



# DEEP LEARNING PROJECT

## Choice of the Number of Hidden Layers and Application to Price Prediction

Gioele Checchi / Manuel Griseri / Samuele Landi

Master 2 : Quantitative Finance

Academic year 2025/2026

# Contents

<b>Introduction</b>	<b>2</b>
<b>1 Overview of the Paper</b>	<b>3</b>
1.1 Theoretical Foundations . . . . .	3
1.1.1 Universal Approximation Properties and Network Depth . . . . .	3
1.1.2 Back-Propagation Algorithm . . . . .	4
1.2 Methodology . . . . .	6
1.2.1 Data Preparation and Pre-processing . . . . .	6
1.2.2 Construction of the Neural Network Models . . . . .	7
1.2.3 Model Evaluation . . . . .	8
<b>2 Reproduction and Critique of the Paper</b>	<b>9</b>
2.1 Reproduction of the Results . . . . .	9
2.1.1 Single Hidden Layer . . . . .	10
2.1.2 Multiple Hidden Layers . . . . .	13
2.2 Criticisms of the study . . . . .	16
<b>3 Beyond the Paper</b>	<b>18</b>
3.1 Comparison with Alternative Models . . . . .	18
3.1.1 Autoregressive Integrated Moving Average . . . . .	18
3.1.2 Support Vector Machine . . . . .	19
3.1.3 Long Short Term Memory . . . . .	21
3.1.4 Hybrid ARIMA-LSTM . . . . .	23
3.2 Alternative Framework . . . . .	24
3.2.1 Alternative Dataset . . . . .	24
3.2.2 Alternative Train-Test Split . . . . .	25
<b>Conclusion</b>	<b>28</b>
<b>References</b>	<b>29</b>

# Introduction

This report focuses on the analysis of the 2019 paper "Choice of the Number of Hidden Layers for Back Propagation Neural Network Driven by Stock Price Data and Application to Price Prediction", authored by Peipei Zhang (School of Data and Computer Science, Shandong Women's University, Jinan, P.R. China) and Chuanhe Shen (Institute of Financial Engineering, Shandong Women's University, Jinan, P.R. China).

Many studies pay attention to avoid falling in local minima and increase predictiveness of BPNN with algorithm and methods without considering its constitution. The scope of this paper is to determine how to set the right amount of epochs, neurons and hidden layers for a BPNN. The main idea of the paper is to analyze the accuracy of the single hidden layer prediction model firstly and then scale the model to a multi hidden layer. Then, to determine the optimal amount of epochs the authors consider for performance metric the Mean Absolute Error (MAE) calculated on a Test set and compare it between 200, 500, and 1000 epochs. Similarly, to test for the best number of hidden layers is chosen between 1 (single layer) and 5, the MAE is used to compare the results for multiple tests over the test data set. Finally, the performance of the single layer and multi layer models are evaluated on accuracy and computing speed.

Stock pricing has historically relied on statistical forecasting methods such as exponential smoothing, linear regression, moving average and ARIMA. These model are acceptable for modeling stationary time series data and short-term forecasting, however, the stock market is dynamic, non linear and non-parametric. The same applies for ARCH and GARCH models which can handle even non linear time series but they fail to discern the forces that model the market. Machine learning models can address these limitations, in particular Backpropagation Neural Networks (BPNN) are shown to outperform ARIMA in forecasting. These models can be further optimized by using techniques such as Levenberg-Marquardt algorithm, momentum learning and self-adaptive learning rate, genetic algorithm, simulating annealing algorithm and wavelets analysis methods.

In Chapter 1, we present the paper as it is: starting from the theoretical framework of neural networks, we then explain how the authors technically proceeded in order to obtain their results. In Chapter 2, we move on to our own work, where we attempted to implement code to reproduce the paper's findings, followed by a series of critical reflections. Finally, in Chapter 3, we extend the content of the original paper by, for example, comparing the proposed model with other classical time-series predictive models, modifying the experimental setting based on our earlier critical observations, and attempting to implement a more recent hybrid model.

# Chapter 1

## Overview of the Paper

As mentioned in the introduction, this first chapter presents the contents of the paper. The first section provides a brief explanation of how a backpropagation neural network works, while the second section examines how this theoretical framework is applied in the paper to perform time series prediction.

### 1.1 Theoretical Foundations

Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function  $f^*$ . For example, for a classifier,  $y = f^*(x)$  maps an input  $x$  to a category  $y$ . A feedforward network defines a mapping  $y = f(x; \theta)$  and learns the value of the parameters  $\theta$  that result in the best function approximation. They are important because they form the foundations of artificial intelligence, for example, in more complex architectures such as transformers, several layers are implemented as MLPs.

In the first subsection, we discuss the advantages of constructing a neural network in the classical manner and the role of adding multiple layers. The second subsection summarizes the mathematical formulation of the Back-Propagation Neural Network (BPNN) used in the study.

#### 1.1.1 Universal Approximation Properties and Network Depth

A linear model can only represent linear functions, and thus cannot capture the nonlinear relationships often required in practical learning tasks. Feedforward neural networks with hidden layers overcome this limitation by providing a universal approximation framework. According to the universal approximation theorem, a feedforward network with a linear output layer and at least one hidden layer using a squashing activation function (e.g., the logistic sigmoid) can approximate any Borel measurable function on a finite-dimensional space, given sufficiently many hidden units. This theorem guarantees that a sufficiently large multilayer perceptron can represent any target function, but it does not ensure that the training algorithm will successfully learn it. Learning may fail due to optimization difficulties or overfitting. Furthermore, the theorem does not address the efficiency of the representation: although a single hidden layer is theoretically sufficient, it may require an infeasibly large number of units.

Subsequent work, established bounds on the number of units needed in shallow networks. In worst-case scenarios, approximating certain functions with a one-hidden-layer network may require an exponential number of hidden units. Deeper networks, however, can represent the same functions far more efficiently. In Figure 1.1, geometric interpretations show how each hidden unit can fold the input space along hyperplanes, creating mirrored or repeated patterns. Composing

many such folds across layers yields highly expressive, piecewise linear functions. This illustrates the exponential representational advantage of depth: many function families can be efficiently represented only by networks whose depth exceeds a certain threshold.

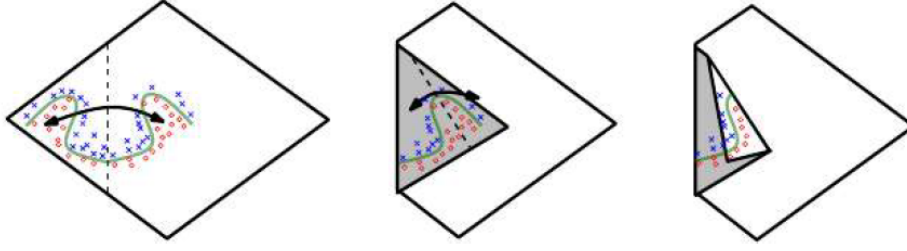


Figure 1.1: Intuitive explanation of the advantage of deeper networks

In summary, while single-hidden-layer networks are universal approximators, deeper architectures often achieve the same expressive power with significantly fewer units and improved generalization. Depth thus provides both representational efficiency and practical advantages in learning complex functions. For further details, refer to Chapter 6 of [2].

### 1.1.2 Back-Propagation Algorithm

#### Network Structure

A feed-forward BPNN consists of an input layer, one or more hidden layers, and an output layer. Let:

- $s_l$  = number of neurons in layer  $l$ ,
- $h_k^{(l)}$  = output of neuron  $k$  in layer  $l$ ,
- $w_{kj}^{(l)}$  = weight from neuron  $j$  in layer  $(l - 1)$  to neuron  $k$  in layer  $l$ ,
- $\theta_k^{(l)}$  = bias of neuron  $k$  in layer  $l$ .

The activation function used is the logistic sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}}, \quad f'(x) = f(x)(1 - f(x)) \quad (1.1)$$

#### Forward Propagation

(1) **Weighted input to neuron  $k$  in layer  $l$ :**

$$\text{net}_k^{(l)} = \sum_{j=1}^{s_{l-1}} w_{kj}^{(l)} h_j^{(l-1)} + \theta_k^{(l)} \quad (1.2)$$

(2) **Output of neuron  $k$  in layer  $l$ :**

$$h_k^{(l)} = f(\text{net}_k^{(l)}) \quad (1.3)$$

For the input layer:

$$h_k^{(1)} = x_k \quad (1.4)$$

## Error Function

Given a training pair  $(x(k), z(k))$ , the prediction error for sample  $k$  is defined using the MAE:

$$E(k) = \sum_{j=1}^{s_L} |z_j(k) - y_j(k)| \quad (1.5)$$

where  $y_j(k) = h_j^{(L)}$  is the output of neuron  $j$  in the output layer. The average MAE over all  $N$  training samples is:

$$E = \frac{1}{N} \sum_{k=1}^N E(k) \quad (1.6)$$

In the case of a single-output network ( $s_L = 1$ ), this simplifies to:

$$E(k) = |z(k) - y(k)| \quad (1.7)$$

## Backward Propagation of Errors

Define the error signal (local gradient) for each neuron.

(1) **Output layer** ( $l = L$ ):

$$\delta_k^{(L)} = \text{sign}(z_k - h_k^{(L)}) f'(\text{net}_k^{(L)}) \quad (1.8)$$

(2) **Hidden layers** ( $l = 2, \dots, L - 1$ ):

$$\delta_k^{(l)} = f'(\text{net}_k^{(l)}) \sum_{i=1}^{s_{l+1}} \delta_i^{(l+1)} w_{ik}^{(l+1)} \quad (1.9)$$

## Parameter Updates (Gradient Descent)

(1) **Weight update:**

$$\Delta w_{kj}^{(l)}(t) = \alpha \delta_k^{(l)} h_j^{(l-1)} \quad (1.10)$$

$$w_{kj}^{(l)}(t+1) = w_{kj}^{(l)}(t) + \Delta w_{kj}^{(l)}(t) \quad (1.11)$$

(2) **Bias update:**

$$\Delta \theta_k^{(l)}(t) = \alpha \delta_k^{(l)} \quad (1.12)$$

$$\theta_k^{(l)}(t+1) = \theta_k^{(l)}(t) + \Delta \theta_k^{(l)}(t) \quad (1.13)$$

Here  $\alpha \in (0, 1)$  is the learning rate.

## Momentum Term

To improve convergence and avoid local minima, the author added a momentum term.

**(1) Weight update with momentum:**

$$\Delta w_{kj}^{(l)}(t+1) = \alpha \delta_k^{(l)} h_j^{(l-1)} + \eta \Delta w_{kj}^{(l)}(t) \quad (1.14)$$

**(2) Bias update with momentum:**

$$\Delta \theta_k^{(l)}(t+1) = \alpha \delta_k^{(l)} + \eta \Delta \theta_k^{(l)}(t) \quad (1.15)$$

where  $\eta \in [0.1, 0.8]$  is the momentum coefficient.

### Mini-Batch Gradient Descent

Instead of using the full training set at each update, mini-batch gradient descent updates parameters using  $M$  randomly selected samples:

$$E_{\text{batch}} = \frac{1}{M} \sum_{k=1}^M E(k) \quad (1.16)$$

All update rules above apply with  $E_{\text{batch}}$  replacing the full error. In MLP training, minibatches are used because they offer a trade-off between the high noise of fully stochastic updates and the computational inefficiency of full-batch gradient descent.

## 1.2 Methodology

The methodological approach of the paper consists of four main components: data preparation, pre-processing, model construction, and model evaluation.

### 1.2.1 Data Preparation and Pre-processing

The authors used daily trading data from the Industrial and Commercial Bank of China covering the period from June 1, 2016 to June 26, 2018. The dataset contains the daily opening price, highest price, lowest price, closing price, and trading volume.

Since raw financial time series are non-stationary (as shown in the reconstruction section), contain different units of measurement, and exhibit substantial scale variation, and because the authors wanted to highlight the intrinsic learning capability of the BPNN, they apply the following log-return transformation:

$$r_t = \ln\left(\frac{u_t}{u_{t-1}}\right) \quad (1.17)$$

where  $u_t$  is the price of the stock at time  $t$ . After applying this transformation, the dataset is divided into a training set and a testing set at June 5, 2018, corresponding to approximately 97% training data and 3% testing data. To construct the input-output pairs for the neural network, the authors reorganize the time series so that, for each day, the opening, highest, lowest, and closing prices together with the trading volume from the previous three days form the input vector, while the closing price of the following day serves as the output. This results in an input layer with  $m = 15$  neurons and an output layer with  $n = 1$  neuron.

## 1.2.2 Construction of the Neural Network Models

### Single-hidden-layer Neural Network

They begin by training a single-hidden-layer Back-Propagation Neural Network, which serves as the baseline model. The number of neurons in the hidden layer, denoted by  $s_2$ , is initially chosen according to the heuristic formulas proposed in [7]:

$$s_2 = \sqrt{mn} \quad (1.18)$$

$$s_2 = \log_2(m) \quad (1.19)$$

$$s_2 = \sqrt{m+1} + \alpha \quad (1.20)$$

where  $m$  and  $n$  denote the numbers of input and output neurons, respectively, and  $\alpha \in [0, 10]$  is a constant. These heuristic expressions collectively indicate that the “optimal” number of hidden neurons lies approximately within the interval  $2 \leq s_2 \leq 15$ . Accordingly, the authors systematically vary the hidden-layer size from  $s_2 = 2$  to 15, training each model using the Back-Propagation algorithm with momentum (reported to lie between 0.1 and 0.8, although the exact value used is not specified) together with a mini-batch gradient descent optimization scheme. Each configuration is trained under three different epoch settings: 200, 500, and 1000 epochs, to examine the effect of training duration on predictive performance.

### Multiple-layers Neural Network

To investigate the impact of multiple hidden layers, the same Back-Propagation algorithm is applied again, this time with a fixed training duration of 1000 epochs, while keeping the number of neurons in the first hidden layer equal to the value that gave the best single-layer prediction. Additional hidden layers are then progressively added, up to a maximum network depth of seven (corresponding to five hidden layers). The number of neurons in each subsequent hidden layer is computed according to equations (1.21)-(1.23) as follows:

$$s_{l+1} = \sqrt{s_l s_{l+2}} \quad (1.21)$$

$$s_{l+1} = \log_2(s_l) \quad (1.22)$$

$$s_{l+1} = \sqrt{s_l + 1} + \alpha \quad (1.23)$$

where  $s_l$  denotes the number of neurons in the  $l$ -th hidden layer,  $s_{l+1}$  denotes the number of neurons in the subsequent  $(l+1)$ -th hidden layer, and  $\alpha$  is a constant in the interval  $[0, 10]$ . Which leads to the following neuron configurations for the corresponding neural network architectures:

Table 1.1: Network architectures and corresponding neuron counts	
Number of Hidden Layers	Neuron Configuration
1 hidden layer	$(m, s_2, n) = (15, 14, 1)$
2 hidden layers	$(m, s_2, s_3, n) = (15, 4, 10, 1)$
3 hidden layers	$(m, s_2, s_3, s_4, n) = (15, 5, 4, 4, 1)$
4 hidden layers	$(m, s_2, s_3, s_4, s_5, n) = (15, 5, 5, 5, 5, 1)$
5 hidden layers	$(m, s_2, s_3, s_4, s_5, s_6, n) = (15, 5, 5, 5, 5, 5, 1)$

### 1.2.3 Model Evaluation

Each trained network is evaluated on the testing set using the MAE between the predicted and actual closing prices, which is defined, in this framework, as follows:

$$\text{MAE} = \frac{1}{T} \sum_{t=1}^T |z_t - y_t| \quad (1.24)$$

where  $T$  is the number of forecasting periods, and  $z_t$  and  $y_t$  denote the actual and predicted closing prices at period  $t$ , respectively, with  $t \in \{1, 2, \dots, T\}$ . The resulting errors are then compared across different architectures to assess whether deeper networks resulted in meaningful improvements over the single-hidden-layer baseline. In addition to numerical evaluation, graphical analyses are employed to help select the most appropriate number of neurons and to visually compare the actual and predicted closing prices for each model used.

# Chapter 2

## Reproduction and Critique of the Paper

In this second chapter, we first attempt to faithfully reproduce the results of the paper by implementing our own Python code, since the original work did not provide a GitHub repository. This attempt was only partially successful, and for this reason we conclude the chapter with a critical discussion that highlights several factors contributing to the irreproducibility of the paper's exact results.

### 2.1 Reproduction of the Results

We retrieved the data through `yfinance` Python package (in the paper is Tushare), then we processed in order to have the same data set structure as the authors. We computed a table to compute skewness, kurtosis and Jarque-Bera test using the library `jarque_bera` from `scipy.stats`. With the help of `matplotlib` package we reproduced the histogram and statistics of the log-return series:

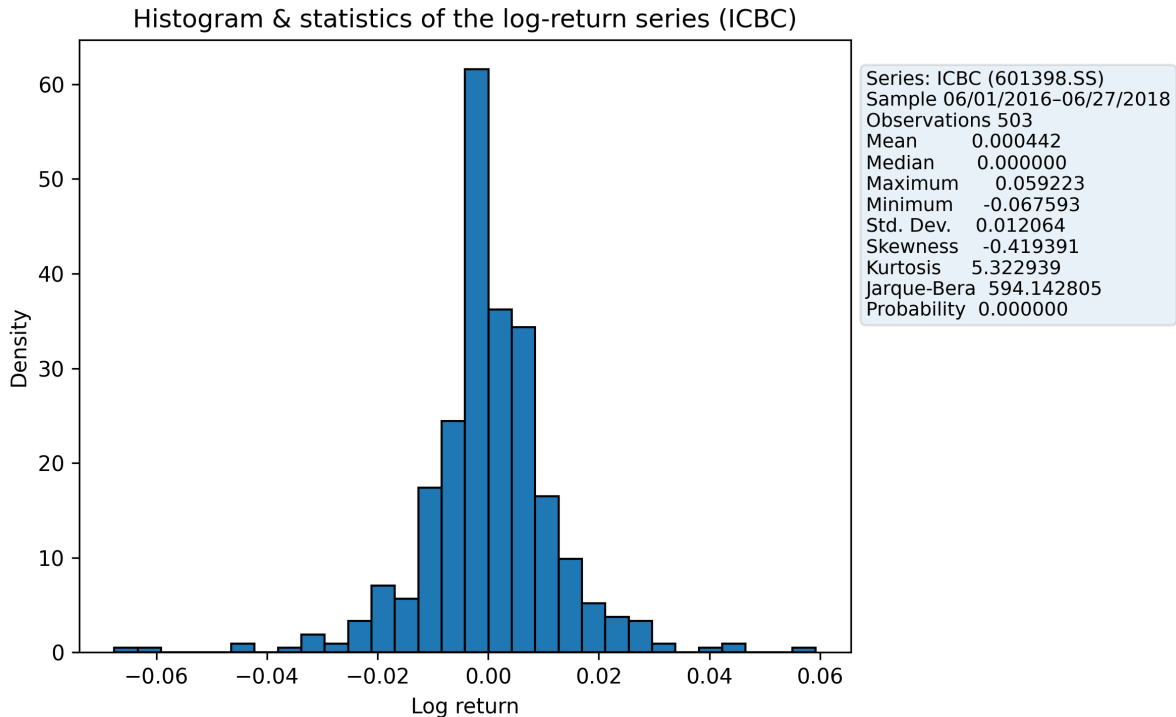


Figure 2.1: Empirical distribution of ICBC log-returns

After checking the similarity with the distribution computed by the authors we also realized the time series of the logarithmic return:

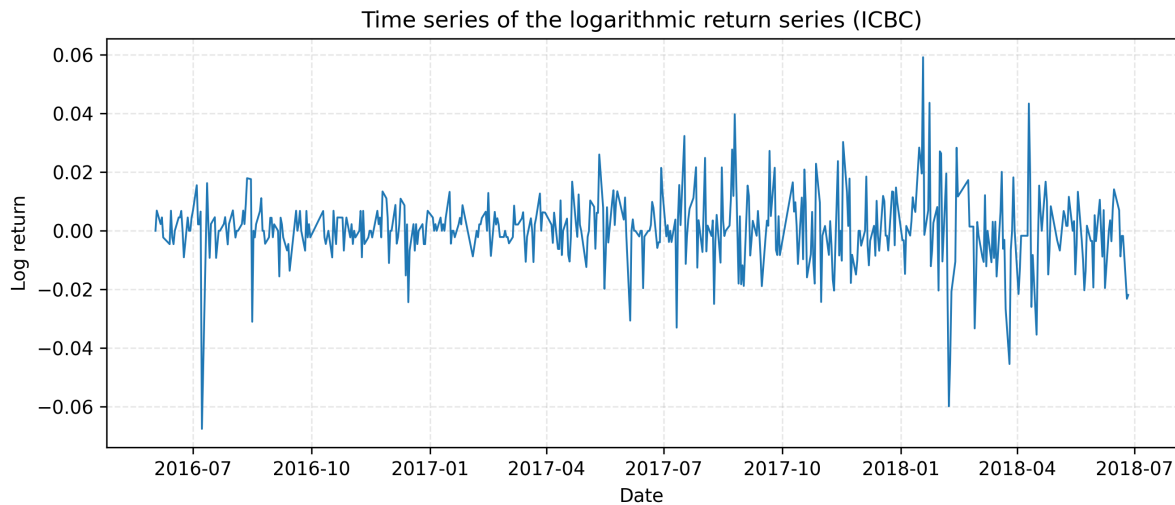


Figure 2.2: Series of ICBC log-returns

### 2.1.1 Single Hidden Layer

First, we defined the `make_sliding_features` function which allows us to compute the logarithmic returns on the open, high, low, close, volume with a two days window:

```

1 def make_sliding_features(d, window=2):
2     if "logret_open" not in d.columns: d["logret_open"] = np.log(d["
   Open"]).diff()
3     if "logret_high" not in d.columns: d["logret_high"] = np.log(d["
   High"]).diff()
4     if "logret_low" not in d.columns: d["logret_low"] = np.log(d["
   Low"]).diff()
5     if "logret_volume" not in d.columns: d["logret_volume"] = np.log(d["
   Volume"].replace(0, np.nan)).diff()
6
7     cols = [f"logret_{c}" for c in ["open", "high", "low", "close", "volume"
   ]]
8
9     feats = []
10    for lag in range(window, -1, -1): # t-2, t-1, t
11        for c in cols:
12            feats.append(d[c].shift(lag))
13    X = pd.concat(feats, axis=1)
14    X.columns = [f"{c}_t-{lag}" for lag in range(window, -1, -1) for c in
   cols]
15
16    y = d["logret_close"].shift(-1) # Target: next-day log-return of
   Close (in t+1)
17    return X, y

```

Second, we separated between train and test sets by

```

1 train_end = pd.Timestamp("2018-06-04")
2 test_start = pd.Timestamp("2018-06-05")
3 test_end = pd.Timestamp("2018-06-26")
4
5 mask_train = (X.index <= train_end)
6 mask_test = (X.index >= test_start) & (X.index <= test_end)
7
8 X_train, y_train = X.loc[mask_train].dropna(), y.loc[mask_train].dropna()
9
10 idx_inter = X_train.index.intersection(y_train.index)
11 X_train, y_train = X_train.loc[idx_inter], y_train.loc[idx_inter]
12
13 X_test, y_test = X.loc[mask_test].dropna(), y.loc[mask_test].dropna()
14 idx_inter_t = X_test.index.intersection(y_test.index)
15 X_test, y_test = X_test.loc[idx_inter_t], y_test.loc[idx_inter_t]
16
17 close_series = df["Close"].astype(float)

```

Third, we fitted the model and calculated the error of the single hidden layer in function of the number of neurons, we did the analysis for 200, 500 and 1000 epochs:

```

1 neurons_list = list(range(2, 16)) # s2 in {2,...,15}
2 epochs_list = [200, 500, 1000]
3
4 def train_and_score(n_hidden, epochs):
5     # Single hidden layer with sigmoid + SGD momentum
6     model = MLPRegressor(
7         hidden_layer_sizes=n_hidden,
8         activation="logistic",
9         solver="sgd",
10        learning_rate_init=0.01,
11        momentum=0.9,
12        batch_size=32,
13        max_iter=epochs,
14        shuffle=True,
15        n_iter_no_change=epochs + 1, # Disable early stopping
16        tol=0.0,
17        random_state=3407
18    )
19    model.fit(X_train_s, y_train.values)
20    pred = pd.DataFrame(index=X_test.index)
21    pred["prev_close"] = close_series.reindex(X_test.index)
22    pred["yhat_ret"] = model.predict(X_test_s)
23    pred["y_pred"] = pred["prev_close"] * np.exp(pred["yhat_ret"])
24    pred["y_true"] = close_series.shift(-1).reindex(X_test.index)
25    # I predict C(t+1) so I need to shift in order to align
26    pred = pred.dropna(subset=["y_pred", "y_true"])
27
28    ytrue = pred["y_true"].to_numpy().reshape(-1) # reshape(-1):
29    # matrix to vector
30    ypred = pred["y_pred"].to_numpy().reshape(-1)
31    mae = np.mean(np.abs(ytrue - ypred))

```

```

30     return mae, ypred, ytrue
31
32 results = {e: [] for e in epochs_list}
33 pred_store = {}
34
35 for e in epochs_list:
36     for s in neurons_list:
37         mae, yhat_price, ytrue = train_and_score((s,), e)
38         results[e].append(mae)

```

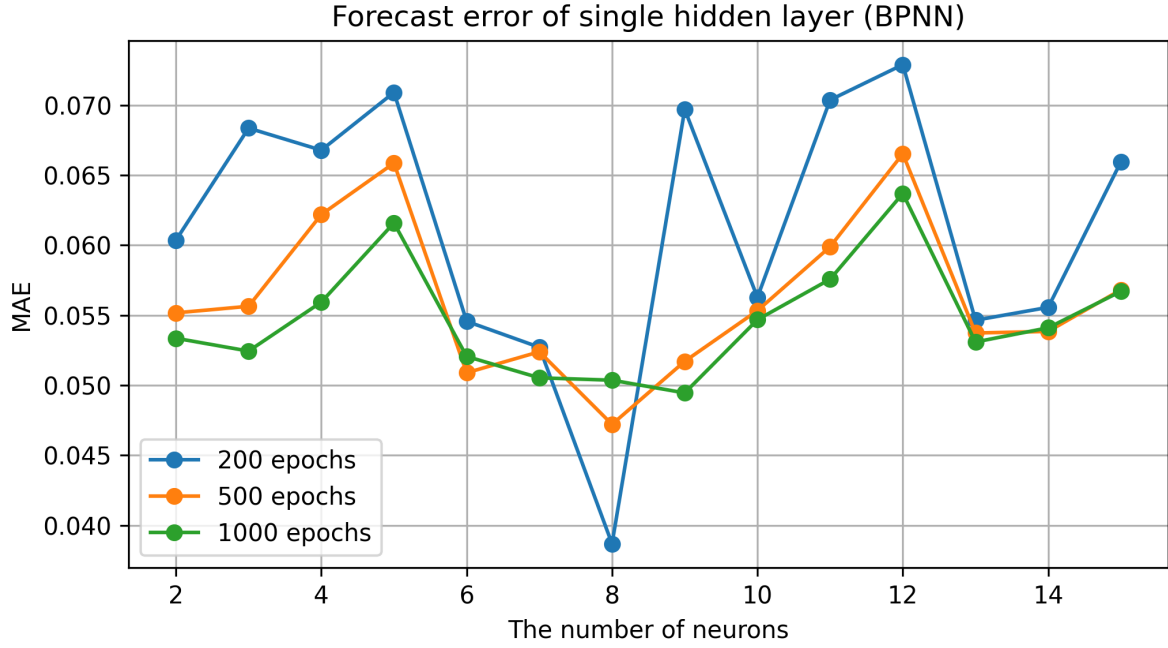


Figure 2.3: MAE on the test set for different neuron counts and epochs

Table 2.1: MAE on the test set for different neuron counts and epochs

Neurons	200 Epochs	500 Epochs	1000 Epochs
2	0.060359	0.055174	0.053364
3	0.068365	0.055644	0.052446
4	0.066789	0.062194	0.055942
5	0.070913	0.065868	0.061596
6	0.054572	0.050898	0.052064
7	0.052719	0.052399	0.050536
8	<b>0.038688</b>	0.047215	0.050368
9	0.069700	0.051716	0.049460
10	0.056284	0.055335	0.054686
11	0.070376	0.059895	0.057578
12	0.072898	0.066532	0.063697
13	0.054650	0.053738	0.053097
14	0.055570	0.053850	0.054108
15	0.065958	0.056822	0.056710

From the image we see that the minimal error is achieved with 8 neurons and 200 epochs. This is the case for our model because the paper identified a configuration with 14 neurons and 1000 epochs as the best one, yielding an MAE of 0.052; therefore, our model performs better. As we will discuss in the critique section, this discrepancy is mainly due to the lack of explicit hyperparameter values reported by the authors. Then, we compared the actual and the predicted closing prices on the test set.

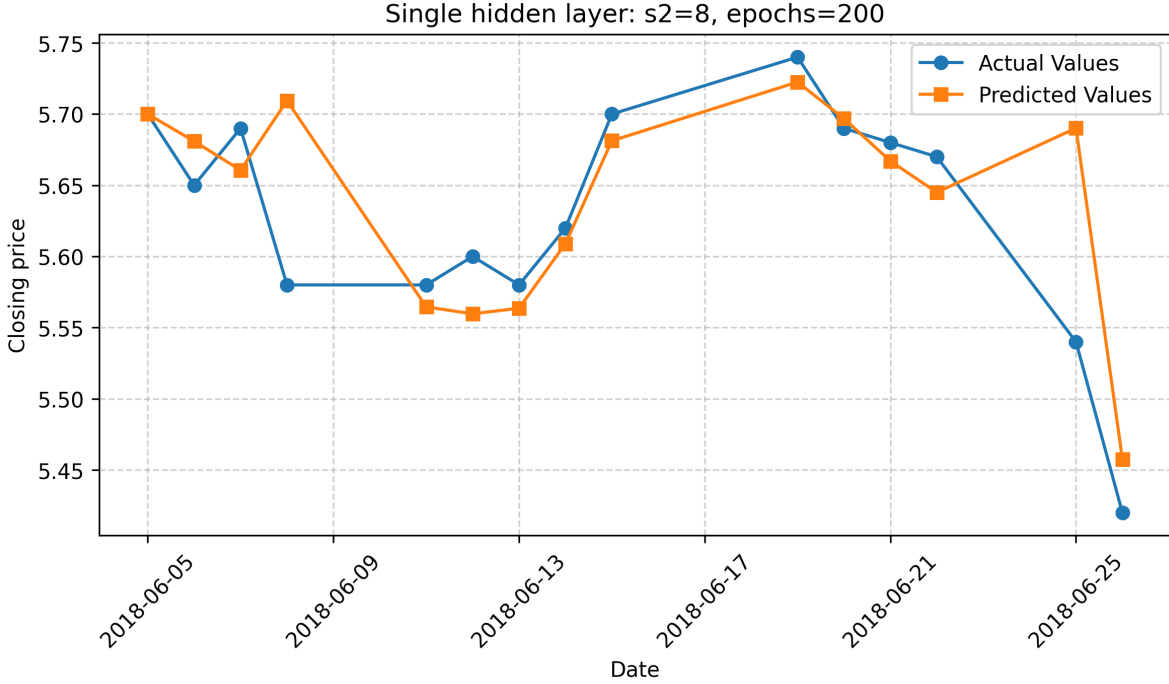


Figure 2.4: Predicted values of the best single hidden layer NN

### 2.1.2 Multiple Hidden Layers

To reproduce the experiment for deeper network architectures, we first defined a dictionary mapping each value of  $L \in \{3, 4, 5, 6, 7\}$  to its corresponding tuple of hidden-layer sizes, following the neuron configurations described in the paper and presented in the table below:

Table 2.2: Hidden-layer architectures and number of trainable parameters for each  $L$

Architecture (L)	Hidden-Layer Sizes	# Parameters
3	(8)	137
4	(4, 10)	125
5	(5, 4, 4)	129
6	(5, 5, 5, 5)	176
7	(5, 5, 5, 5, 5)	206

After fixing the number of training epochs to 200 for all models, we then constructed the true output series by shifting the closing-price vector by one day ahead, ensuring that the network always predicts the following day's closing price. These values were aligned with the indices of the test set so that each prediction could be compared directly with its corresponding ground truth. Next, we initialized a results table containing the test dates and actual closing prices and looped over all architectures, calling the function `train_and_score(arch, EPOCHS)`, which trains the network with the specified hidden-layer structure

```

1 mae_results = {}
2 table = pd.DataFrame({"Date": dates_out, "Actual Values": ytrue_out})
3 table = table.set_index("Date")
4
5 for L, arch in architectures.items():
6     mae_tmp, ypred_vec, _ = train_and_score(arch, EPOCHS)
7     table[f"L={L}"] = ypred_vec
8     mae_results[L] = mae_tmp

```

This procedure returns, for each architecture, the MAE of the model on the test set, the vector of predicted closing prices, and the fitted model. For each value of  $L$ , the predicted prices are added as a new column ( $L=3$ ,  $L=4$ , etc.) in the results table, while the MAE values are stored in a separate dictionary. At the end of the loop, we print the tables that compare the actual closing prices with the predictions of each architecture during the test period:

Table 2.3: Sample results of L-layer BPNN of ICBC

Date	Actual	L=3	L=4	L=5	L=6	L=7
2018-06-06	5.65	5.6809	5.7048	5.7013	5.7033	5.7074
2018-06-07	5.69	5.6606	5.6515	5.6606	5.6523	5.6573
2018-06-08	5.58	5.7095	5.6719	5.6967	5.6927	5.6974
2018-06-11	5.58	5.5646	5.6168	5.5868	5.5828	5.5872
2018-06-12	5.60	5.5597	5.5638	5.5926	5.5813	5.5870
2018-06-13	5.58	5.5636	5.6528	5.6072	5.6019	5.6066
2018-06-14	5.62	5.6090	5.6207	5.5854	5.5831	5.5871
2018-06-15	5.70	5.6812	5.6075	5.6201	5.6229	5.6272
2018-06-19	5.74	5.7225	5.7145	5.6977	5.7037	5.7072
2018-06-20	5.69	5.6968	5.7300	5.7391	5.7439	5.7478
2018-06-21	5.68	5.6670	5.6489	5.6996	5.6921	5.6975
2018-06-22	5.67	5.6450	5.6962	5.6881	5.6822	5.6871
2018-06-25	5.54	5.6901	5.6440	5.6864	5.6723	5.6775
2018-06-26	5.42	5.4576	5.5825	5.5383	5.5435	5.5473

and the summary table listing the MAE obtained by each model depth:

Table 2.4: MAE summary table for architectures  $L = 3, \dots, 7$

Architecture (L)	MAE
3	0.038688
4	0.058108
5	0.053376
6	0.052243
7	0.053561

This procedure allows us to directly compare the forecasting performance of networks with different numbers of hidden layers under identical training conditions, as done by the authors. However, we again observe a discrepancy: while the paper reports the best model as the one with two hidden layers, in our case the single-hidden-layer architecture remains the best.

Before producing the comparison plots, we added an initial row to the results table corresponding to the first test date (05/06/2018). Since this day has no predicted values from the sliding-window

construction, we assigned the actual closing price to all model columns so that all curves begin on the same date. This was implemented by creating a new row with index 2018-06-05 and concatenating it to the top of the existing results table. To visually compare the behaviour of the different network depths, we defined a helper function `plot_compare`, which plots on the same axes the actual closing prices, the predictions from the  $L = 3$  architecture, and the predictions from another chosen architecture  $L \in \{4, 5, 6, 7\}$ . The function sets the labels, legend markers, grid, and axis formatting:

```

1 def plot_compare(L_other, ax, title):
2     ax.plot(table.index, table["Actual Values"], marker="s", label="
        Actual Values")
3     ax.plot(table.index, table["L=3"], marker="o", label="L=3")
4     ax.plot(table.index, table[f"L={L_other}"], marker="D", label="L=
        other")
5     ax.set_xlabel("Date")
6     ax.set_ylabel("Closing price")
7     ax.set_title(title)
8     ax.grid(True, linestyle="--", alpha=0.6)
9     ax.tick_params(axis="x", rotation=45)

```

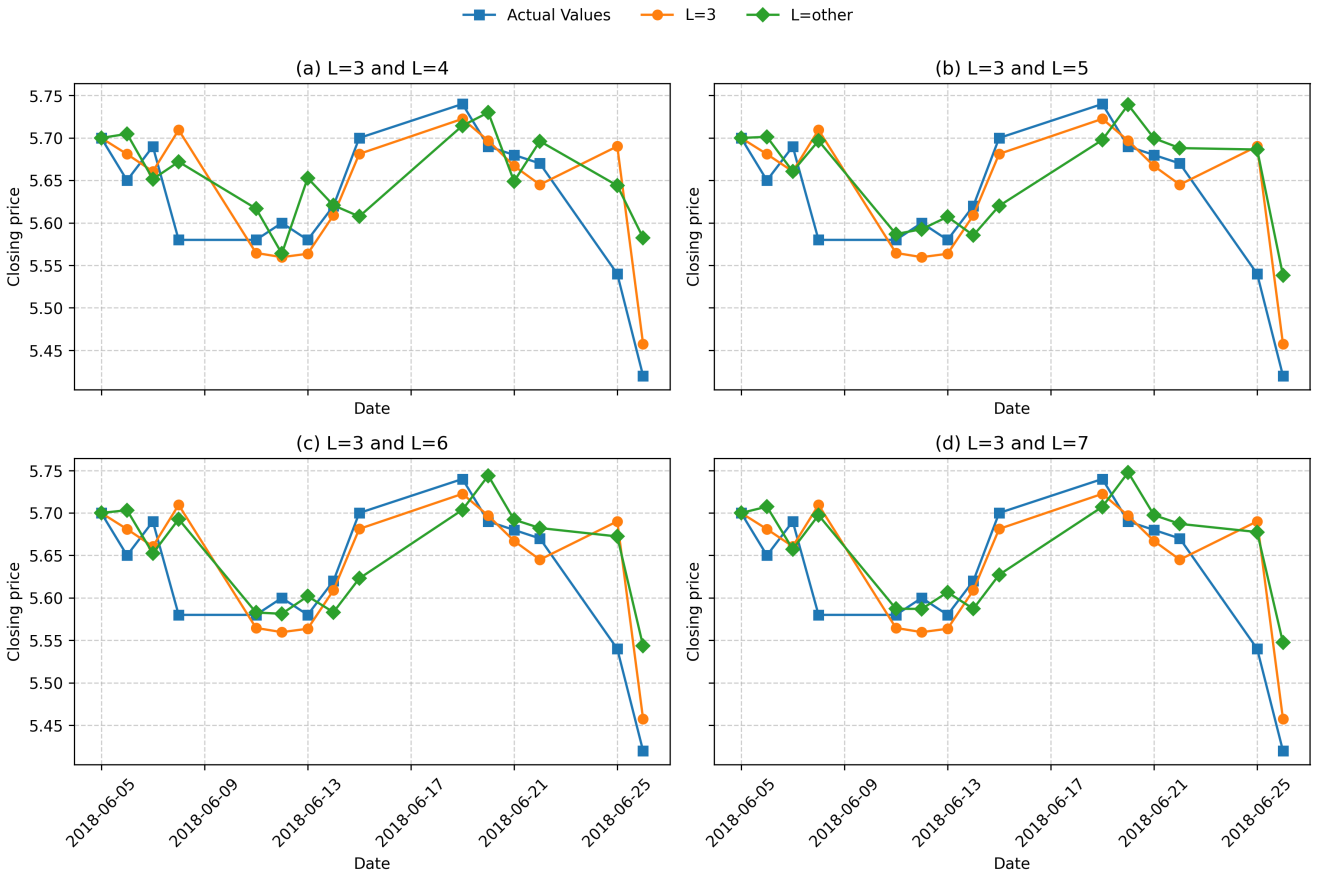


Figure 2.5: Comparison with single hidden layer prediction model and multiple hidden layers prediction model

We then created a  $2 \times 2$  panel of subplots, where each subplot compares the baseline single-hidden-layer model ( $L = 3$ ) against one deeper architecture ( $L = 4$ ,  $L = 5$ ,  $L = 6$ , or  $L = 7$ ). All subplots

share the same axes ranges for consistent comparison. A common legend was placed above the grid, giving us the grid above, which provides a clear side-by-side visualization of how prediction accuracy evolves as the number of hidden layers increases.

## 2.2 Criticisms of the study

**Train/Test split** The dataset is split between 97% training and 3% test, this is not good practice for financial time series given the high variability of the data. The test range for the paper corresponds to 3 weeks which is too short to reliably evaluate a model. The performance may seem good if the market is stable during that period. The consequences of this is that the MAE has high variance and could change significantly if the test window gets shifted by only one week. The better solution would be to have the split at around 70-85% for the train and 15-30% for the test. A better yet solution could be to use the rolling-window evaluation to have an even more fair evaluation.

**Validation set** The absence of the validation set means that hyperparameters and the number of layers or neurons are chosen on based on the test set. The problem with this is that the model may overfit the test data and the accuracy then is not unbiased, this also result in an over optimistic predictive performance and makes doing comparisons unreliable.

**Reproducibility** Most of the key hyperparameters are missing, therefore the paper is not accurately reproducible. Machine learning research must allow others to run the same procedure, using the same data and obtaining comparable results. In this paper this is not possible because it doesn't give us:

1. **Learning rate**, this probably is the most important parameter that we are missing. A different learning rate from the one used by the authors may result in divergence instead of convergence.
2. **Momentum**, about this parameter the only thing said is that it typically ranges between 0.1 and 0.8. Momentum determines whether the model finds a minimum, its training speed and overall its stability.
3. **Starting weights**, weights and biases can change the outcome significantly, expecially in smaller datasets.
4. **Mini Batch size**, these can dramatically impact the learning dynamics of the model.
5. **Random seed**, even if all other parameters were known, the model would still produce different results.

Without these details, the results cannot be validated, which significantly weakens the scientific rigor of the work.

**Performance measures** The paper claims that *"Two hidden layers improve prediction compared to one hidden layer"*, yet the only metric considered is the MAE computed over only 15 test points. The MAE is a valid measure of average error, nevertheless it is insufficient by itself to compute financial forecasting. Metrics such as Root Mean Squared Error (which penalizes more big errors) and  $R^2$  must be used to provide a more complete evaluation of predictive performance. In addition, an important indicator to look out for while computing previsions is the direction of

the market. For this matter computing the directional accuracy can be useful, it measures the percentage of times for which the model predicted the correct direction of the market. A model with low error but wrong direction is useless for practical use.

**Missing output data** Table 1 in the paper shows the actual values of the market sided with the predicted values from each model tested. The table is therefore incomplete as it doesn't give the actual MAE values computed (so, actually, Table 3.3 is not available in the original work). This is directly linked with the issue of having only one error metric, the difference between the actual value and the predicted value doesn't account for the relativity of the error.

# Chapter 3

## Beyond the Paper

In this final chapter, we present several developments we carried out on our own initiative, taking inspiration from the results obtained while attempting to replicate the findings of the paper. The first idea that came to mind was to compare the best-performing MLP model (the one with 8 neurons and 200 epochs) with other classical models used for time series prediction; we anticipate here that it maintained its leading position. Subsequently, we tested the best MLP model on another dataset and compared it with a more complex model to assess whether, even with different data, "simplicity pays off". Finally, we also addressed what we consider to be one of the main issues of the paper, an unbalanced train-test split, and it is therefore interesting to compare the results with those obtained previously.

### 3.1 Comparison with Alternative Models

The best MLP is, overall, a relatively simple model (137 parameters), although even simpler approaches such as ARIMA exist, and of course more complex ones such as LSTMs. In this section, we will therefore compare it with the most renowned statistical time-series predictor (ARIMA), a well-known machine learning predictor (SVM), and the most widely used deep-learning predictor (LSTM). Finally, we will also test a more recent idea involving a hybrid ARIMA-LSTM predictor.

#### 3.1.1 Autoregressive Integrated Moving Average

ARIMA models represent one of the most established statistical approaches for time-series forecasting. They are characterized by three parameters:  $p$ , which specifies the order of the autoregressive component;  $d$ , indicating how many times the series must be differenced to achieve stationarity, and  $q$ , the order of the moving-average component. Together, these elements form the ARIMA( $p, d, q$ ) structure. Formally, an ARIMA model can be expressed through the equation

$$\phi(B)(1 - B)^d y_t = \theta(B)\varepsilon_t \quad (3.1)$$

where the backshift operator  $B$  (such that  $By_t = y_{t-1}$ ) is used to describe temporal dependencies. The polynomials  $\phi(B)$  and  $\theta(B)$  represent the autoregressive and moving-average parts respectively, while  $\varepsilon_t$  denotes a white-noise process. This formulation highlights the flexibility of ARIMA in capturing both persistence through AR terms and random shock propagation through MA terms.

ARIMA models offer several advantages: they are mathematically interpretable, computationally efficient, and work particularly well for linear and (possibly differenced) stationary time series. Their simplicity and solid theoretical foundation have made them a reference point in forecasting for decades. However, they also suffer from some limitations. Being inherently linear models, they

tend to underperform when the data exhibit nonlinear or highly complex dynamics. For datasets with long-term dependencies or intricate temporal structures, ARIMA may not fully capture the underlying patterns, motivating the use of more sophisticated approaches.

By optimizing on the usual dataset and using the standard train-test split, the optimal model obtained was an ARIMA(2,1,2), achieving a test MAE of 0.051435. We can therefore observe that it does not outperform the MLP (which achieved a test MAE of 0.038688), as the latter is able to capture nonlinear relationships. However, the ARIMA model does perform better than all MLP architectures with more than one hidden layer, suggesting that those deeper structures are not well suited to capturing the underlying nonlinearity: they introduce noise, becoming more complex than ARIMA while achieving lower performance. Moreover, in the ARIMA model we rely solely on the lags of the close price, whereas the MLP has access to more information since its input includes the lags of five different series. For this reason, one might consider fitting an ARIMAX model; however, after some experimentation, we noticed that it tends to overfit rather easily. It is therefore preferable to remain with the simpler ARIMA(2,1,2) specification. In the figure below, we can visually observe that the ARIMA model performs reasonably well, although still not as effectively as the MLP.

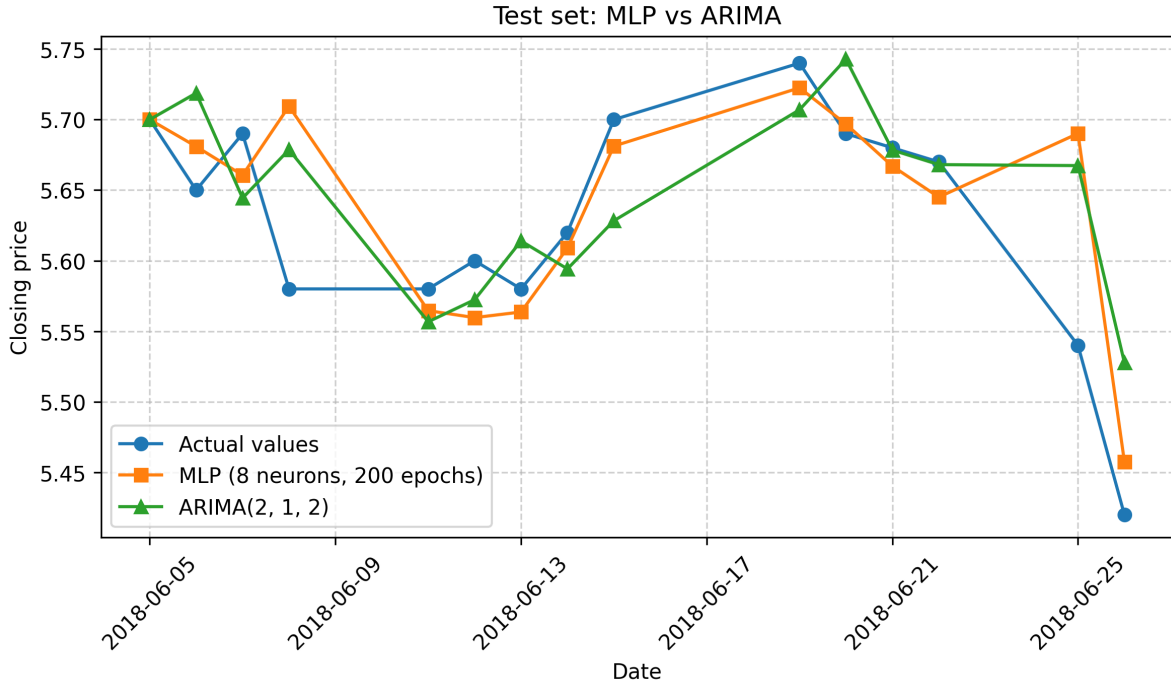


Figure 3.1: ARIMA vs best MLP

### 3.1.2 Support Vector Machine

Support Vector Regression (SVR), the regression counterpart of Support Vector Machines, is based on a few key parameters that govern its flexibility and predictive capacity. One of these is the choice of kernel function (typically linear, polynomial, or radial basis function), which determines how the input data are mapped into a higher-dimensional feature space where linear relationships can be more easily captured. The parameter  $C$  regulates the balance between model complexity and the tolerance to errors: a large value of  $C$  forces the model to fit the training data more closely, whereas a smaller value allows for greater generalization. Another important parameter is  $\epsilon$ , which defines the width of the so-called  $\epsilon$ -insensitive tube within which prediction errors are not penalized. For nonlinear kernels such as the RBF, the parameter  $\gamma$  plays a central role,

controlling the influence radius of individual data points in the transformed feature space. SVR seeks a function

$$f(x) = \langle w, x \rangle + b \quad (3.2)$$

and determines  $w$  and  $b$  by minimizing the objective

$$\frac{1}{2} \|w\|^2 + C \sum_i (\xi_i + \xi_i^*) \quad (3.3)$$

under the constraints that deviations of  $f(x_i)$  from the true targets remain within the  $\epsilon$  margin whenever possible, introducing slack variables  $\xi_i$  and  $\xi_i^*$  only when this tolerance is exceeded.

Among its advantages, SVR is particularly effective for capturing nonlinear patterns through kernel methods, and it often performs well even with limited data, while being relatively resilient to outliers thanks to the  $\epsilon$ -insensitive loss. Nevertheless, it also presents some limitations: models with nonlinear kernels can become computationally expensive, especially for large datasets, and the necessity of tuning multiple hyperparameters ( $C$ ,  $\epsilon$ ,  $\gamma$ ) can make the training process delicate. Moreover, the resulting model may be less interpretable than simpler statistical approaches, especially when complex kernels are involved.

After a simple hyperparameter tuning process, we obtained  $C = 1$ ,  $\epsilon = 0.01$ , and a gamma value set to scale, which led to the worst performance so far, with a test MAE of 0.070463. This result may be related to the discussion in the previous section: attempting to make the model overly sophisticated can hinder its ability to discern which information is truly relevant for prediction. Such behaviour is evidently linked to the nature of the dataset, whose level of complexity is sufficient for a simple single-hidden-layer model to perform well, yet not high enough to justify significantly more advanced architectures than ARIMA. Another possibility, common in the context of SVMs, is that a much more precise and fine-grained hyperparameter tuning strategy would be required to fully exploit the model's potential. As usual, we also report the comparison plot between the predictions.

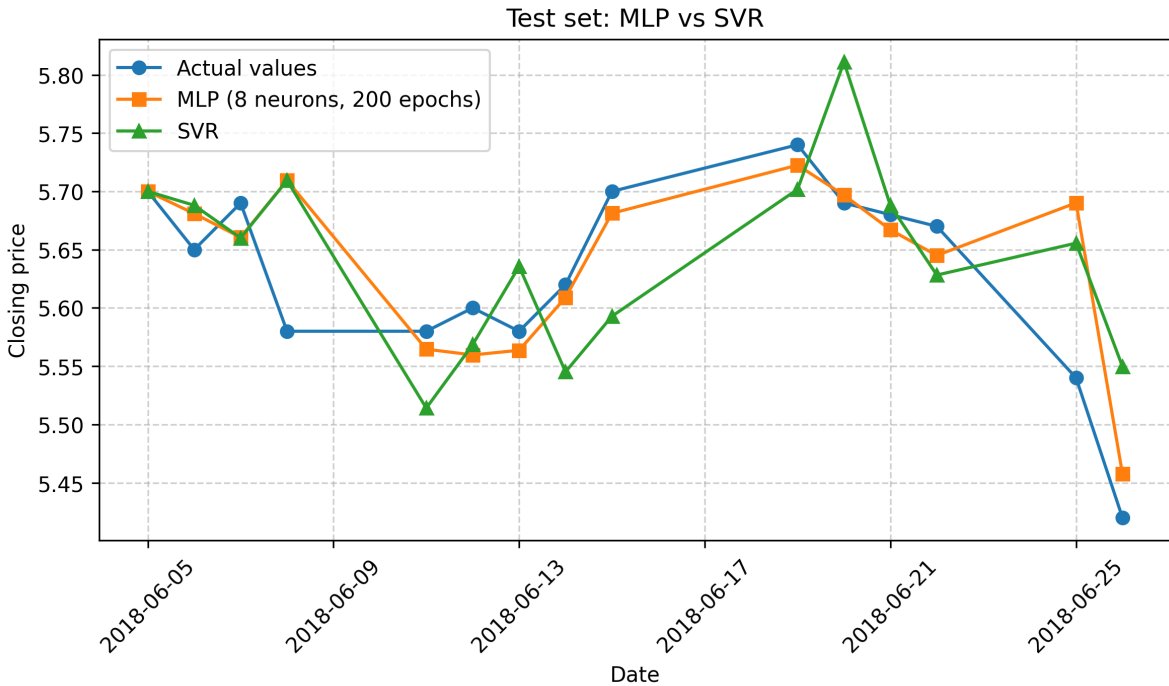


Figure 3.2: SVM vs best MLP

### 3.1.3 Long Short Term Memory

An LSTM is a type of recurrent neural network designed to capture both short-term and long-term dependencies in sequential data. It maintains a cell state  $c_t$ , which represents the long-term memory, and a hidden state  $h_t$ , which represents the short-term memory. The information flow is regulated by three gates:

- **Forget gate:**

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \quad (3.4)$$

It decides which information from the previous cell state  $c_{t-1}$  should be forgotten.

- **Input gate:**

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i), \quad \tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \quad (3.5)$$

It controls which new information is added to the cell state.

- **Cell state update:**

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \quad (3.6)$$

This update preserves long-term memory while incorporating new relevant information.

- **Output gate:**

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o), \quad h_t = o_t \odot \tanh(c_t) \quad (3.7)$$

It determines the short-term memory (hidden state) used for output and future computations.

To train the model, it is first necessary to standardize the input time series. In our case, we used the five OHLCV series with the previous ten lags. Using only three lags in order to be fair with the MLP resulted in worse performance. Once the prediction is obtained (a single scalar corresponding to the close price), an inverse standardization is applied to bring the output back to the original scale. Thanks to the Python `Keras` library, we implemented a simple LSTM architecture, which is shown in Figure 3.3. Dense layers are required to produce the final prediction, and the ReLU activation function was employed. A dropout rate of 50% was introduced to regularize the network and mitigate overfitting. Figure 3.4 also reports the evolution of the training MAE and validation MAE during the learning process.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 10, 64)	17,920
lstm_1 (LSTM)	(None, 64)	33,024
dense (Dense)	(None, 128)	8,320
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 1)	129

**Total params:** 59,393 (232.00 KB)

**Trainable params:** 59,393 (232.00 KB)

**Non-trainable params:** 0 (0.00 B)

Figure 3.3: LSTM model architecture

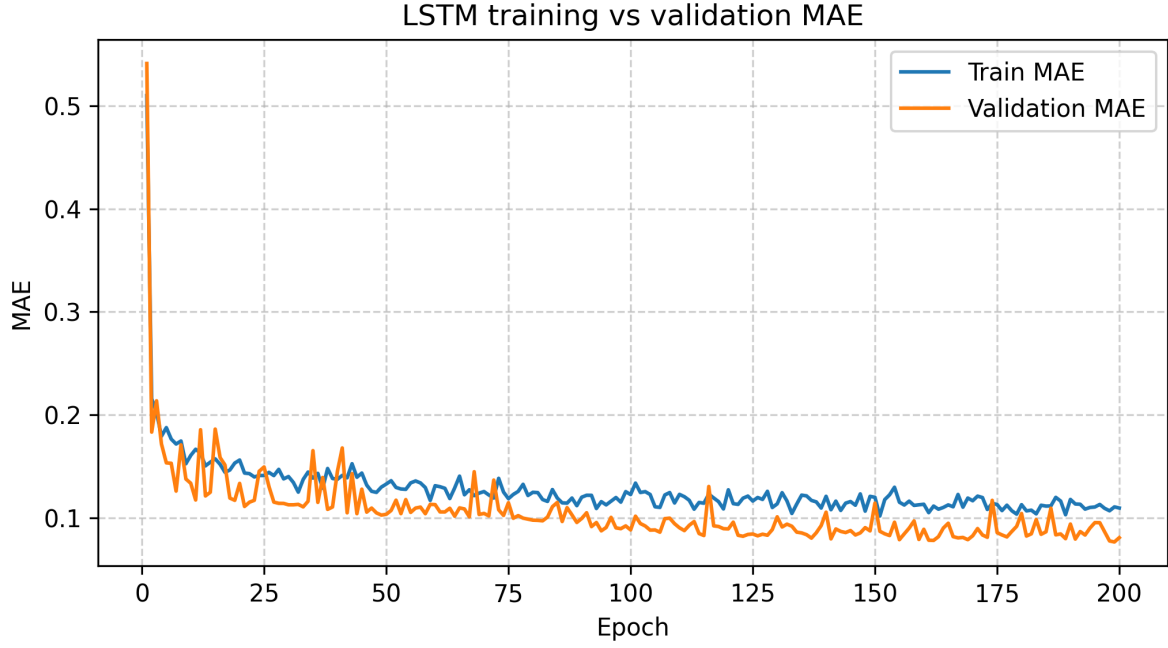


Figure 3.4: Training history of the LSTM model

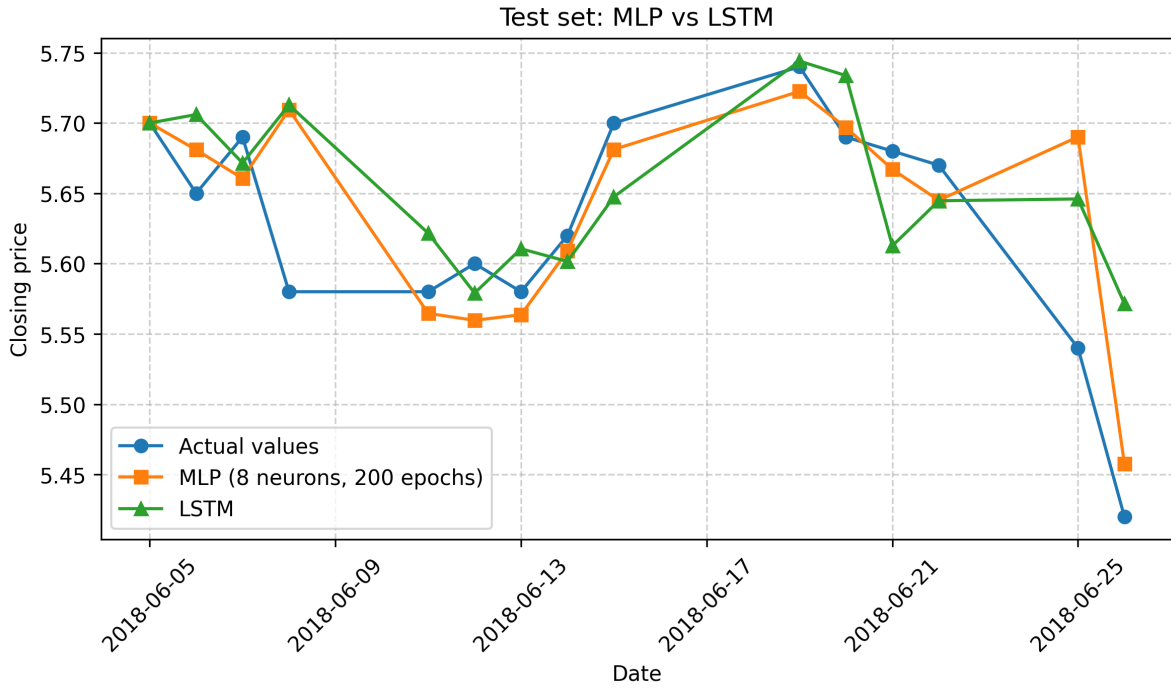


Figure 3.5: LSTM vs best MLP

The number of parameters can be computed explicitly. For instance, in the first LSTM layer, each gate contains  $64 \times 64$  parameters associated with the previous short-term memory,  $5 \times 64$  parameters corresponding to the five OHLCV values at the current lag, and 64 bias terms. Since an LSTM has four gates, this results in a total of 17,920 parameters for the layer. This is undoubtedly the most complex model used so far. Although LSTMs are particularly well suited for time series data, the obtained test MAE of 0.054952 does not outperform even the ARIMA model. We hypothesize that this behavior arises for the same reasons discussed for the SVM: the data are not

sufficiently complex to justify such an advanced model. In this sense, using an LSTM is akin to using a cannon to kill a fly; it is more expensive than a fly swatter, and it does not necessarily guarantee better results.

### 3.1.4 Hybrid ARIMA-LSTM

A more judicious use of an LSTM is suggested in [1]. The idea proposed in the paper is relatively straightforward: an ARIMA model is first fitted on the close price series, and the corresponding residuals are extracted. An LSTM is then trained on this residual series (after appropriate standardization). The final prediction of the close price is obtained by summing the ARIMA forecast with the LSTM prediction of the ARIMA residuals (after inverse standardization). From an intuitive perspective, this approach allows each model to focus on what it does best: the ARIMA model captures simple and linear patterns in the data, while the recurrent neural network is responsible for modeling the remaining complexity that the simpler model fails to explain. Thanks to the synergy between these two models, the hybrid approach achieves better performance than both the standalone LSTM and the standalone ARIMA, with a test MAE of 0.050557. However, it still does not outperform the best-performing MLP.

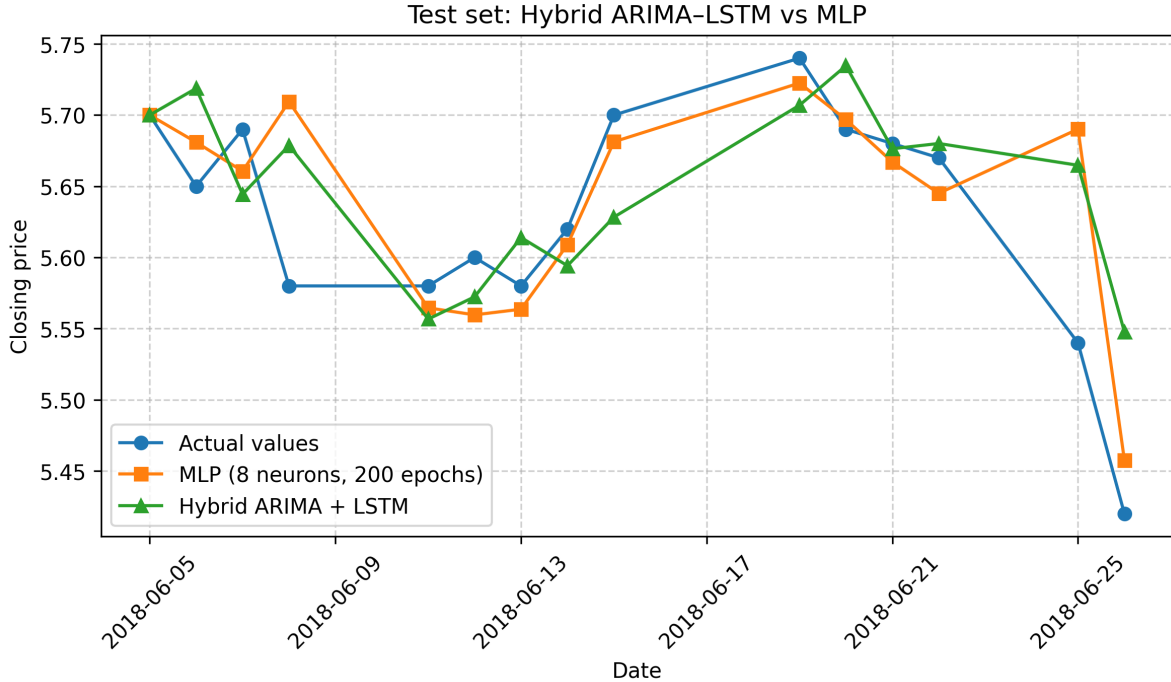


Figure 3.6: Hybrid ARIMA-LSTM vs best MLP

Here the hybrid LSTM is simpler than before (single layer), but even when tested alone, the same LSTM performed worse than in the previous section. Below, we also report a summary table:

Table 3.1: Summary table for all models

Model	MAE	# Parameters	Train time (s)
Best MLP	0.038688	137	1.40
ARIMA	0.051435	5	0.89
SVR	0.070463	16	0.50
LSTM	0.054952	59393	43.75
Hybrid	0.050557	13162	30.75

## 3.2 Alternative Framework

Up to this point, we have extended the analysis by introducing additional models, while keeping the experimental setup adopted in the reference paper unchanged. In this section, we don't add new models, but the experimental framework itself is modified in order to assess how the observations and conclusions drawn in the previous sections are affected by this change.

### 3.2.1 Alternative Dataset

Given the dominance of the single-hidden-layer MLP with 8 neurons trained for 200 epochs over all other models, particularly the more complex ones, we investigated whether this result holds more generally by applying the exact same model to a different dataset. The new data consist of the close price series of Microsoft stock from February 2013 to February 2018. This dataset is approximately twice as large as the previous one and was split using a standard 80-20 train-test partition. On this dataset, an Encoder-Decoder LSTM with an attention layer had previously been fitted, achieving reasonably good performance with a test MAE of 0.657825. Nevertheless, our best-performing MLP again yields slightly superior results, with a test MAE of only 0.613138. As shown in Figure 3.7, which compares the predictions of the two models, the MLP tracks the true values more closely, whereas the Encoder-Decoder architecture appears to be shifted and delayed in capturing trends.

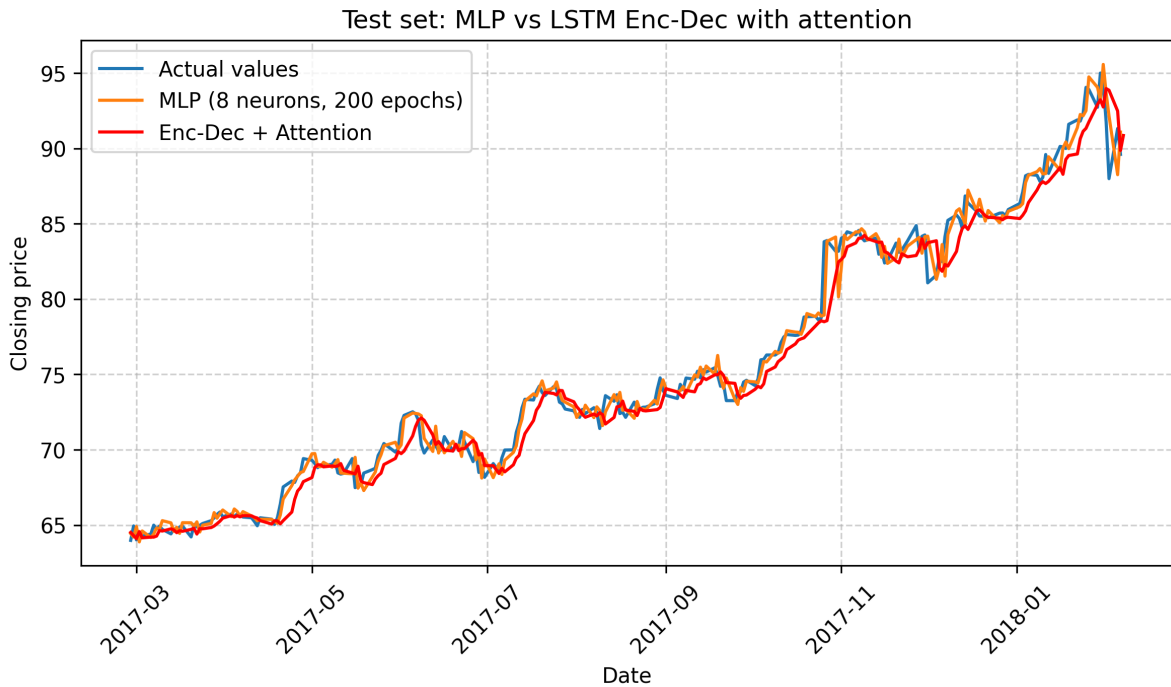


Figure 3.7: Best MLP on Microsoft stock prices

Once again, the "simple is better" philosophy seems to prevail. Although the precise architecture of the Encoder-Decoder model is not of primary interest in this context, it is worth noting that it is highly complex, comprising 50,433 parameters. This observation and philosophy is consistent with the nature of financial time series, which are often heavily dominated by noise.

### 3.2.2 Alternative Train-Test Split

We now return to the ICBC dataset considered in the reference paper, but adopt a standard 80-20 train-test split and rerun the same experimental code. The first notable observation is a reduced variability in the performance of the best single-hidden-layer architecture with respect to changes in the random seed. As shown in Figure 3.8, the optimal configuration corresponds to 4 neurons trained for 1000 epochs. Moreover, the figure is more consistent with the theoretical expectation that increasing the number of training epochs should generally improve parameter estimation. Indeed, for each fixed number of neurons, the MAE obtained at 500 epochs lies strictly between the MAE achieved at 1000 epochs and that obtained at 200 epochs.

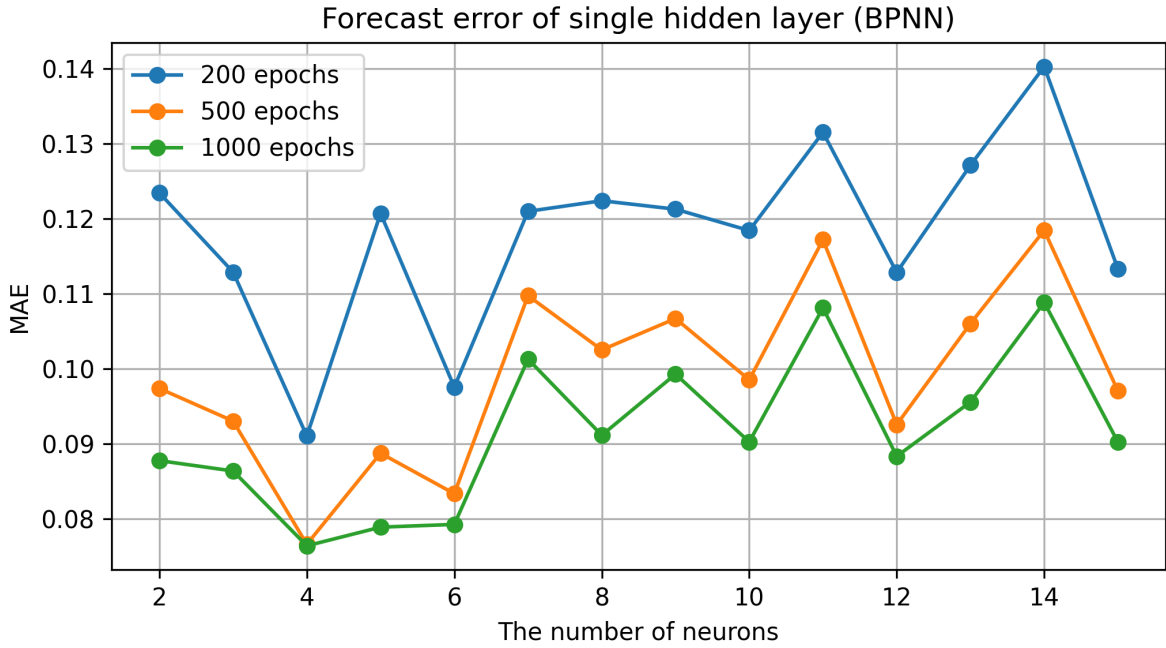


Figure 3.8: MAE on the test set for different neuron counts and epochs (big test)

Table 3.2: MAE on the test set for different neuron counts and epochs (big test)

Neurons	200 Epochs	500 Epochs	1000 Epochs
2	0.123456	0.097380	0.087754
3	0.112891	0.093030	0.086373
4	0.091043	0.076600	<b>0.076400</b>
5	0.120718	0.088743	0.078884
6	0.097560	0.083353	0.079254
7	0.120993	0.109747	0.101285
8	0.122407	0.102519	0.091143
9	0.121280	0.106698	0.099282
10	0.118433	0.098531	0.090250
11	0.131527	0.117201	0.108124
12	0.112851	0.092557	0.088338
13	0.127138	0.106031	0.095574
14	0.140287	0.118453	0.108859
15	0.113325	0.097058	0.090230

It can also be noted that the errors are systematically higher when compared to experiments conducted with a smaller test set. Furthermore, the single-hidden-layer model still outperforms multi-hidden-layer architectures, but with a less decisive margin when compared to models with  $L = 6$  or  $L = 7$ .

Table 3.3: MAE summary table for architectures  $L = 3, \dots, 7$  (big test)

Architecture (L)	MAE
3	0.076400
4	0.087000
5	0.080025
6	0.077778
7	0.077364

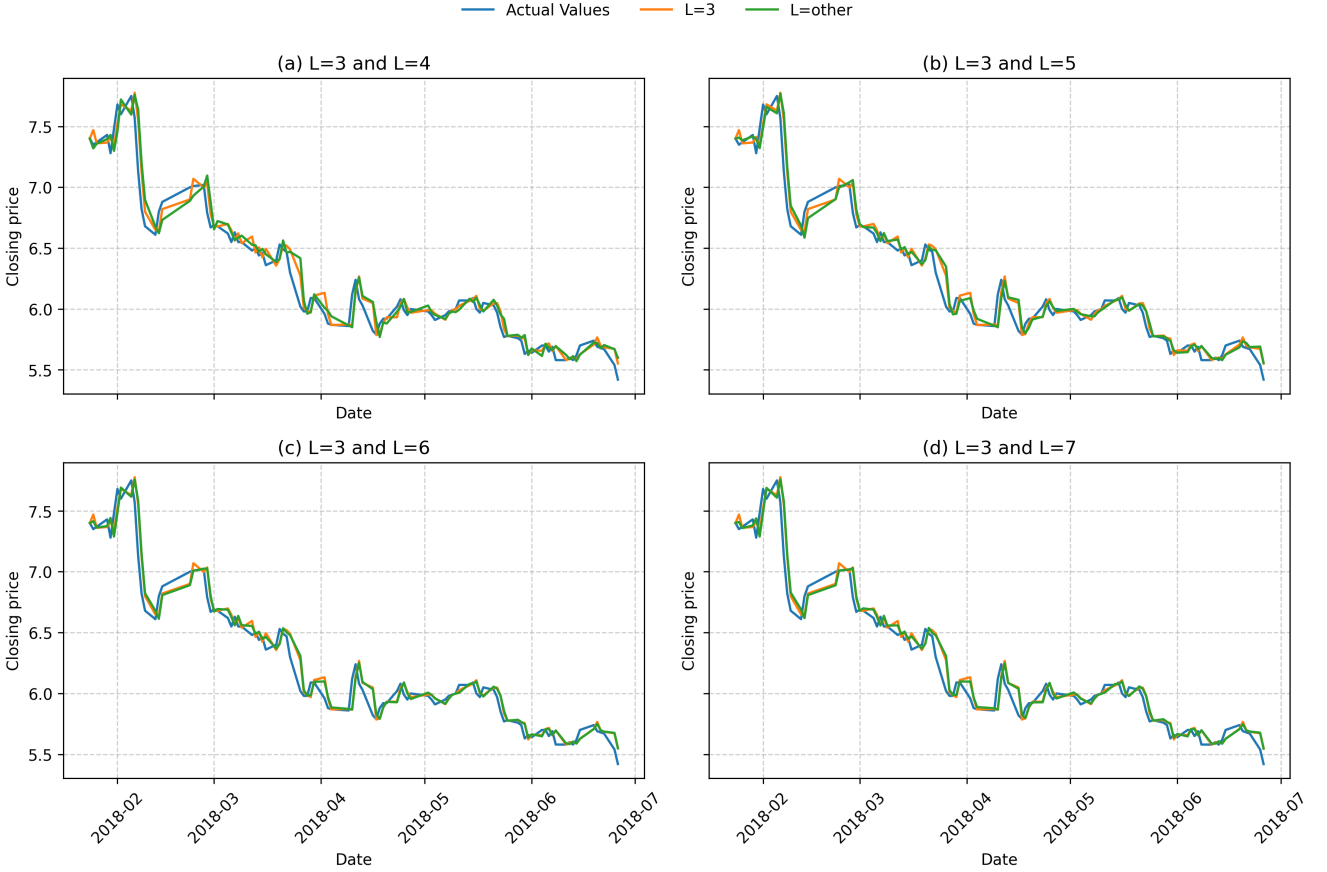


Figure 3.9: Comparison with single hidden layer prediction model and multiple hidden layers prediction model (big test)

We also report in Table 3.4 the updated performance results of the remaining models. It can be observed that SVM and LSTM remain the least suitable approaches for predicting these data. However, in this setting, the best-performing MLP is surprisingly outperformed by the hybrid model. The synergy between ARIMA and an LSTM trained on the residuals therefore emerges as the most effective strategy when a more appropriate train-test split is adopted.

Table 3.4: Summary table for all models (big test)

Model	MAE	# Parameters	Train time (s)
Best MLP	0.076400	69	4.14
ARIMA	0.076319	4	0.88
SVR	0.083854	16	0.49
LSTM	0.186859	59393	39.89
Hybrid	0.075818	13161	30.17

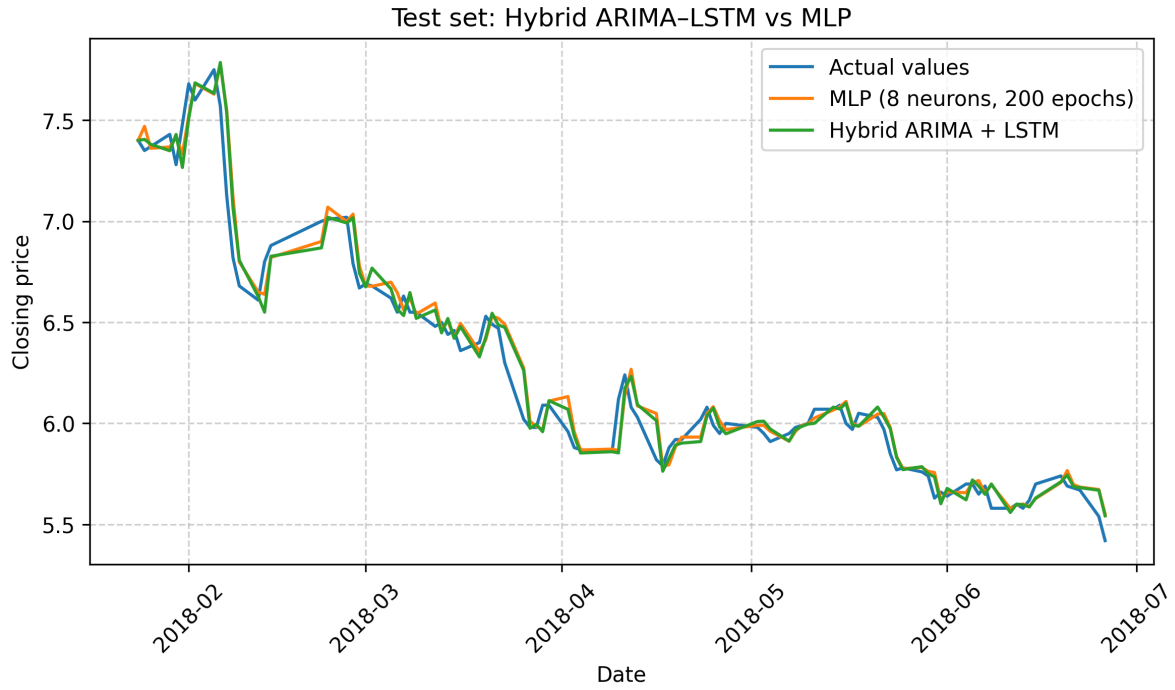


Figure 3.10: Hybrid ARIMA-LSTM vs best MLP (big test)

# Conclusion

This report followed a progressive and structured path aimed at understanding, reproducing, and critically extending an existing study on neural networks for financial time series prediction. We began by introducing the theoretical foundations of neural networks, with particular emphasis on multilayer perceptrons, universal approximation properties, and the backpropagation algorithm. We then presented the reference paper in detail, clarifying its methodology, experimental design, and main conclusions. Subsequently, we attempted to reproduce the results of the paper, highlighting several reproducibility issues and methodological weaknesses, most notably the extremely small test set and the lack of explicit disclosure of the numerical values of several key hyperparameters. Building on this critical analysis, we extended the original framework by introducing additional models, including ARIMA, SVM, LSTM, and a hybrid ARIMA-LSTM approach, and by modifying the experimental setting through alternative datasets and more appropriate train-test splits. This allowed us not only to enrich the original results, but also to directly address some of the limitations identified in the paper.

Several important lessons can be drawn from this analysis. First, the experiments clearly show that using a very small test set can lead to overly optimistic conclusions: in such settings, model performance may appear exceptionally good simply due to favorable randomness, for instance through a particular choice of the random seed. When the test set is enlarged and made more representative, performance becomes more stable but also systematically worse, revealing the true predictive difficulty of the task. Second, for financial time series such as stock prices, simplicity often proves to be a strength rather than a weakness. In this study, a simple single-hidden-layer MLP consistently outperformed deeper MLP architectures and a pure LSTM model, which suffered from overfitting, noise sensitivity, or insufficient data to justify its complexity. At the same time, the hybrid ARIMA-LSTM model emerged as a strong competitor when a more appropriate train-test split was adopted, showing that combining simple linear structure with nonlinear residual modeling can be more effective than relying on a fully complex model alone. Overall, the MLP and the hybrid approach appear to be the most reliable choices in this context.

Finally, it is important to emphasize the intrinsic complexity and relativity of the results obtained. Financial time series are noisy, non-stationary, and highly dependent on both the amount and the nature of the available data. As a consequence, the No Free Lunch theorem is highlighted and there is no universally optimal model that can be expected to perform well across all datasets, time periods, or market conditions. Model performance is always conditional on the data, the evaluation framework, and the assumptions made during training. This reinforces the need for careful experimental design, robust validation strategies, and a critical mindset when interpreting empirical results in financial forecasting.

# References

- [1] A. Adithi. Time series forecasting: A hybrid model for predictive analysis. *Medium*, 2024.
- [2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [3] Thi Bao Tram Ngo. Lecture notes, slides and code on deep learning. ENSIIE, 2025.
- [4] Vincent Runge. Classification supervisée. Course handout, Université d'Évry, 2025.
- [5] Ruey S. Tsay. *Analysis of Financial Time Series*. Wiley Series in Probability and Statistics. John Wiley & Sons, 2nd edition, 2005.
- [6] Peipei Zhang and Chuanhe Shen. Choice of the number of hidden layers for back propagation neural network driven by stock price data and application to price prediction. *Journal of Physics: Conference Series*, 2019.
- [7] Thomas M. Zurada. *Introduction to Artificial Neural Systems*. West Publishing Company, 1992.