AiAp – MiniProject 2

Simon Hager, Kevin Löffler

Intro

Today, many interesting problems can be solved with machine learning. One that has always fascinated us is stock market prediction. If you ask investors they will always tell you that it is impossible for any human – no matter how skilled or experienced – to reliably predict what the market is going to do next. But a computer is a different story.

Computers are already active in the stock market. A survey by the Aite Group estimates that computers engaging in algorithmic trading are responsible for more than 70% of orders, while only representing about 2% of all investment firms. The prevailing strategy that is employed is high-frequency trading. That means computers open and close positions in milliseconds trying to profit from minor course changes.

In this paper, we want to explore different strategies to tackle this problem that has long been thought to be unsolvable.

Idea

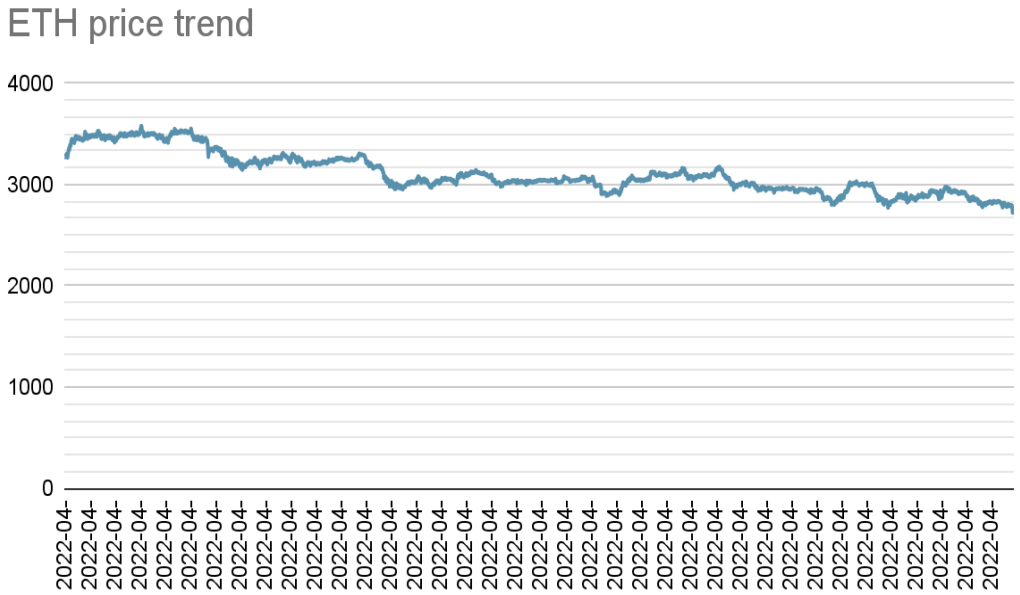
We do not have the resources to employ a high frequency strategie like the institutional traders. For one, that would require a supercomputer in very close proximity to a stock exchange and a very low latency signal. The best feed we could find was the one from alpaca trading. They provide a very powerful api with access to historical as well as live data and a paper account with fake money to test our model and algorithm. We evaluated different types of signals:

1. Individual trades
We could subscribe to every trade for a specific symbol
+ A lot of data points for our model to make predictions
+ Short testing / iteration cycles
- To much data to be handled by our computers

2. Minute Bars
A summary of all trades done in the last minute
+ Compromise between too much and too little data
+ A minute is enough time for our model to make a prediction
- Not as many opportunities to make trades

3. Hour Bars
A summary of all the trades done in the last hour
+ Bigger trends for the model to recognize
- Very long wait times
- Unpractical to develop and test

We picked minute bars as a compromise because they provide short iteration cycles while still allowing our computer enough time to make a prediction and act on it. From those minute bars we use the volume weighted average price as our datapoint. We ran a preliminary test with a data vector with additional data points like the open or close price and so on, but they did not result in a more accurate prediction but increased the training time by a lot.



Data

Data Acquisition

We gathered data from two REST endpoints: one for stock data and one for crypto. We gathered around 100’000 continuous minute bars per symbol, extracted the relevant features (vw price and timestamp) and exported them into a csv file.

average-price	timestamp
3279.5485485579	2022-04-01T09:00:00Z
3281.1	2022-04-01T09:00:00Z
3276.699709582	2022-04-01T09:01:00Z
3277.1603050346	2022-04-01T09:02:00Z
3277.691774973	2022-04-01T09:03:00Z
3277.4296405528	2022-04-01T09:04:00Z
3278.5151515152	2022-04-01T09:04:00Z
3277.8032514736	2022-04-01T09:05:00Z
3275.9810009588	2022-04-01T09:06:00Z
3278.1	2022-04-01T09:06:00Z
3275.4623372092	2022-04-01T09:07:00Z
3276.8157442986	2022-04-01T09:08:00Z

Data Preprocessing

A different script read the csv files and loaded the data into panda dataframes. After that we normalized the data to a range between zero and one and split it into a testing set (80%) and a training set (20%). Because we work with time series data we want to use long short term memory cells. These LSTMs need data in a specific format. One of the hyperparameters we used was the step size. It defines how many data points our model uses to make a prediction.

Model

We found the following model architecture to work best for us:

Four pairs of keras LSTM layers with 50 units and a 0.2 dropout layer, followed by a Dense layer with a single output: the prediction. The first three LSTM layers preserve the input shape.

The trained model is saved with all its weights, compilation information, optimizer and its state with the tf.keras.models.save_model() api.

Hyperparameters

To find the best parameters for the model we tested 72 different combinations. On the left side are the tested hyperparameters and on the right side are the training results. The most suitable combinations for our purpose are:

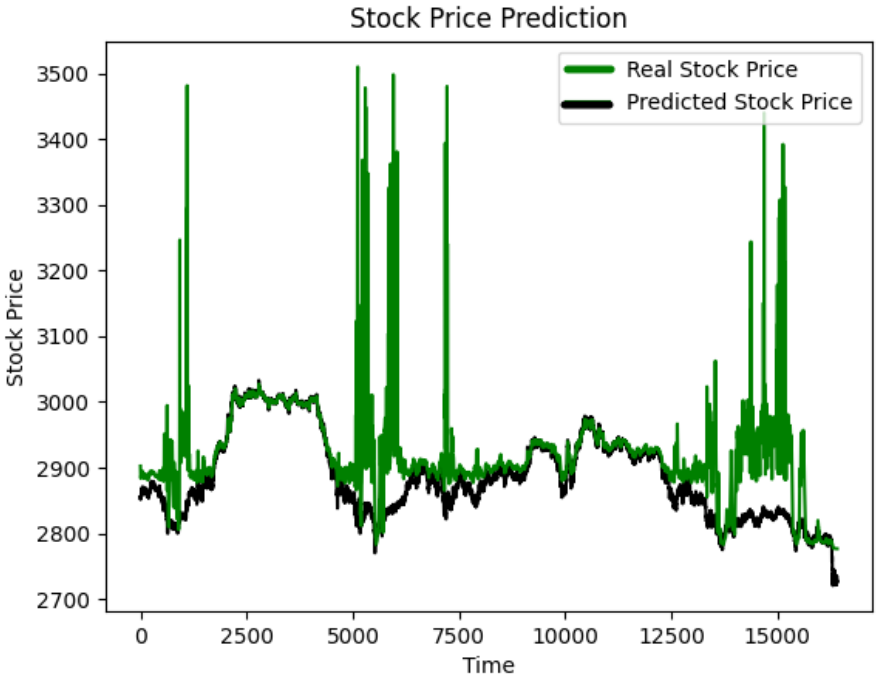
	units	step	batch_size	learning rate	dropout	test loss	neg error[1]	pos error[2]
1	50	30	64	0.001	0.2	0.00031	-0.0140	0.0124
2	50	30	64	0.0005	0.2	0.00035	-0.0134	0.0121
3	50	30	64	0.001	0.1	0.00034	-0.0141	0.0125

[1] Sum of all negative differences between price predictions and their real prices
[2] Sum of all positive differences between price predictions and their real prices

The decision is based on three criteria

- Lowest test loss
- Lowest error values
- Most symmetric error values

Loss is a well-known general metric. But in order to make a really concrete statement about the effectiveness of the model in trading, the metrics must be adapted to the intended use or purpose. That is why the error metric exists. This gives us a quantitative estimate of what financial impact buy or sells will have based on the predictions of the trained model.



Trading

To get the live signals we implemented a websocket client that connects to the alpaca live trading api. It receives a signal every minute, extracts the latest course and then hands the data over to the prediction algorithm.

The prediction algorithm takes the price data from the recent past [$p_{t - step\_size}$, p_t] and predicts the next price p_{t+1}. This prediction then needs to be translated to an action that can be taken. There are three general actions:

1. Buy
2. Sell
3. Hold, aka no action

Because our prediction returns an exact price we want to use that information and tune our agent accordingly. When the predicted price is a lot higher than the current price the agent should buy more compared to when the predicted price is just marginally higher. Additionally we want the agent to sell more aggressively than to buy, so we implemented two slightly different action functions:

$$f(x) = \frac{1}{1 + e^{-a(x-1)}} + 0.5 \quad \{x \mid x \geq 1\}$$

$$f(x) = \frac{0.5}{1 + e^{-a(x-1)}} + 0.75 \quad \{x \mid x \leq 1\}$$

The parameter “a” defines how aggressive the agent should trade. We found that a value of 50 leads to a very passive or conservative agent while a value of 200 leads to almost erratic behavior. Tests with our dataset have found a value between 100 and 125 to be most optimal.

The formula returns a value in the range [0.5, 1.5]. This is the factor by which the current holding should be changed:

0.5 = sell half of the current holding
1.0 = do nothing
1.1 = buy an additional 10%.

The agent gets the current holding from the api and then places a buy or sell order respectively.

Result

We let the agent trade on its own for several hours and it resulted in an average profit of 0.18% per hour. This would mean - given constant performance - the agent would outperform the market by 1’572% over an entire year. If this performance is actually realizable is currently unclear because our tests only ran for a few hours.

This mini project was an interesting and intensive challenge. We needed to learn about many different concepts: how to use LSTM’s, how to format data as a time series, how to use the alpaca api and some algorithmic challenges when coming up with a function to translate predictions into actions.