Università degli Studi di Milano

MSc. in Data Science for Economics

Statistical Methods for Machine Learning Exam

Gas Fee Prediction: A Deep Learning Approach

Federico Vinci
federico.vinci1@studenti.unimi.it

March 24, 2023

Abstract

The aim of the project is to attempt to forecast the minimum gas price fee accepted by validators so as to get a transaction processed within the Ethereum Blockchain. The forecast would be used as a complementary tool to assist users willing to do operations within the specified network to make wiser choices when it comes to pay the more convenient gas fee. By leveraging 2 LSTM Neural Network architectures, the models will take as input a time series of 600,000 mined blocks, where each block contains the minimum gas price accepted by the validator, to attempt a one-minute and one-hour forecast of the same variable. Data will be object of a preliminary time series analysis that will help to make correct choices during the modeling phase. The trained LSTM architectures will prove to be a good tool when it comes to single-step forecasting, but won't be as efficient in multi-step task. To improve the accuracy of the multi-step model, a *manual* hyper-parameter tuning will be pursued.

Contents

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction

1.1 Background

Ethereum is an open-source, blockchain-based decentralized computing platform that allows developers to build and deploy decentralized applications, known as *d-apps*. Specifically, Ethereum is mostly famous for serving as a development platform for smart contracts. Smart contracts are self-executing contracts that differ from their respective physical form (ie. contracts) by the terms of agreement, which, in the first case, are directly written into code and stored in blocks that build the widely-known blockchain.

One of the virtues of blocks (if not the biggest) is their immutability: essentially, blocks cannot be altered or interfered with by any party once they have been executed. The underlying creed that brings together those who operate within this environment is freedom. Smart contracts are designed to eliminate the need for intermediaries, such as lawyers or banks, in the execution or revision of contractual agreements. Within the Ethereum environment there are plenty of operations that can be done: buying cryptocurrencies (so called *ether*), NFT, Decentralized Finance Tokens and many more. Throughout this project, for the sake of simplicity, the only operation that will be taken into account will be the cryptocurrency exchange, in particular the exchange related to the enforced currency of Ethereum, *ether*.
To avoid compromising the truthfulness and authenticity of the Ethereum environment, specific network nodes known as *miners* are called upon to reach logical consensus on the operations (that form the block itself and are requested by users) through voting on the validity and order of the broadcasted transaction under specific and rigorous protocols (Mars et al. 2021).[4] The past Ethereum validation protocol named *"Proof of Work"* featured miners solving extremely difficult cryptographic puzzle problems before approving operations within a block. As for this, miners received financial reward for carrying the task (ie. *mining a block*) of "approving" all transactions included in the mined block. The financial reward was determined by an auction process that saw miners selecting those transactions that suit their needs not only in terms of balancing out the huge electricity expenses that would derive from automating the approval operations, but also for profit purposes. The fees required for executing transactions in the network are known as *gas fees* and will be explained later in further detail. Note that the "reward and profit" intuition will be fundamental for understanding the underlying need that this paper tries to solve.

Since the end of 2022, Ethereum has switched to an alternative consensus algorithm named *Proof of Stake*. It works by allowing chosen network participants (ie. ex-miners) to "stake" their cryptocurrency holdings as collateral in order to validate transactions and create new blocks on the chain. The amount of stake a participant has is proportional to their chances of being chosen to validate the next block. Within this framework, the role of miners in the traditional

sense of Proof of Work has been replaced by validators. Similarly, these figures are once again responsible for validating transactions, creating new blocks, and adding them to the blockchain. Validators are selected based on the amount of cryptocurrency they have staked as collateral on the network. The more cryptocurrency a validator has staked, the more likely they are to be selected to validate the next block. Despite the switch of consensus algorithms, the reward criteria have not changed: Likewise, Proof of Stake adopts a *gas*-based metering approach where the fee value (the economic reward given to validators) is not fixed and works like an auction mechanism (Mars et al. 2021),[4] resulting in a microeconomic scenario where, on one hand, demanders try to minimize fees, on the other hand, suppliers (ie. validators) try to maximize their reward but at the same time try to make profit out of it. Before each operation is executed, demanders are to set two specific parameters that will determine the "priority" of the request to miners: **gas price** and **gas limit**.

The former is the amount of **ether** (Ethereum crypto-currency) that a user is willing to pay per unit of gas consumed during a transaction. It represents "the fee" that a user is willing to pay to validators for processing their transaction. Gas price is denominated in **Gwei** (1 Gwei = 0,00000000116 eth = 0,00000173 € at current prices). The gas price is set by the user who initiates the transaction, and it determines how quickly the transaction is processed. As stated, validators prioritize transactions with higher gas prices, as they stand to earn more ether by processing those transactions. Setting the appropriate gas price for a transaction can be a balancing act between paying enough to have the transaction processed quickly, but not overpaying and wasting ether unnecessarily (Mars et al. 2021).[4]

The latter refers to the maximum amount of gas (measured in gas price, Gwei units) that a user is willing to spend on a particular transaction. Gas limit is set by the user who initiates the transaction and represents the maximum amount of gas that can be consumed during the transaction. Setting the appropriate gas limit for a transaction is important because it determines the maximum amount of ether that a user can potentially spend on the transaction. If the gas limit is set too low, the transaction may not be completed successfully and the gas used up to that point will be lost. If the gas limit is set too high, the user may end up paying more than necessary for the transaction.

This example may help to clarify: Let's say A wants to send 1 eth to B on the Ethereum network. To do this, A initiates a transaction on the Ethereum network and specifies that he is willing to pay a gas price of 50 Gwei for every unit of gas consumed during the transaction. Now, let's say that A arbitrarily sets the gas limit for this transaction to be at 100,000 units of gas. This means that A is willing to spend up to 100,000 units of gas to execute this transaction. The gas limit is set by A and represents the maximum amount of gas A is willing to spend on this transaction. Based on the gas price A set, the total gas fee for this transaction would be calculated as follows:

$$GF = GP \cdot GL$$
$$GF = 50\ Gwei \cdot 100,000$$
$$GF = 5,000,000\ Gwei = 0.005\ eth = 7,38€$$

where GF stands for Gas Fee, GP for Gas Price and GL for Gas Limit.

Gas Fee might be considered as an endogenous microeconomic variable, as it could also be affected by other variables in its environment. More specifically, it could be influenced by:

- **Network congestion**: When there are many transactions being sent to the Ethereum network, the demand for block space increases, which can drive up the gas price.
- **Ethereum price**: The price of Ether can also affect the gas price. When the price of Ether is high, users may be willing to pay more for transaction processing.
- **User behavior**: Finally, user behavior can also affect the gas price. For example, some users may be willing to pay a premium to participate in a popular decentralized application, which can drive up the gas price for transactions related to that d-app.
- **Gas limit**: Each block on the Ethereum network has a limit on the amount of gas that can be used. If many users are trying to use the maximum amount of gas, the gas price may increase.
- **Block time**: The time it takes to mine a block on the Ethereum network can affect the gas price. If blocks are being mined quickly, users may be willing to pay more for faster transaction processing.

During the "Gas Price Auction Process", a validator chooses which transactions to include in the next block based on the gas price offered by users. As highlighted earlier, validators prefer transactions with higher gas prices because they will earn more fees for including them in the block. It is therefore crucial for the user to adjust the minimum gas price to bid so as to have her/his transaction processed by the validator and not spend an excessive amount of money. This would potentially lead to a significant accumulation of saved Ethereum.

1.2 Scope of the Project

The project attempts to forecast minute by minute what is likely going to be the minimum gas price accepted by the validator in the next mining pool by leveraging Neural Network architectures based on Long-Short Term Memory Layers. The proposed forecasting approaches will be two:

- **The Single-Step approach** will attempt to forecast the $t+1$ minimum gas price by taking as input the last $t-n$ observations (window size).
- **The Multi-Step approach** will attempt to forecast each $t+m$ minimum gas price by taking as input the last $t-n$ observations (window size).

2 Data

2.1 Dataset

Once validated, transactions are then stored in the blockchain. `web3py` Python library helps to access blockchain for accessing data. As for this task, `getBlocks.py` script has been created: it displays functions that take as input the start block and end block that a user wants to retrieve along with all the features of interests: which in this case are:

- `gas`: time series data about each block's minimum gas price approved by the validator (measured in GWei).
- `price`: time series data about the price of Ethereum currency at the mining moment.
- `n_txn`: the number of transactions approved per block.

The final dataset contained 600,000 observations, which were extracted block-wise and therefore not time-uniformly. In order to establish temporal order, a resampling process was being conducted. This resampling involved aggregating the data at one-minute intervals using mean as an aggregation method. Additionally, the gas feature was originally denominated in Wei: a conversion was being carried out to convert it to GWei unit. Two screenshots of the `df.head()` shows how the data appeared before and after the corrections were implemented. For the sake of computational expenses, only the gas feature will be utilized for forecasting purposes.

dt	n_txn	eth	gas
2022-11-05 00:40:47	140	1647.492702	1.050702e+10
2022-11-05 00:40:59	133	1647.493372	1.031394e+10
2022-11-05 00:41:11	133	1647.493372	1.156523e+10
2022-11-05 00:41:23	120	1647.504291	1.130654e+10
2022-11-05 00:41:35	129	1646.815941	1.118033e+10

(a) before pre-processing

dt	gas
2022-11-05 00:40:00	10.410478
2022-11-05 00:41:00	11.395897
2022-11-05 00:42:00	11.727410
2022-11-05 00:43:00	10.569524
2022-11-05 00:44:00	11.760463

(b) after pre-processing

Figure 1: Dataset before and after pre-processing

3 Preliminary Approach

3.1 Train-Test Split

It is generally not recommended to involve the test set during the pre-processing phase because of data leakage, which can lead to over-optimistic performance estimates on the test set: as for this reason, a train-test split (80%, 20% respectively) is being initially executed. An additional 20% validation set is kept aside from the training set and utilized during the training (*fitting*) phase. Thus, the following operations will involve only the train part of the dataframe.

3.2 Single-Step Forecasting and Multi-Step Forecasting

When dealing with time series forecasting, it is essential to have a clear understanding of the desired prediction horizon. This is especially important because the further into the future you want to predict, the greater the uncertainty becomes. As Brownlee[2] (2018) suggests, there are two methods for time series forecasting:

- **Single-Step Forecasting**: given $t - n$, predict the value at $t + 1$
- **Multi-Step Forecasting**: given $t - n$, predict the value at $t + m$

Where t refers to the present, $t + 1$ refers to one time unit ahead and n, m refers to arbitrary parameters.

Once decided which approach to take, it is essential to ensure that the data can be effectively subsetted to correctly feed the neural network. One valid and well-known approach is called *windowing*. Windowing is a method that refers to the technique of splitting a time series dataset into smaller, fixed-size windows or subsets. Each window contains a sequence of contiguous data points

from the time series. Windowing is used to create a *rolling forecast*, where a model is trained on a window of data, and then used to make a prediction for the next data point (single or multiple) in the sequence. The window is then moved forward by one step, and the process is repeated for the next window of data. Windowing is useful because it allows a model to learn patterns in the time series at different scales and resolutions, and can be helpful to capture both short-term and long-term trends in the data. It also helps to overcome the issue of temporal dependence, where observations close together in time are more likely to be correlated than observations that are further apart.

The function `split_sequence` helps to do the job in the case of Single-Step Forecasting: It takes as input the time series and the number of steps (n) desired for splitting and it splits the time series into:

- `X_train`: timely-disposed `np.array` (containing `np.arrays` of "slided n window" elements).
- `y_train`: `np.array` containing the $n+1$ observations for each t in time, one for each sliding window element.

The following image representing a windowed time series with number of steps = 5 may help clarify what is inside the `X_train`:

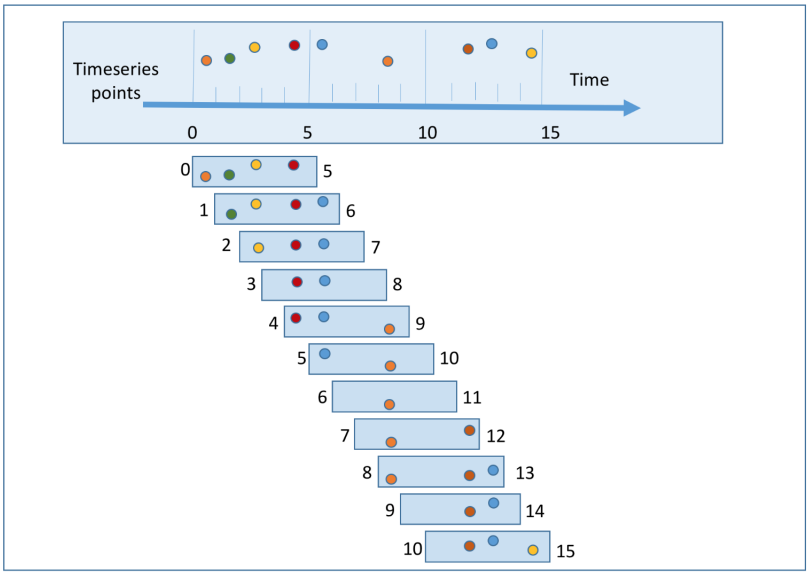

Figure 2: *Windowing* a Time Series, $n = 5$

As for the `y_train`, each observation is the $n+1$ observation for each time t. By creating a different function, windowing has also been applied for multi-step forecasting purposes, but, in this case, the two functions differ in the *y_train*

where the y is an array object which contains, for each time t different arrays of length m.

3.3 Evaluation Metrics

Tasks that involve numerical predictions (such as the forecast task) need specific metrics that differ from the ones utilized for classification tasks. Reliable and broadly diffused metrics are:

- **Mean Absolute Error (MAE)** It measures the average absolute difference between the actual and predicted values. Generally, it provides a measure of the average magnitude of the errors in the predictions.
- **Mean Squared Error (MSE)** It measures the average squared difference between the actual and predicted values.
- **Root Mean Squared Error (RMSE)** It is the square root of the MSE, and it provides a measure of the average size of the errors in the predictions in the same units as the original data.
- **Mean Absolute Percentage Error (MAPE)** It measures the percentage difference between the actual and predicted values. It provides a measure of the accuracy of the predictions relative to the actual values.

The mentioned measures play different roles at various stages of the project. During the training phase, the preferred loss function is often the Mean Squared Error (MSE), because it assigns higher penalties to predictions that deviate significantly from the actual value, which is important during training phases. However, when it comes on evaluating different models, metrics such as Root Mean Squared Error (RMSE) or Mean Absolute Percentage Error (MAPE) are more suitable. The reason is because these metrics do not take into account values under specific exponential form (MSE) which can be misleading when evaluating models. Contrairly, MAPE can result intuitive as it takes into account the average percentage prediction deviation from the actual values. By analyzing these metrics, we can better understand the accuracy of the model and select the best one for our specific goal.

3.4 Time Series Analysis

Gas prices can experience frequent peaks due to fluctuations in the demand for network resources. When demand for transactions or smart contract execution is high, gas prices tend to rise accordingly. These peaks can create outliers in data sets that need to be addressed during the pre-processing phase.
Here's a plot of the pre-processed data. A cut-off method has been applied to trim all those observations that were considered outliers, such as gas prices higher than 300.

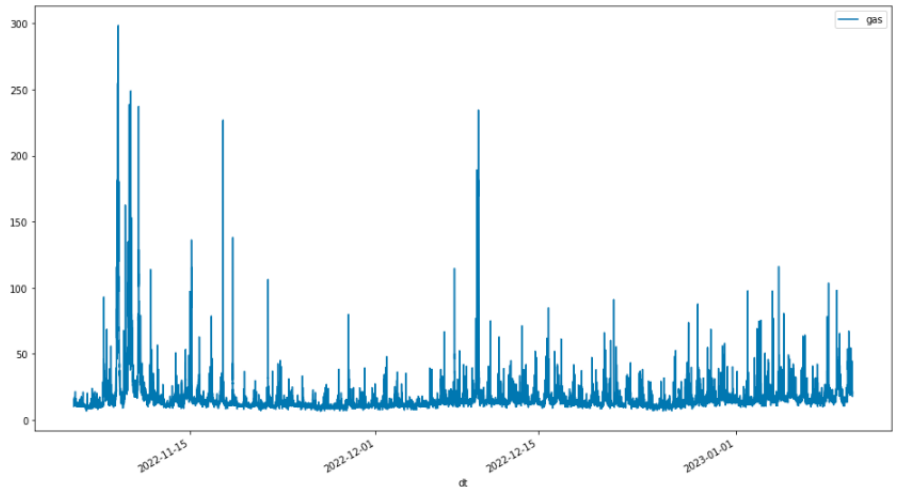

Figure 3: A plot of `gas` feature after pre-processing.

In the field of Time Series Analysis, an essential task that should be undertaken is to make the data stationary. Although, Brownlee[2] (2018) claims that:

> *"It is good practice to manually identify and remove systematic structures from time series data to make the problem easier to model (e.g. make the series stationary), and this may still be a best practice when using recurrent neural networks. But, the general capability of these networks suggests that this may not be a requirement for a skillful model. Technically, the available context of the sequence provided as input may allow neural network models to learn both trend and seasonality directly."*

Nevertheless, for the sake of information, a Dickey-Fuller test will be executed on the data to check whether they are stationary. The `check_stationarity` function will be used.

```
ADF Statistic: -17.284
p-value: 0.000
Critical Values:
        1%: -3.430
        5%: -2.862
        10%: -2.567
The time series is stationary.
```

Figure 4: A plot of `check_stationarity` function output.

The function sets up a null hypothesis that the time series has a unit root, hence it is non-stationary. Since the p-value is minor than 0.05 we can reject the null hypothesis at 95% interval by claiming that the time series does not

have a unit root.

Autocorrelation is a measure of the correlation between observations of a time series that are separated by a certain time lag. In time series analysis, autocorrelation can be used to improve the windowing process by helping to select a suitable window size. If there is high autocorrelation between observations separated by a certain time lag, it indicates that the time series has a strong pattern or trend that repeats itself over that time interval. In this case, using a larger window size may capture more of the underlying pattern and improve the accuracy of the forecast.

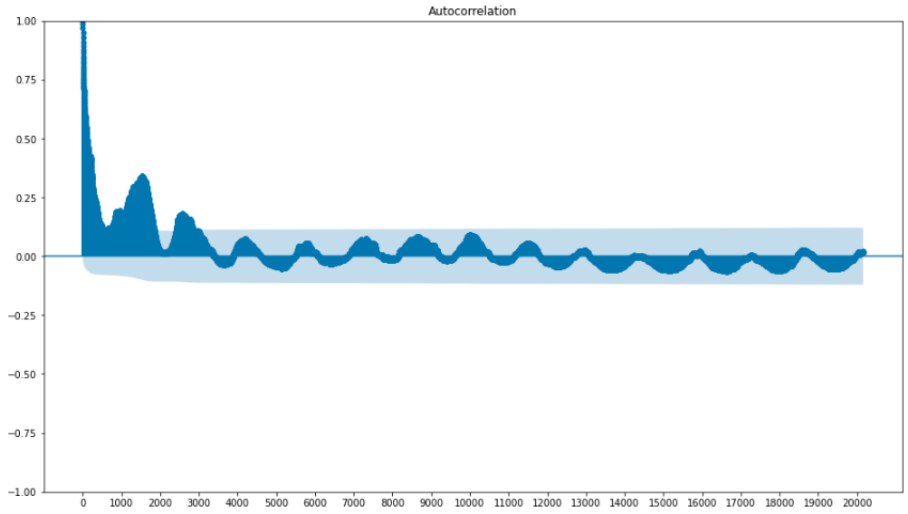

Figure 5: Autocorrelation Plot

By observing the `plot_acf` by `statsmodels` library over 2 weeks of data (expressed in minutes along the `x-axis)` we can clearly see that there's significant autocorrelation up to 2800 minutes of data which is approximately corresponding to two days. This information would be useful when selecting the right window length both for the single-step and multi-step approach.

When dealing with Time Series, it is also useful to take a look at the principal components that build up the time series under the hood: trend, seasonal and residual components.

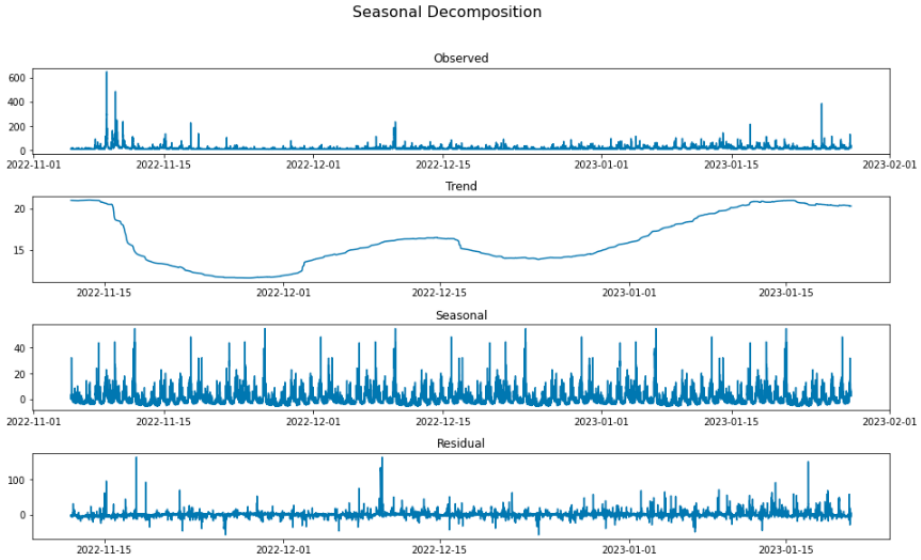

Figure 6: Time Series Decomposition

The picture above displays 4 graphs that show the components of the time series. The trend graph remarks the absence of any form of long period trend (stationary data). Similarly, the seasonal component graph underlines the presence of seasonality within its component: when it comes to forecasting, seasonality can be an helpful indicator for model performance.

4 Modeling Approach

In the following section, neural network architectures will be deployed for time series forecasting tasks using Long-Short Term Memory layers as the main type of layers along with, if needed, densely connected layers. The library `Tensorflow` will be used for such task.

4.1 LSTM Layers

LSTM layers are a type of recurrent neural network (RNN) layer that utilize gated units to selectively control the flow of information in the network. Each LSTM unit has an internal memory cell, which can retain information for a reasonably long period of time. This makes them particularly effective for time series forecasting. The internal memory cell consists of three gating mechanisms, which are implemented as sigmoid and tanh layers, that regulate the information flow into and out of the memory cell (Hochreiter and Schmidhuber, 1997).[3] The three gates in an LSTM unit are as follows:

- **Forget Gate**: determines which information to discard from the memory cell based on the current input and the previous output.
- **Input Gate**: controls the input of new information into the memory cell by selectively updating the cell state.
- **Output Gate**: determines which information to output to the next layer or to the final prediction based on the current input and the previous output.

The initial step is to input the time series data into the model, which is typically divided into input sequences (e.g., past 10 time steps) and target sequences (e.g., next time step(s)). Once the input data is provided, the first gate that the data encounters is the input gate, which is responsible for deciding which information from the input sequence should be passed onto the next step. This is accomplished by using a weight matrix and a bias term to multiply the input sequence and passing the result through a sigmoid activation function.

The forget gate, on the other hand, is responsible for determining which information from the previous time step should be retained or forgotten. To achieve this, the previous hidden state is multiplied by a weight matrix and a bias term and then passed through an activation function. Meanwhile, the output gate decides which information from the memory cell should be passed onto the next step. This is accomplished by multiplying, once again, the memory cell with a weight matrix and a bias term and then passing the result through a sigmoid activation function.

The hidden state, which is the output of the LSTM cell, is then passed on to the next time step. This is accomplished by multiplying the output of the output gate with the output of the memory cell and passing the result through a tanh activation function. To generate a forecasted value for the next time step, the hidden state is passed through an output layer in the final step of LSTM forecasting. This process is repeated for each time step in the input sequence to produce a sequence of forecasted values. The LSTM model's weights and biases are learned through a training process that minimizes the difference between the predicted values and the actual values in the training data (Hochreiter and Schmidhuber, 1997).

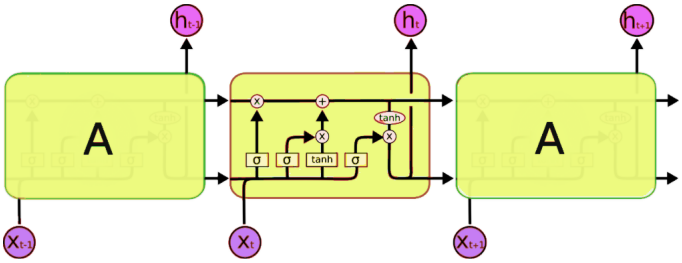

Figure 7: Overview of LSTM Layer[1]

In the image above, the repeating module in an LSTM containing four interacting layers is displayed.

4.2 Modeling Approach

When dealing with modeling neural network architectures, although it is a good practice to experiment with the high number of hyperparameters that orbit around the architecture, it is erroneously common to strive to find the optimal hyperparameter combinations (ie. the combination of layers, units per layer, loss function, optimizer, dropout probability, learning rate, decay, activation function etc.) in order to wish to find the global optimum. Instead, it is more recommended to gradually seek local minimums that might work for the case and then search those hyperparameters that might help to improve the local minimum. Additionally, time series forecasting requires another *parameter* to be taken into account, namely the window span element. The task of finding multiple local optimums thus becomes increasingly harder, but, more relevantly, extremely time consuming and computationally infeasible. Nonetheless, parsimoniousness and theoretical knowledge can help a lot so as to set a method to go with. Instead of creating several LSTM architectures (one for each desired window span), only two architectures were created whose structure would seamlessly be embedded with prefixed window span elements (at least theoretically). It is therefore relevant to define which window length of interest.

Single-Step Forecasting:

- 1 day window span element (1440 minute-wise gas minimum price observations) and next minute as input.
- 1 hour window span element (60 minute-wise gas minimum price observations) and next minute as input.
- 3 minute window span element (3 minute-wise gas minimum price observations) and next minute as input.

Multi-Step Forecasting:

- 1 day window span element (1440 minute-wise gas minimum price observations) and next 60 minute window span element as input.
- 1 hour window span element (60 minute-wise gas minimum price observations) and next 60 minute window span element as input.

The choice of this recursive window-span structure derives from empirical evidence. For example, the first two window-spans comprise 1440 minute observations: the autocorrelation function displayed earlier showed a relatively high degree of autocorrelation (up to 30-40%) with the past day observation along with little less seasonality. Perhaps including such a wide window size can help the LSTM layer to better detect one day ahead's forecast. This will be mostly relevant (at least theoretically) when dealing with multi step forecasting. Similarly, the 1 hour window-span, along with the 3 minute forecast (to some extent), can help the model to capture hourly-wise patterns that could help the forecast on the short run (1 minute ahead forecast) or medium run (10 minute or more forecast, but not in this project). Recall that the autocorrelation was high (>60%) within the first hour: this can certainly help in the short-medium run. Ultimately, by thinking computationally-wise, it is also fundamental to be able to obtain predictions in a relatively short amount of time: for sure the lower the input is, the faster the prediction will be.
Amongst the numerous layers combinations that neural networks can engage on, mainly two architectures have been created for the scope of this project:

"Vanilla" LSTM Architecture:

- LSTM layer with 32 units, 'tanh' activation function.
- Dropout Layer with 0.1 rate
- Dense Layer, 1 densely connected output.

Adam Optimizer, learning rate at 0.1%.

"Deep" LSTM Architecture

- LSTM layer with 64 units, 'tanh' activation function.
- Dropout Layer with 0.1 Rate
- LSTM layer with 32 units, 'tanh' activation function.
- Dropout Layer with 0.1 Rate
- LSTM layer with 16 units, 'tanh' activation function.
- Dropout Layer with 0.1 Rate
- Dense Layer with 32 units, 'tanh' activation function.
- Dense Layer with 1 output unit. 'tanh' activation function.

Adam Optimizer, learning rate 0.01%

Once created the two architectures, the function `single_compile_fit`, will handle the fitting phase of the models. The function will take as input a model of the class `Sequential`, a `X_train np.array` of elements reshaped with the desired window length followed by a `y_train` and a `checkpoint_path`, a `path` to which the fitting will be saved in case of the necessity of interrupting the training phase. The `single_compile_fit` also includes a `model.compile` function set with MSE loss function and Adam Optimizer. The `batch_size` will be set to 32.

The primary difference between LSTM models with different numbers of layers lies in their ability to capture more complex patterns and dependencies in the input sequence. LSTM models with more layers have a greater capacity to learn complex relationships between the input and output sequences, and therefore those might be able to make more accurate predictions on complex time series data or other sequential data. In other words, increasing the number of layers can increase the model's ability to extract higher-level features from the input data. However, there are also trade-offs to consider when increasing the number of layers in an LSTM model. Deeper LSTM models are more computationally expensive to train and can be more prone to overfitting. Additionally, deeper models may require more training data. Therefore, the decision of how many layers to use in an LSTM model depends on the complexity of the problem, the amount of available training data, and the computational resources available for training the model. In practice, it is often necessary to experiment with different numbers of layers to find the optimal configuration that achieves the best trade-off between model complexity, accuracy, and computational efficiency. For obvious reasons, only two models were provided to observe how these would behave with different window-spans of data. In conclusion, a final manual hyperparameter tuning will attempt to improve the best model's performance by editing the architectural structure of the underlying neural network following a theoretical methodology.

5 Empirical Results

This section will be entirely dedicated to assessing the training results of the Neural Network Architectures.

5.1 Single Step Forecasting Results

As expected, the results obtained for single steps forecasting were satisfactory. In fact, when forecasting, it is generally not hard to predict next time step value as compared to the multi-step approach. Here's a summary table that might help to understand the results. These are obtained by getting the last epoch's value for each metric and fit. On top, the metrics of different window span fittings on the Vanilla Architecture are being shown. Likewise, on bottom,

the deep architecture. As mentioned earlier, when selecting the best model a trade off between training MSE and validation MSE should be taken into account, although more weight towards the validation MSE should be given. While doing so, it is also important not to underestimate the importance of the other metrics.

Vanilla	1440_1	60_1	3_1
Train MSE	23.92	26.88	22.75
Train MAE	1.24	1.31	1.21
Train MAPE	6.26	6.76	6.13
Validation MSE	3.41	3.50	4.39
Validation MAE	1.02	1.02	1.18
Validation MAPE	5.39	5.19	5.84

(a) "Vanilla" Architecture Results

Deep	1440_1	60_1	3_1
Train MSE	35.34	33.75	30.75
Train MAE	1.93	1.53	1.52
Train MAPE	8.21	7.66	7.62
Validation MSE	5.86	4.40	4.83
Validation MAE	1.78	1.22	1.22
Validation MAPE	7.09	6.02	5.82

(b) "Deep" Architecture Results

Figure 8: Single-step Architectures' Fitting

On average, the models' predictions are around 5-7% off compared to the actual values, which results in models that are generally able to predict the next-minute value. The other featured metrics also have optimistic results. Nevertheless, when considering the best model, it is relevant to remark that computational speed needs to be taken into consideration, especially in a real-time scenario where a user has to check rapidly what the next minimum gas price will be. If a faster model is to be selected, then a light Vanilla model has to be chosen. Anyways, for the Vanilla architecture, the 1440_1 (1440-minute input, 1-minute output) will be selected. Apparently, the window size magnitude compensates for the lack of complexity of the neural network, which is likely able to learn what the next time step is going to be.

Moving onto the Deep Architecture, the performance was not as good as the precedent one. It seems as if the deeper architecture, at least in the training set, is not able to entirely capture (or instead *overcapturing*) the hidden temporal relationships that lay between different observations. This phenomenon

can be identified in the train MSE, which is on average 10 units higher than the previous one. This can be seen on longer window sizes like 1440_1. However, it seems that the Validation MSE is not as bad as expected from the Train MSE. Without any doubt, the best Deep model is the 3_1. Although its Validation MSE is not the best one amongst the three, the 3_1 model behaves better when it comes to Train MSE. At the same time, this model is able to generalize well also on the validation set by scoring an MSE of 4.83. The MSE, MAE, and MAPS are also well above the acceptance levels. Despite the input window size might seem too small to correctly predict the future, this "penalization" is being compensated by the complexity of the neural network that is able to capture most of the hidden features. Given the importance of rapid prediction in real-time scenarios, the Vanilla Model 3_1 is considered the best option for single-step forecasting (as well as being the model which possesses the best metrics).

Two graphs were examined to take a look at both the architectures' training histories.

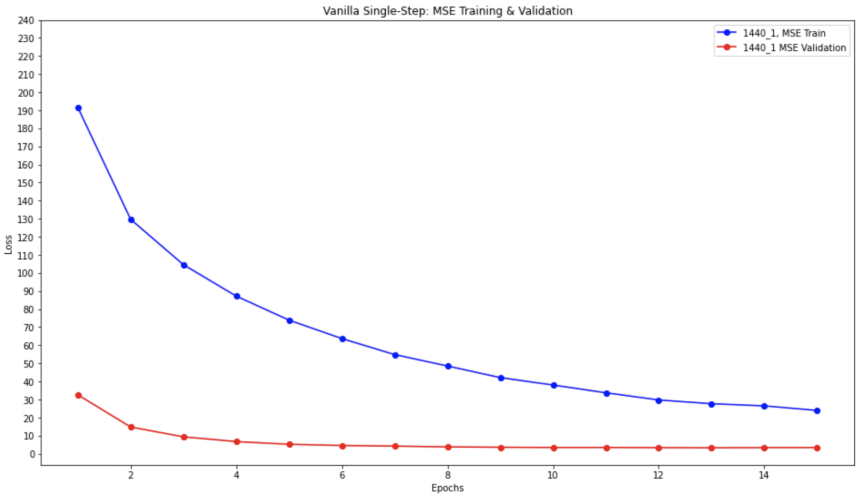

Figure 9: Vanilla 1440_1: Training History

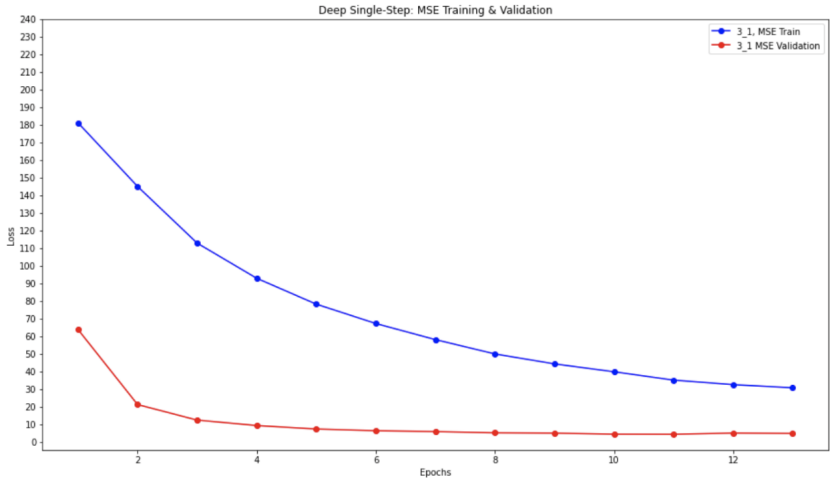

Figure 10: Deep 3_1: Training History

5.2 Multi-Step Forecasting Results

Intuitively, multi-step forecasting often turns out to be a much harder task compared to single-step forecasting, especially when dealing with aleatory assets such as cryptocurrency. Not only, the task would get more complicated the more one decides to go ahead on time. As the pre-fixed goal was to push the prediction as far as possible whilst trying not to compromise the outcome, the presented results were not as satisfactory as single-step forecasting.

Vanilla	1440_60	60_60
Train MSE	119.02	111.35
Train MAE	3.62	3.56
Train MAPE	17.03	17.27
Validation MSE	36.55	41.48
Validation MAE	2.84	3.08
Validation MAPE	12.47	13.24

(a) "Vanilla" Architecture Results

Deep	1440_60	60_60
Train MSE	104.39	175.14
Train MAE	3.76	5.59
Train MAPE	19.08	31.62
Validation MSE	38.11	71.24
Validation MAE	2.90606	4.58
Validation MAPE	12.43	19.01

(b) "Deep" Architecture Results

Figure 11: Multi-step Architectures' Fitting

As the length of the output increased, there was a corresponding increase in the likelihood of significant deviations from actual values. This, in turn, caused both the training MSE and the MAPE to rise. This indicates a mediocre forecasting ability. It is also worth mentioning that the model named Deep 60_60 performed the poorest out of all the models. It had a mean squared error (MSE) of 175 during training and a validation MSE of 71.24. Reasons for this outcome could be, once again, the excessive complexity of the model along with the short window size: it seems like one hour of observations are not enough to forecast the next hour. Differently, this time, the best model happens to be Deep 1440_60. Although the model does not shine when it comes to MSE, this model is the one with the least MSE along with satisfactory metrics. The validation MAPE turns out to be the lowest among all with an overall percentage of 12%, which is considered reasonably good. Let's see how Deep 1440_60 behaves in the test set:

```
708/708 [==============================] - 32s 43ms/step
708/708 [==============================] - 30s 42ms/step
708/708 [==============================] - 29s 41ms/step
708/708 [==============================] - 32s 45ms/step
MSE: 112.039 RMSE: 10.559 MAPE: 0.218 MAE: 4.21
```

Figure 12: Deep 1440_60, Predictions (on test set)

The `multi_metrics` outputs the four metrics of interest when it comes to evaluating the prediction output with the actual `y_test` values. The results are even worse when compared to the validation set ones. To try to see whether this model can perform better, there will be an attempt to see whether tuning the model by editing its structure would help to improve the performance in the training and validation set. The new architecture's structure will be the following:

- LSTM layer with 100 units, 'relu' activation function.
- LSTM layer with 100 units, 'relu' activation function.
- Dropout Layer with 0.1 rate
- Dense Layer with 60 units, 'tanh' activation function.
- Dense Layer with 1 units, 'tanh' activation function.

Some changes were made here.

1. The Layers were shortened and made more complex by adding more LSTM units and a different activation function.
2. Since MSE in Deep structure made no sign of lowering, the 2 dropout layers were cut off, thus remaining only one dropout layer.
3. The Dense layer has been removed to lighten the architecture.

Despite these changes, the metrics did not improve considerably.

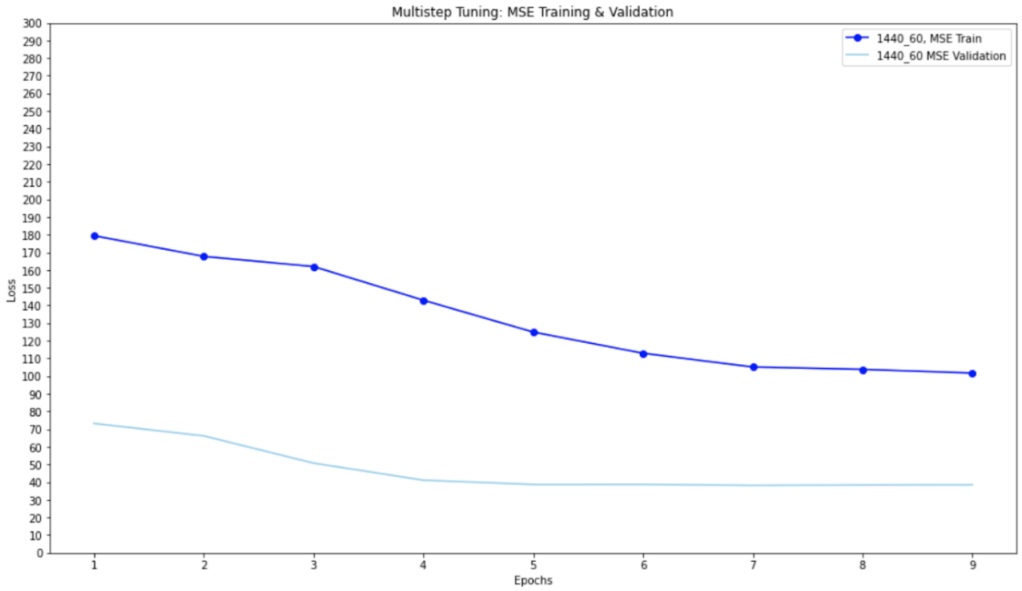

Figure 13: Multistep Tuning: MSE Training History

The plot above shows the history of the training. The MSE has slightly lowered although it still hasn't reached acceptable results, where the validation error remained pretty much around the same results.

6 Conclusions

The goal of the project was to try to predict the lowest gas price fee that validators would approve in order to ensure the processing of a transaction within the Ethereum Blockchain. This has been achieved through the implementation of two LSTM Neural Network architectures that took as input a time series of 600,000 mined blocks, each containing the minimum gas price accepted by the validator, to predict the minimum gas price fee for one-minute and one-hour intervals. Before the modeling phase, the data underwent a preliminary time series analysis that helped to gain insights about the time series behaviour over time, such as its seasonality and its autocorrelation up to a certain lag. This furtherly helped to make reasonably logic decisions when dealing with the right windowing span to feed the neural network with. While the single-step LSTM model proved to be an effective and reliable tool by reaching satisfactory metrics, the multi-step model, despite undergoing hyper-parameter tuning, wasn't able to produce relying results on the test set. However, further implementation of the project can be considered in order to improve the accuracy of the model.

1. **Include other variables in the model**. Although the feature taken into considerations was just one (`gas`), there were two other variables,

namely `price` and `n_txn`, that have not been considered for the sake of model's lightness. But what if the inclusion of these two other variables led to an overall better accuracy? Since the two variables have different unit of measurements and scale a normalisation would be needed.

2. **Change the resampling interval**. The current method uses a 1-minute up-sampling method, but what if using a higher frequency such as 5 or 10 minutes led to different results? However, increasing the frequency would require extracting more data, as neural networks (particularly LSTM layers) require a large number of observations to function effectively.

3. **Change the window-length** Although the input data had a range of observation spans from broad to short, this may not have been sufficient. It's possible that using a middle or longer span of observations could improve the forecast accuracy.

4. **Change the architecture**. On this project, only two types of architectures were covered, which result in 2 combinations of LSTM architectures being involved with relative hyper-parameters. However, there exist multiple combinations that take into consideration both the parameters within each layer and the layer themself. Not only, there exist different combinations also involving different neural networks layers such as CNN, or GRU layers. Perhaps trying adding (or substituting) those layers can lead to a different result.

5. **Just accept the result as for what it is**. In the financial and crypto environment, forecasting can be challenging due to the randomness of certain variables. Sometimes, despite attempting various methods, it may be necessary to accept that a specific outcome is the best achievable result.

References

[1] Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[2] Jason Brownlee. *Deep Learning for Time Series Forecasting*. 2018.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] Rawya Mars, Amal Abid, Saoussen Cheikhrouhou, and Slim Kallel. A machine learning approach for gas price prediction in ethereum blockchain. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 156–165. IEEE, 2021.