sorted by predicted changes in IVs for both the raw option strategy and the delta-hedged strategy between January 2018 and August 2023. We report the mean, median, standard deviation (sd), minimum (min), and maximum (max) of weekly returns for the long, short, and long–short portfolios, as well as the Sharpe ratios (SR), where *, **, and *** denote significance at the 10 %, 5 %, and 1 % levels, respectively.

Profitability is generally higher for liquid and actively traded options. While strategies on low-volume or illiquid contracts yield weaker or statistically insignificant returns, Sharpe ratios are notably higher for high-volume and narrow-spread options, indicating that predictive signals are stronger in liquid, information-rich markets.

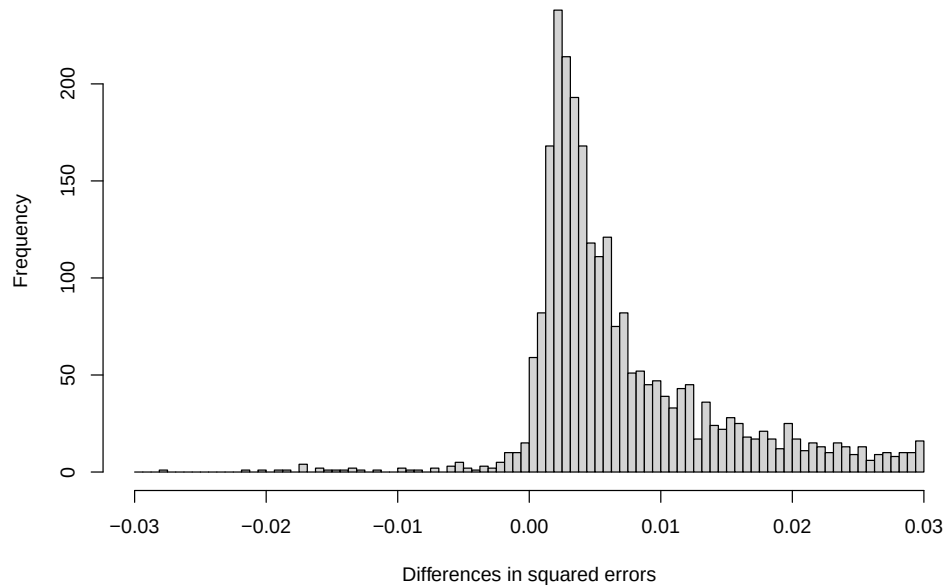
		Volume						
			mean	median	sd	min	max	SR
Low volume	raw	long	-2.25%	-19.03%	82.68%	-100.00%	303.92%	-0.038
		short	1.20%	-21.59%	90.16%	-100.00%	422.71%	
		long-short	-3.45%	-1.53%	91.48%	-431.76%	293.52%	
	delta	long	-0.78%	-0.63%	1.97%	-9.73%	8.27%	-0.050
		short	-0.64%	-0.73%	2.78%	-16.61%	13.36%	
		long-short	-0.15%	-0.08%	2.96%	-13.98%	11.55%	
			mean	median	sd	min	max	SR
High volume	raw	long	4.92%	-15.16%	91.27%	-100.00%	509.29%	0.146***
		short	-7.62%	-30.32%	84.53%	-100.00%	358.91%	
		long-short	12.54%	4.65%	85.70%	-268.46%	319.54%	
	delta	long	-0.12%	-0.23%	1.52%	-8.61%	5.00%	0.149***
		short	-0.59%	-0.64%	3.25%	-27.44%	13.36%	
		long-short	0.47%	0.55%	3.17%	-13.49%	27.04%	

		Liquidity						
			mean	median	sd	min	max	SR
Low liquidity	raw	long	-2.23%	-25.79%	86.25%	-100.00%	414.51%	0.016
		short	-3.89%	-31.46%	93.58%	-100.00%	594.27%	
		long-short	1.66%	1.61%	102.01%	-519.83%	416.17%	
	delta	long	-0.55%	-0.69%	2.35%	-11.36%	12.84%	0.111**
		short	-0.94%	-0.93%	3.33%	-19.92%	22.20%	
		long-short	0.38%	0.62%	3.46%	-23.13%	13.47%	
			mean	median	sd	min	max	SR
High liquidity	raw	long	10.89%	-13.37%	96.59%	-100.00%	514.17%	0.188***
		short	-5.33%	-21.32%	82.61%	-100.00%	351.69%	
		long-short	16.22%	7.64%	86.23%	-262.19%	321.49%	
	delta	long	-0.03%	-0.19%	1.64%	-7.67%	6.15%	0.176***
		short	-0.55%	-0.42%	2.81%	-25.20%	11.23%	
		long-short	0.52%	0.48%	2.96%	-12.77%	24.59%	

IA.5 Additional figures

Figure IA.3: Improvements in forecasting accuracy of universal over stock-specific LSTM models

This figure shows a histogram of the differences in MSEs between stock-specific LSTM models and a universal LSTM model. The stock-specific LSTM models are trained on the time series of ATM standardized IVs for each stock individually while the universal LSTM model is trained on pooled data across all underlyings. The MSEs are calculated on the stock level over the test period **January 2018 to August 2023**. Each observation in the histogram therefore corresponds to the difference between the MSEs of the universal and the stock-specific LSTM model for a *particular* stock. Differences greater than zero indicate an improved forecasting accuracy of the universal model compared to the stock-specific model. We observe lower forecasting errors for more than **96.8 %** of the stocks.



References

- ABIODUN, O. I., A. JANTAN, A. E. OMOLARA, K. V. DADA, N. A. MOHAMED, AND H. ARSHAD (2018): “State-of-the-art in artificial neural network applications: A survey,” *Heliyon*, 4, 1–41.
- AKIBA, T., S. SANO, T. YANASE, T. OHTA, AND M. KOYAMA (2019): “Optuna: A Next-generation Hyperparameter Optimization Framework,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
- BAKSHI, G. AND N. KAPADIA (2003): “Delta-Hedged Gains and the Negative Market Volatility Risk Premium,” *Review of Financial Studies*, 16, 527–566.
- BENGIO, Y. (2012): “Practical Recommendations for Gradient-Based Training of Deep Architectures,” *Neural Networks: Tricks of the Trade*, 7700, 437–478.
- BENGIO, Y., P. FRASCONI, AND P. SIMARD (1993): “The problem of learning long-term dependencies in recurrent networks,” in *International Conference on Neural Networks 1993*, 1183–1188.
- BENGIO, Y., P. SIMARD, AND P. FRASCONI (1994): “Learning long-term dependencies with gradient descent is difficult,” *IEEE transactions on neural networks*, 5, 157–166.
- BERGSTRA, J., R. BARDENET, Y. BENGIO, AND B. KÉGL (2011): “Algorithms for hyperparameter optimization,” in *Advances in neural information processing systems*, 2546–2554.
- BERNALES, A. AND M. GUIDOLIN (2014): “Can we forecast the implied volatility surface dynamics of equity options? Predictability and economic value tests,” *Journal of Banking & Finance*, 46, 326–342.
- BIANCHI, D., M. BÜCHNER, AND A. TAMONI (2021): “Bond Risk Premiums with Machine Learning,” *The Review of Financial Studies*, 34, 1046–1089.
- BOLLERSLEV, T. (2008): “Glossary to ARCH (GARCH),” *CREATES Research Papers*.
- BOX, G. AND G. M. JENKINS (1970): *Time Series Analysis: Forecasting and Control*, San Francisco: Holden-Day.
- BROWN, R. G. (1956): “Exponential Smoothing for Predicting Demand,” *Presented at the Tenth National Meeting of the Operations Research Society of America, San Francisco*, 1956.
- BURASCHI, A. AND J. JACKWERTH (2001): “The Price of a Smile: Hedging and Spanning in Option Markets,” *Review of Financial Studies*, 14, 495–527.
- CAO, J. AND B. HAN (2013): “Cross section of option returns and idiosyncratic stock volatility,” *Journal of Financial Economics*, 108, 231–249.
- CHEN, L., M. PELGER, AND J. ZHU (2023): “Deep Learning in Asset Pricing,” *Management Science*, 70.
- CHO, K., B. VAN MERRIENBOER, C. GULCEHRE, D. BAHDANAU, F. BOUGARES, H. SCHWENK, AND Y. BENGIO (2014): “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation,” *arXiv*.
- CYBENKO, G. (1989): “Approximation by superpositions of a sigmoidal function,” *Mathematics of Control, Signals, and Systems*, 2, 303–314.

- DUCHI, J. C., E. HAZAN, AND Y. SINGER (2011): “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” *Journal of Machine Learning Research*, 12, 2121–2159.
- EL HIHI, S. AND Y. BENGIO (1995): “Hierarchical Recurrent Neural Networks for Long-Term Dependencies,” in *Proceedings of the 8th International Conference on Neural Information Processing Systems*, Cambridge, MA, USA: MIT Press, NIPS’95, 493–499.
- ELEND, L., S. A. TIDEMAN, K. LOPATTA, AND O. KRAMER (2020): “Earnings Prediction with Deep Learning,” *Lecture Notes in Computer Science*, 12325, 267–274.
- ELMAN, J. (1990): “Finding structure in time,” *Cognitive Science*, 14, 179–211.
- FISCHER, T. AND C. KRAUSS (2018): “Deep learning with long short-term memory networks for financial market predictions,” *European Journal of Operational Research*, 270, 654–669.
- GOODFELLOW, I., Y. BENGIO, AND A. COURVILLE (2016): *Deep Learning*, Cambridge, MA, USA: MIT Press.
- GOYAL, A. AND A. SARETTO (2009): “Cross-section of option returns and volatility,” *Journal of Financial Economics*, 94, 310–326.
- GRAVES, A. (2013): “Generating Sequences With Recurrent Neural Networks,” *arXiv*.
- GRAVES, A. AND N. JAITLEY (2014): “Towards End-To-End Speech Recognition with Recurrent Neural Networks,” *International Conference on Machine Learning*, 1764–1772.
- GRAVES, A., M. LIWICKI, S. FERNÁNDEZ, R. BERTOLAMI, H. BUNKE, AND J. SCHMIDHUBER (2009): “A novel connectionist system for unconstrained handwriting recognition,” *IEEE transactions on pattern analysis and machine intelligence*, 31, 855–868.
- GU, S., B. KELLY, AND D. XIU (2020): “Empirical Asset Pricing via Machine Learning,” *Review of Financial Studies*, 33, 2223–2273.
- HASTIE, T., R. TIBSHIRANI, AND J. H. FRIEDMAN (2017): *The elements of statistical learning: Data mining, inference, and prediction*, Springer series in statistics, New York, NY: Springer, second edition, corrected at 12th printing 2017 ed.
- HAYKIN, S. (2009): *Neural networks and learning machines*, New York: Pearson, 3rd ed.
- HAYWOOD, J. AND G. T. WILSON (1997): “Fitting Time Series Models by Minimizing Multistep-Ahead Errors: A Frequency Domain Approach,” *Journal of the Royal Statistical Society. Series B (Methodological)*, 59, 237–254.
- HEATON, J. B., N. G. POLSON, AND J. H. WITTE (2017): “Deep learning for finance: deep portfolios,” *Applied Stochastic Models in Business and Industry*, 33, 3–12.
- HOCHREITER, S. (1991): “Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München,” .
- HOCHREITER, S. AND J. SCHMIDHUBER (1997): “Long short-term memory,” *Neural computation*, 9, 1735–1780.
- HOLT, C. (1957): “Forecasting seasonals and trends by exponentially weighted moving averages,” *O.N.R. research memorandum*.

- HOPFIELD, J. J. (1982): “Neural networks and physical systems with emergent collective computational abilities,” *Proceedings of the National Academy of Sciences of the United States of America*, 79, 2554–2558.
- HORNIK, K., M. STINCHCOMBE, AND H. WHITE (1989): “Multilayer feedforward networks are universal approximators,” *Neural Networks*, 2, 359–366.
- HU, G. AND K. JACOBS (2020): “Volatility and Expected Option Returns,” *Journal of Financial and Quantitative Analysis*, 55, 1025–1060.
- JACOBS, R. A. (1988): “Increased rates of convergence through learning rate adaptation,” *Neural Networks*, 1, 295–307.
- JAEGER, H. (2007): “Discovering multiscale dynamical features with hierarchical Echo State Networks,” *Jacobs University Technical Report*.
- JIANG, J., B. KELLY, AND D. XIU (2023): “(Re-)Imag(in)ing Price Trends,” *The Journal of Finance*, 78, 3193–3249.
- KINGMA, D. P. AND J. BA (2014): “Adam: A Method for Stochastic Optimization,” *arXiv*.
- LI, X. AND X. WU (2015): “Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition,” in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4520–4524.
- LINNAINMAA, S. (1976): “Taylor expansion of the accumulated rounding error,” *BIT Numerical Mathematics*, 16, 146–160.
- LÜTKEPOHL, H. (1991): “Vector Autoregressive Moving Average Processes,” in *Introduction to multiple time series analysis*, ed. by H. Lütkepohl, Berlin: Springer, 217–240.
- MCNEIL, A. J., R. FREY, AND P. EMBRECHTS (2015): *Quantitative risk management: Concepts, techniques and tools*, Princeton series in finance, Princeton and Oxford: Princeton University Press, revised ed.
- MUTH, J. F. (1960): “Optimal Properties of Exponentially Weighted Forecasts,” *Journal of the American Statistical Association*, 55, 299–306.
- OPPENHEIM, A. V. AND R. W. SCHAFER (1975): *Digital signal processing*, Englewood Cliffs, NJ: Prentice-Hall, 24th ed.
- PASCANU, R., C. GULCEHRE, K. CHO, AND Y. BENGIO (2014): “How to construct deep recurrent neural networks,” *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*.
- PEARLMUTTER, B. A. (1995): “Gradient calculations for dynamic recurrent neural networks: a survey,” *IEEE transactions on neural networks*, 6, 1212–1228.
- PINEDA (1987): “Generalization of back-propagation to recurrent neural networks,” *Physical Review Letters*, 59, 2229–2232.
- ROTHFUSS, J., F. FERREIRA, S. WALTHER, AND M. ULRICH (2019): “Conditional Density Estimation with Neural Networks: Best Practices and Benchmarks,” *arXiv*.

- RUMELHART, D. E., G. E. HINTON, AND R. J. WILLIAMS (1986): “Learning representations by back-propagating errors,” *Nature*, 323, 533–536.
- SAK, H., A. SENIOR, AND F. BEAUFAYS (2014): “Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition,” *arXiv*.
- SCHAUL, T., I. ANTONOGLOU, AND D. SILVER (2014): “Unit tests for stochastic optimization,” *International Conference on Learning Representations*.
- SCHMIDHUBER, J. (1992): “Learning Complex, Extended Sequences Using the Principle of History Compression,” *Neural computation*, 4, 234–242.
- SCHUSTER, M. AND K. K. PALIWAL (1997): “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, 45, 2673–2681.
- SHI, X., Z. CHEN, H. WANG, D.-Y. YEUNG, W.-K. WONG, AND W.-C. WOO (2015): “Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, Cambridge, MA, USA: MIT Press, NIPS’15, 802–810.
- SIRIGNANO, J. AND R. CONT (2019): “Universal features of price formation in financial markets: perspectives from deep learning,” *Quantitative Finance*, 19, 1449–1459.
- SRIVASTAVA, N., G. HINTON, A. KRIZHEVSKY, I. SUTSKEVER, AND R. SALAKHUTDINOV (2014): “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, 15, 1929–1958.
- SRIVASTAVA, N., E. MANSIMOV, AND R. SALAKHUTDINOV (2015): “Unsupervised Learning of Video Representations Using LSTMs,” in *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, JMLR.org, ICML’15, 843–852.
- SUTSKEVER, I., O. VINYALS, AND Q. V. LE (2014): “Sequence to Sequence Learning with Neural Networks,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, Cambridge, MA, USA: MIT Press, NIPS’14, 3104–3112.
- TEALAB, A. (2018): “Time series forecasting using artificial neural networks methodologies: A systematic review,” *Future Computing and Informatics Journal*, 3, 334–340.
- TIELEMAN, T. AND G. HINTON (2012): “Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of its Recent Magnitude,” *COURSERA: Neural Networks for Machine Learning*, 26–30.
- VINYALS, O., L. KAISER, T. KOO, S. PETROV, I. SUTSKEVER, AND G. HINTON (2015): “Grammar as a Foreign Language,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, Cambridge, MA, USA: MIT Press, NIPS’15, 2773–2781.
- WERBOS, P. J. (1990): “Backpropagation through time: what it does and how to do it,” *Proceedings of the IEEE*, 78, 1550–1560.
- WILSON, D. AND T. R. MARTINEZ (2003): “The general inefficiency of batch training for gradient descent learning,” *Neural Networks*, 16, 1429–1451.

- XUE, X., J. FENG, Y. GAO, M. LIU, W. ZHANG, X. SUN, A. ZHAO, AND S. GUO (2019): “Convolutional Recurrent Neural Networks with a Self-Attention Mechanism for Personnel Performance Prediction,” *Entropy*, 21, 1–16.
- ZHAN, X., B. HAN, J. CAO, AND Q. TONG (2022): “Option Return Predictability,” *Review of Financial Studies*, 35, 1394–1442.