Deep Learning for Time Series

Alen Pavlovic
CQF Institute, Fitch Learning

Table of Contents

Abstract

Short-term asset return is a challenging quantity to predict. Efficient markets produce near-normal daily returns with no significant correlation between r_t, r_{t-1}. This exam is a limited exercise in supervised learning for predicting Bitcoin up/down moves.

Keywords: deep-learning, machine learning, supervised learning, TensorFlow

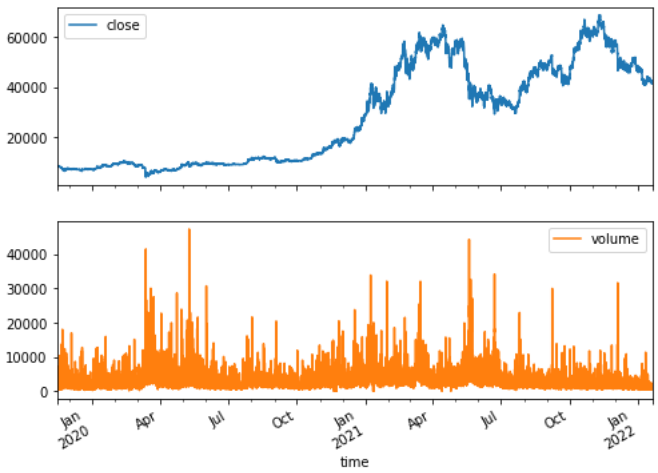
Deep Learning for Time Series

Summary

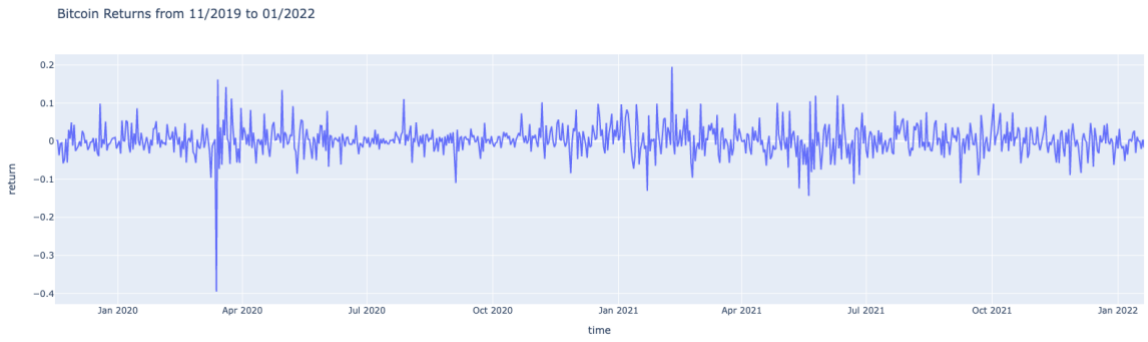
If one believes the data carries an autoregressive structure: a recurrent neural network model can be a successful alternative to time series regression.

The Bitcoin Dataset

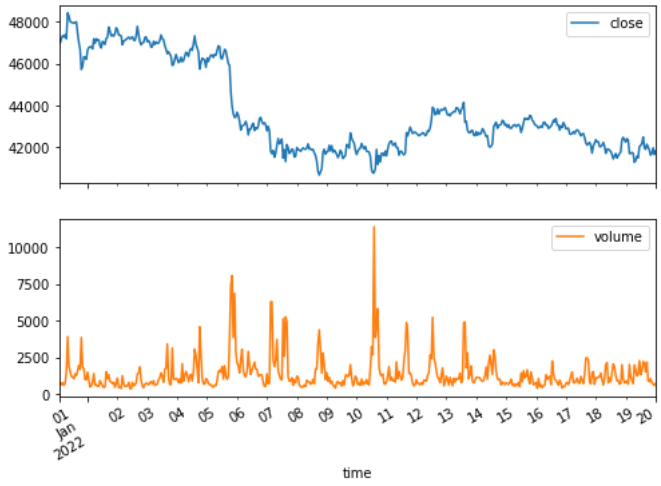
This report uses a Bitcoin spot (BTCUSDT) time-series dataset recorded from the Binance API, available data is from November 2019 until January 2022 with roughly 20,000 entries of the data. This dataset contains OHLCV data bucketed every hour to speed up optimization algorithms. In this project, we will deal with **hourly predictions** of up/down moves. Here is the evolution of a close price and volumes(in bitcoin) over time.



From the photo above you can see the incredible growth of Bitcoin price over the last two years, from March 2020 when the price dropped to ~4,000 USD range to a maximum of 69,000 USD.



The worse return happened on March 12, 2020, while the best return was on February 2, 2021. The “bad day” is really easily observable from the figure above.



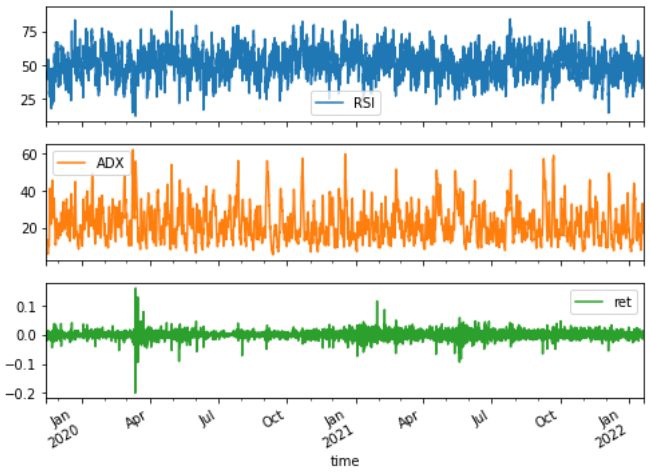
The recent Bitcoin journey is not that optimistic, there is a lot of uncertainty in the market recently on the macroeconomic level and we can see that Bitcoin follows the conventional financial markets. At the time of writing this paper, the price is around 35,000USD.

Feature Engineering

Before diving in to build a model, it’s important to understand your data and be sure that you’re passing the model appropriately formatted data.

	time	open	high	low	close	volume
0	2019-11-15 00:00:00	8646.38	8659.41	8610.87	8611.98	1248.453018
1	2019-11-15 01:00:00	8611.99	8627.00	8526.11	8559.96	2567.333631
2	2019-11-15 02:00:00	8557.95	8590.06	8526.11	8538.61	2428.512671
3	2019-11-15 03:00:00	8538.16	8579.99	8517.00	8558.15	1777.637624
4	2019-11-15 04:00:00	8558.13	8571.08	8531.61	8557.45	1010.078664

From the screenshot above you can see the start date of the data and the buckets of one hour. For this exercise, we will use Ta-lib to add technical indicators to the data. A full list can be in the Python Script feature selection added with the report.



The figure represents a couple of indicators generated- RSI, ADX, and returns. Behaviour is random, which makes this project just more interesting.

Variance Threshold

Feature selector that removes all low-variance features. This feature selection algorithm looks only at the features (X), not the desired outputs (y), and can thus be used for unsupervised learning.

```
variance_threshold = sklearn.feature_selection.VarianceThreshold(threshold = 0.01)
variance_threshold.fit(train_df_x) # # fit finds the features with zero variance

# get_support is a boolean vector that indicates which features are retained
# if we sum over get_support, we get the number of features that are not constant
sum(variance_threshold.get_support())
```

```
48
```

```
# print the constant features
print(
    len([
        x for x in train_df_x.columns
        if x not in train_df_x.columns[variance_threshold.get_support()]
    ]))

[x for x in train_df_x.columns if x not in train_df_x.columns[variance_threshold.get_support()]]
```

```
24
['ret',
 'ret_1',
 'ret_2',
 'ret_3',
 'ret_4',
 'ret_5',
```

It doesn't come as a surprise to see those return columns are low variance columns and we will drop them.

Select K Best

The SelectKBest method selects the features according to the k highest score. By changing the 'score_func' parameter we can apply the method for both classification and regression data. Selecting the best features is an important process when we prepare a large dataset for training. It helps us to eliminate a less important part of the data and reduce training time.

As we're trying to predict the price we have a regression problem and the appropriate score function to use is *f_classif* with 10 features selected.

	0
0	BOP
1	CCI
2	FASTK
3	FASTD
4	STOCH
5	ULTOSC
6	WILLR
7	ADOSC
8	ret
9	ret_1

Let's compare the results with BorutaPy.

BorutaPy

Boruta is an all-relevant feature selection method, while most others are minimal optimal; this means it tries to find all features carrying information usable for prediction, rather than finding a possibly compact subset of features on which some classifier has a minimal error.

	Feature	Ranking
4	volume	1
44	BOP	1
45	CCI	1
52	FASTK	1
53	FASTD	1
54	STOCH	1
55	ULTOSC	1
56	WILLR	1
58	ADOSC	1
61	ret	1
62	ret_1	1
77	ret_16	1
84	ret_23	1

Final Features

Taking into account all of the above about the variance threshold and the features given by the SelectKBest and BorutaPy, the following set of features will be used in the final predictions.

- Close
- Volume
- BOP
- CCI
- FASTK
- FASTD
- STOCH
- ULTOSC
- WILLR
- ADDOSC
- Return

Split the data

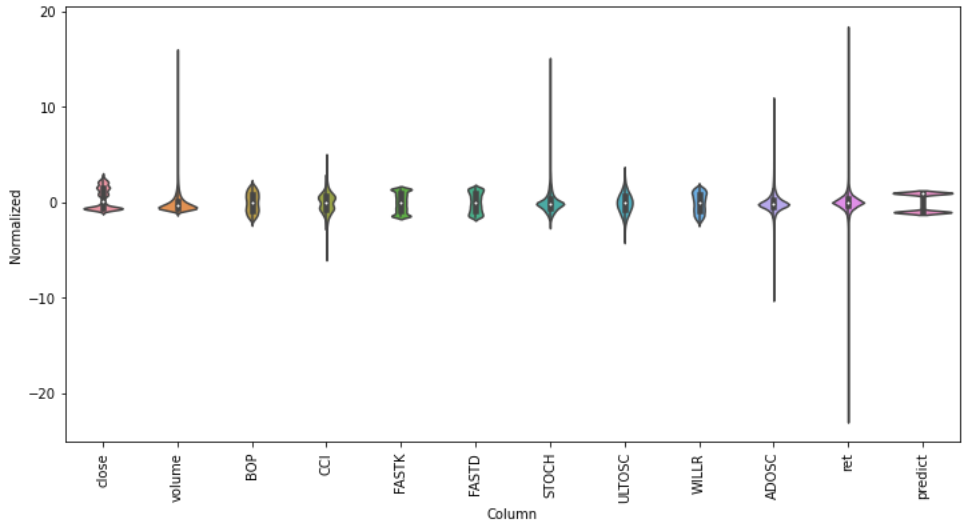
We will use a (70%, 20%, 10%) split for the training, validation, and test sets. Note the data is not being randomly shuffled before splitting. This is for two reasons:

1. It ensures that chopping the data into windows of consecutive samples is still possible.
2. It ensures that the validation/test results are more realistic, being evaluated on the data collected after the model was trained.

Normalize the data

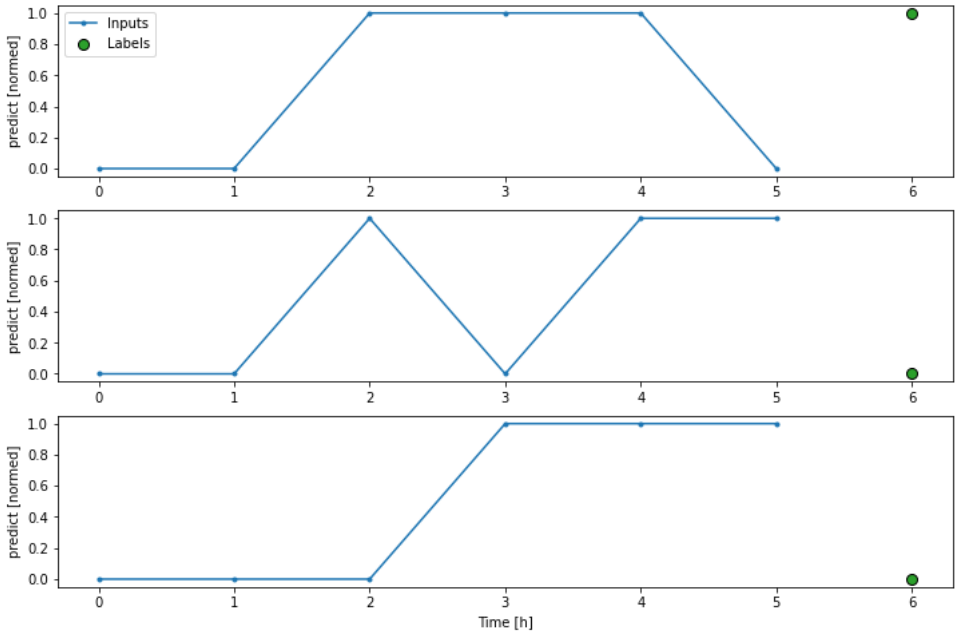
It is important to scale features before training a neural network. Normalization is a common way of doing this scaling: subtract the mean and divide by the standard deviation of each feature. The mean and standard deviation should only be computed using the training data so that the models have no access to the values in the validation and test sets.

It's also arguable that the model shouldn't have access to future values in the training set when training, and that this normalization should be done using moving averages.



Now, peek at the distribution of the features. Some features do have long tails, but there is no obvious error.

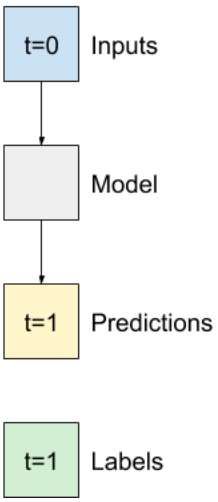
Time Windowing Plot



An important feature in time-series prediction is data windowing. Per the figure above you can see that if we're trying to predict value at t = 6 we will be using data until t = 5. We will create batches of 24 entries, so each sample will hold 24 hours of data (1 day).

Models

The simplest model you can build on this sort of data is one that predicts a single feature's value—1 time step (one hour) into the future based only on the current conditions. So, start by building models to predicted the predict(up/down) moves value one hour into the future.

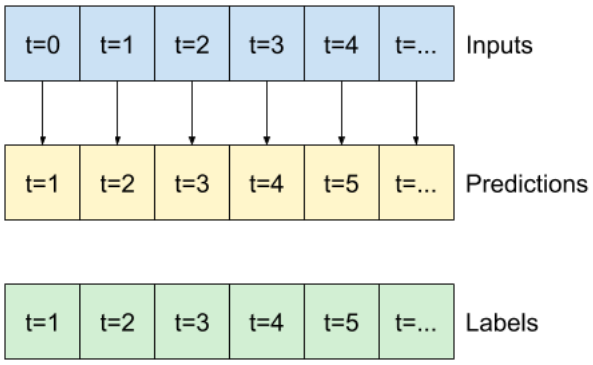


All models will be trained **50 epochs**.

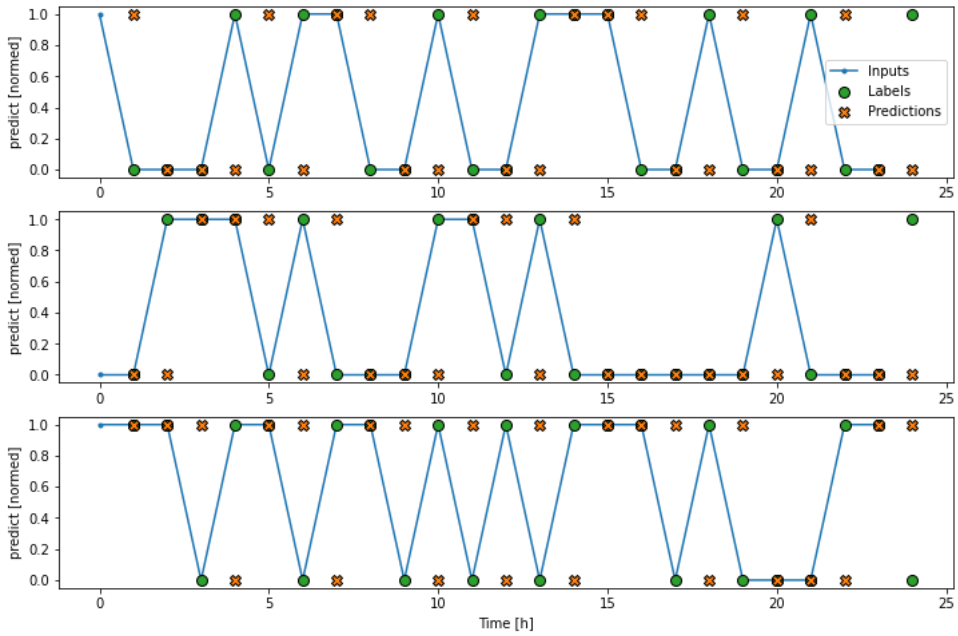
Baseline

Before building a trainable model it would be good to have a performance baseline as a point for comparison with the later more complicated models. This first task is to predict up/down moves one hour into the future, given the current value of all features. The current values include the current price.

So, start with a model that just returns the previous prediction as to the prediction, predicting "No change". This is a reasonable baseline since we're not expecting the price to change significantly over a period of one hour all the time. Of course, this baseline will work less well if you make a prediction further in the future. The summary of the model is visible below:



By plotting the baseline model's predictions, notice that it is simply the labels shifted right by one hour:

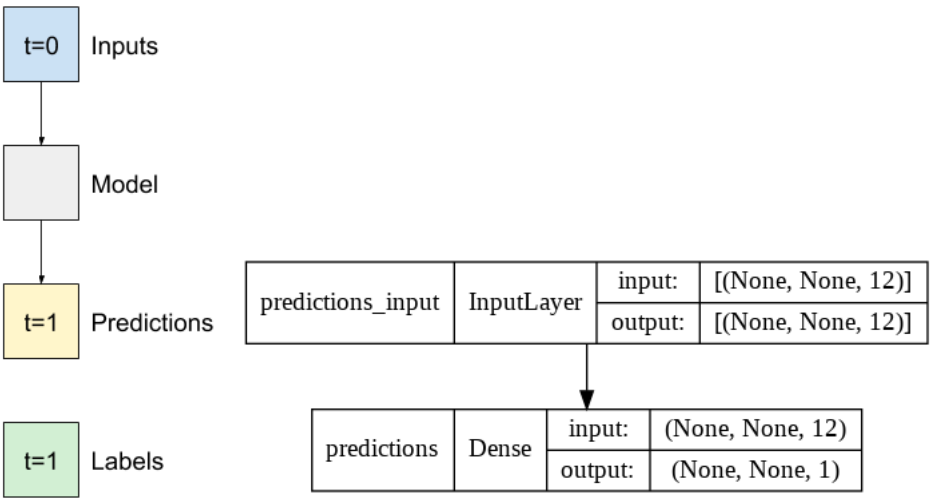


In the above plots of three examples, the single-step model is run over the course of 24 hours. This deserves some explanation:

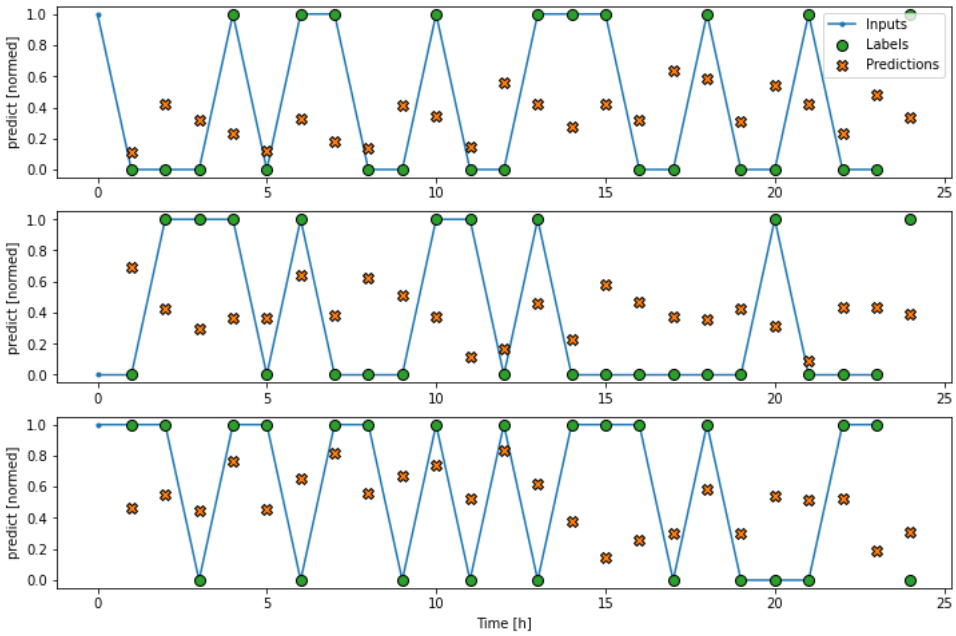
- The blue *Inputs* line shows the correct prediction at each time step. The model receives all features, this plot only shows the prediction(up/down move)
- The green *Labels* dots show the target prediction value. These dots are shown at the prediction time, not the input time. That is why the range of labels is shifted 1 step relative to the inputs.
- The orange *Predictions* crosses are the model's predictions for each output time step. If the model were predicting perfectly the predictions would land directly on the *Labels*.

Linear Model

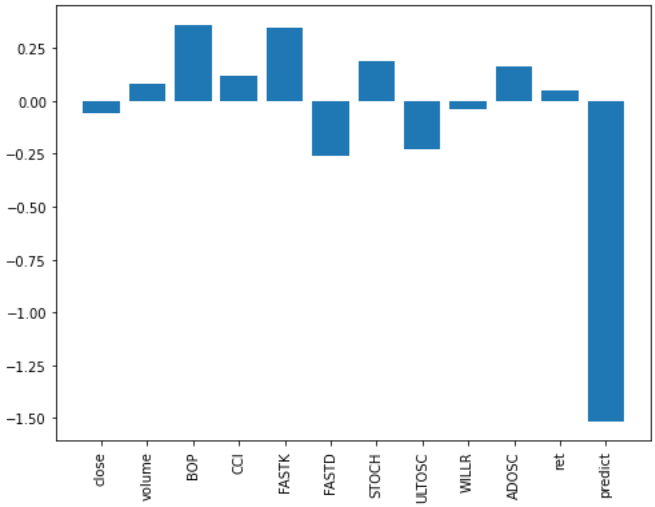
The simplest trainable model you can apply to this task is to insert linear transformation between the input and output. In this case, the output from a time step only depends on that step:



Here is the plot of its example predictions on the predictions, note how in many cases the prediction is better than just returning the input temperature, but in a few cases it's worse:



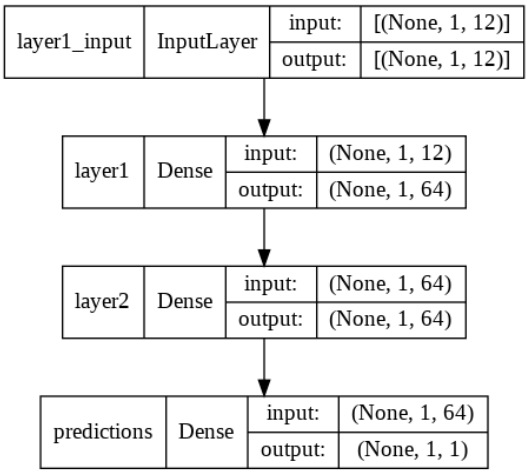
Output layer in this case is using the sigmoid function, so we can see the probability exactly of up/down moves and not just the binary value. While the majority of the value is around the 0.5 thresholds there are also extremes. One advantage to linear models is that they're relatively simple to interpret. You can pull out the layer's weights and visualize the weight assigned to each input:



Sometimes the model doesn't even place the most weight on the input *predict*. This is one of the risks of random initialization.

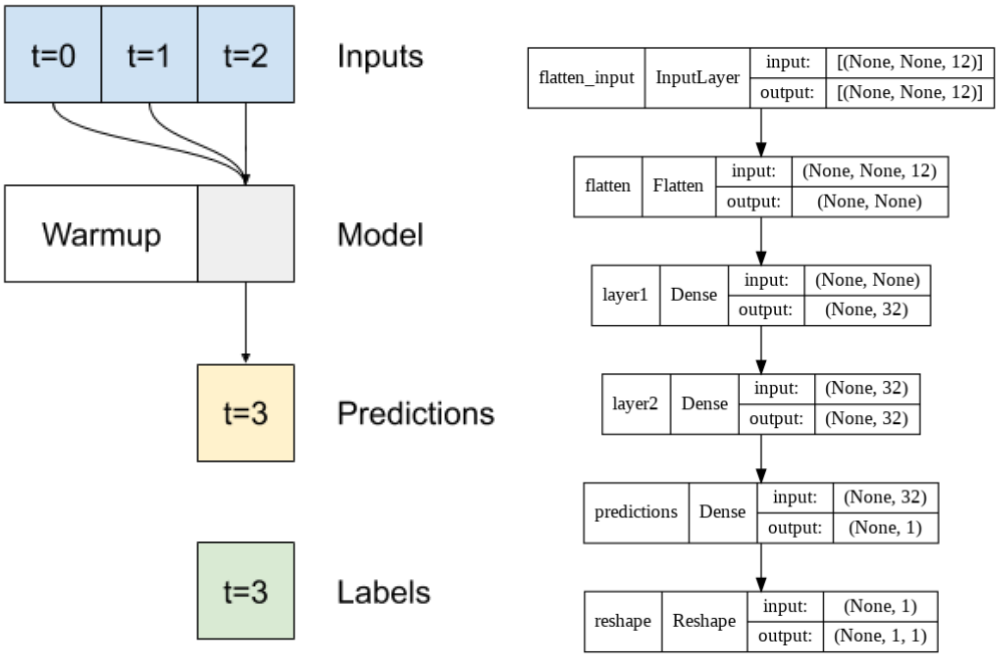
Dense

Here's a model similar to the linear model, except it stacks several a few Dense layers between the input and the output.

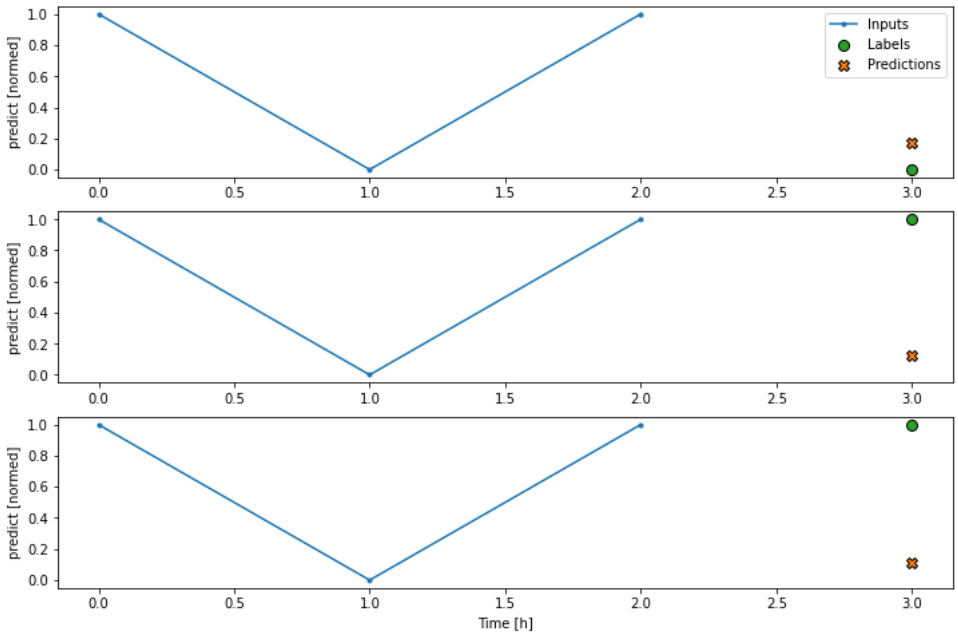


Multi-step Dense

A single-time-step model has no context for the current values of its inputs. It can’t see how the input features are changing over time. To address this issue the model needs access to multiple time steps when making predictions:



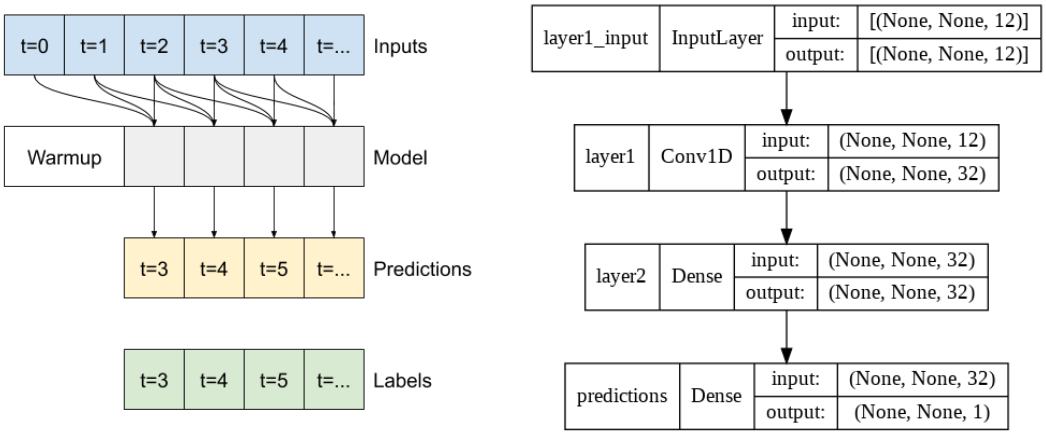
The *baseline*, *linear* and *dense* models handled each time step independently. Here the model will take multiple time steps as input to produce a single output.



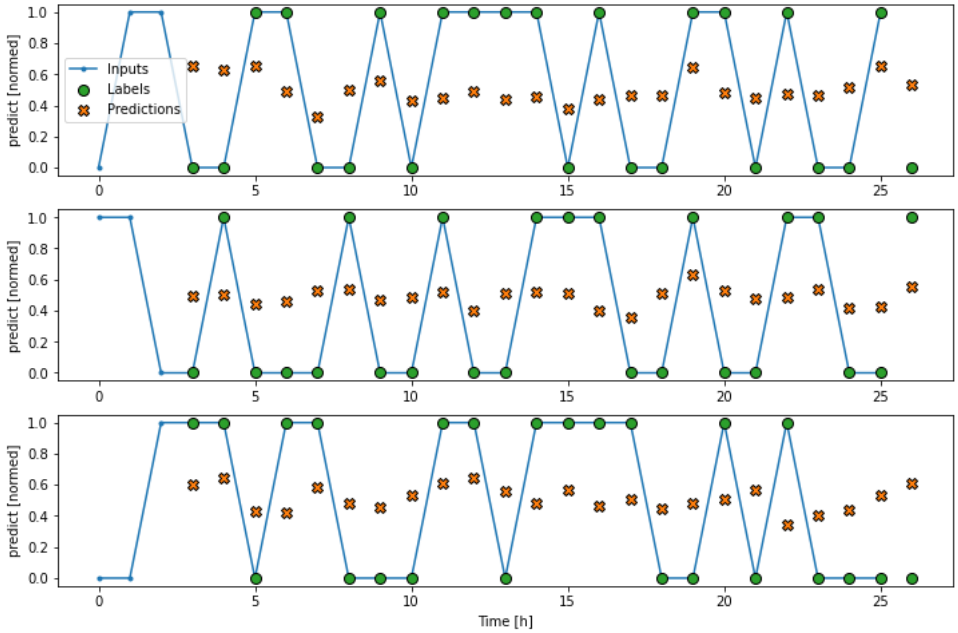
The main downside of this approach is that the resulting model can only be executed on input windows of exactly this shape. The convolutional models will fix this problem.

Convolution Neural Network

The difference between this *conv_model* and the *multi_step_dense* model is that the *conv_model* can be run on inputs of any length. The convolutional layer is applied to a sliding window of inputs:



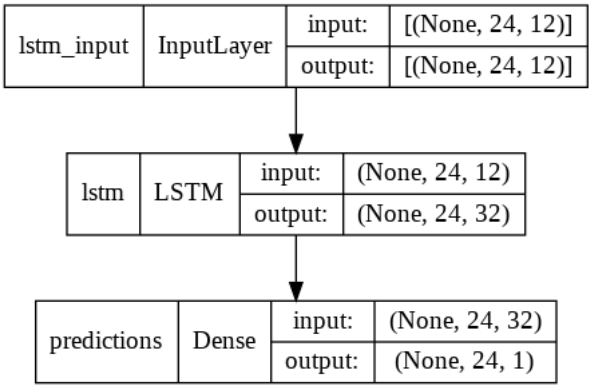
Note the 3 input time steps before the first prediction. Every prediction here is based on the 3 preceding time steps:

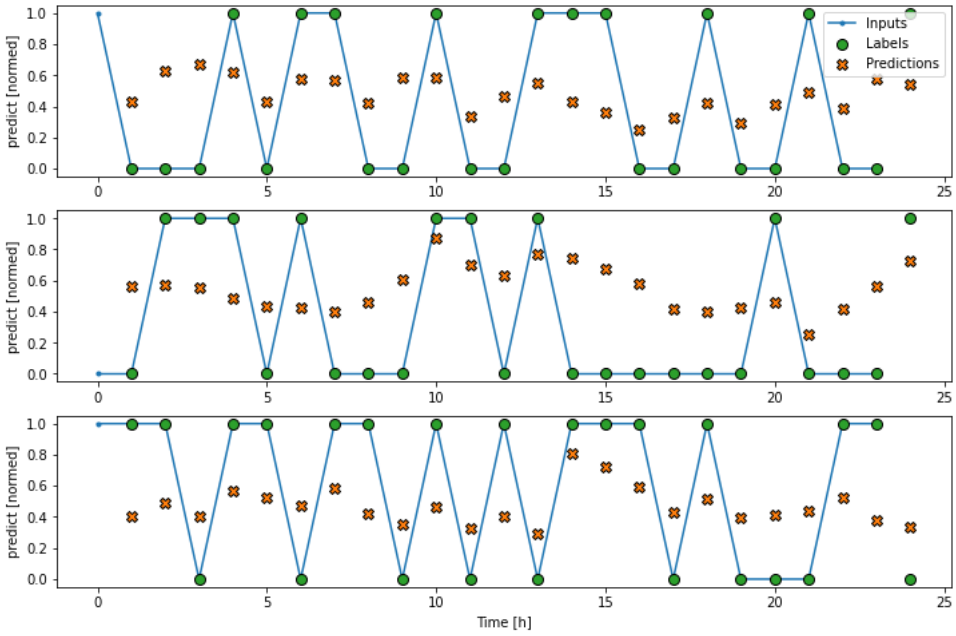


We can see how conservative the CNN model is, a lot of probabilities around the threshold of 50%, but there are a few with a 70% move up or 30% move down and the predictions are correct. The model has a lot more parameters to learn and the ability to fit the data.

Recurrent neural network

A Recurrent Neural Network (RNN) is a type of neural network well-suited to time series data. RNNs process a time series step-by-step, maintaining an internal state from time-step to time-step.

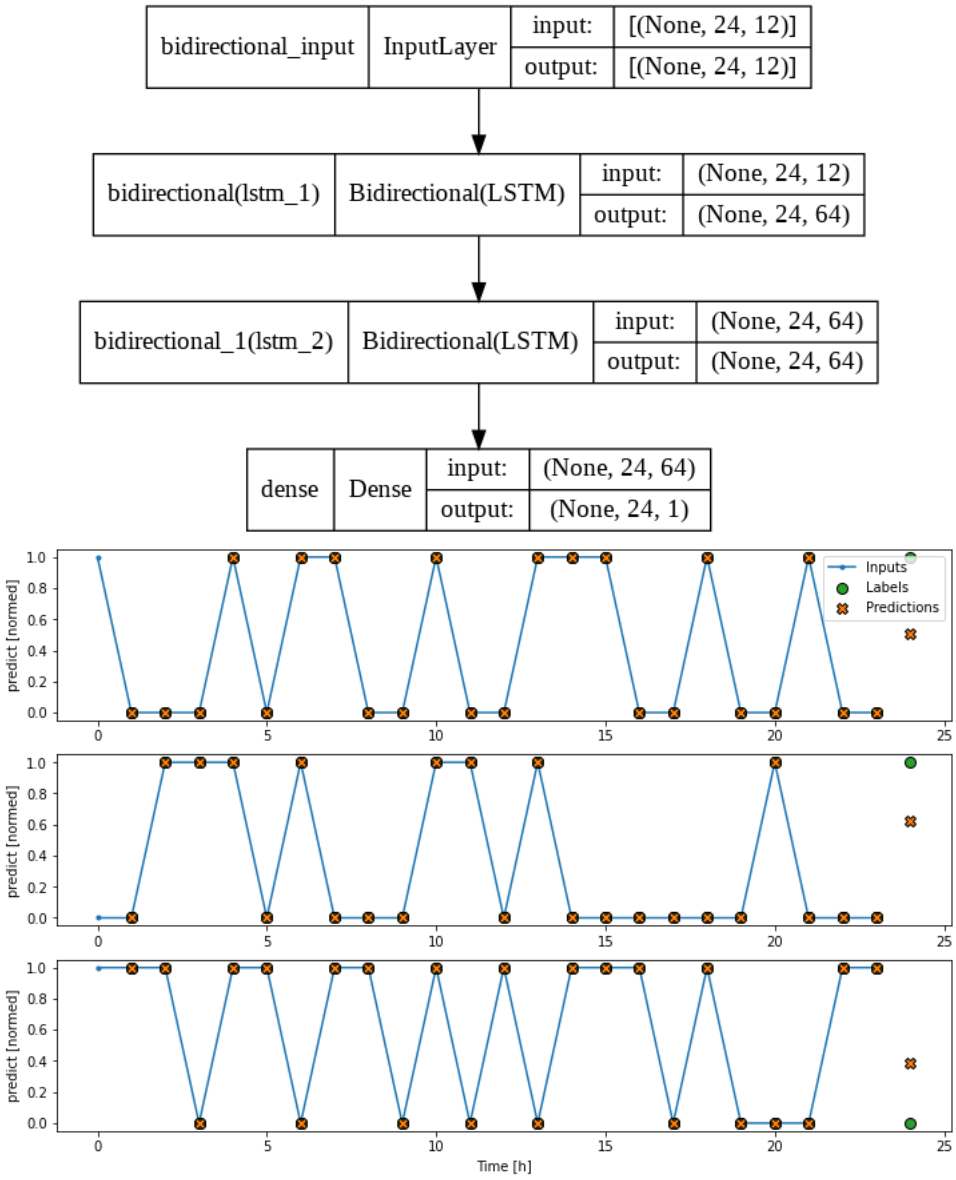




The advantages of the model are obvious, the model is using LSTM and offers a lot of good predictions.

Bidirectional

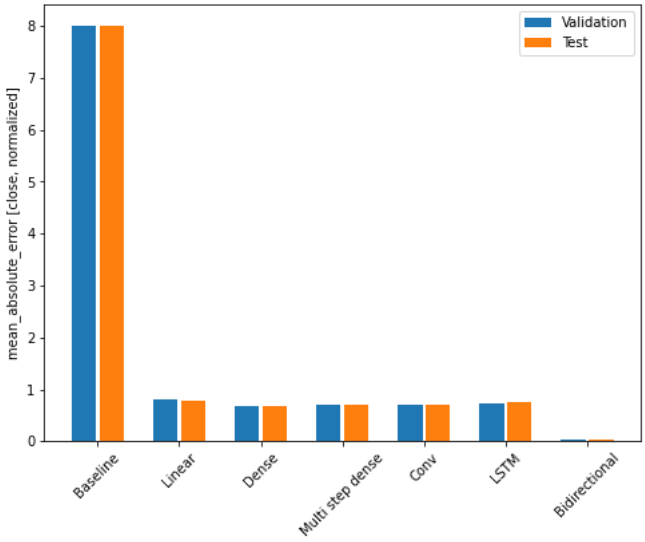
The most complicated model so far, is bidirectional LSTM of two layers.



The results are impressive. The model run in the example above is the following:

```
bidirectional_model = tf.keras.models.Sequential([
  tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32,
return_sequences=True)),
  tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32,
return_sequences=True)),
  tf.keras.layers.Dense(1, activation = 'sigmoid'),
  ])
```

Performance



Not surprising that the Bidirectional model is the best. Performance table below:

Model	Loss
Baseline	7.9940
Linear	0.7862
Dense	0.6896
Multi step dense	0.6896
Conv	0.7009
LSTM	0.7628
Bidirectional	0.0294

The loss on bidirectional model is pretty low, model is most likely overfitted.

Hyperparameters

Model Dense and Dropout

Trial ID	Show Metrics	num_units	dropout	optimizer	Binary Crossentropy
3af151b620727...	☐	64.000	0.10000	adam	0.49387
3df0d7cf35bec...	☐	32.000	0.20000	sgd	0.49450
3ec2aed9e075...	☐	32.000	0.20000	adam	0.49443
4b9e1b33ce75...	☐	128.00	0.20000	sgd	0.49439
5751d85a2bfe4...	☐	64.000	0.20000	adam	0.49438
6826c7fa3322...	☐	32.000	0.10000	adam	0.49478
7b29a731e3da...	☐	32.000	0.10000	sgd	0.49483
8152feef9da0d...	☐	256.00	0.10000	adam	0.0000
925f5ca53d9d...	☐	256.00	0.10000	sgd	
a8673a99dbfe...	☐	128.00	0.20000	adam	0.49448
af867036546f...	☐	64.000	0.20000	sgd	0.49436
b8ffc1aba3abc...	☐	64.000	0.10000	sgd	0.49432
d28ff7e45100a...	☐	128.00	0.10000	sgd	0.49385
ec32ab7f5674...	☐	128.00	0.10000	adam	0.49453

Using hyperparameters optimization on a model with dense layers and dropout we can see that more units does not make model necessarily better.

LSTM

Trial ID	Show Metrics	num_units	filter_units	optimizer	Mae
ef35b1fcc5674...	☐	256.00	16.000	adam	0.020544
58c2bff41ffaa9...	☐	256.00	16.000	sgd	0.020601
ad2cee388c09...	☐	32.000	16.000	adam	0.020659
540177c70130...	☐	128.00	16.000	sgd	0.020707

LSTM with more units is definitely better performing but the learning process is taking a lot longer.

Next Steps

Throughout this project, it was demonstrated how to prepare the data for the machine learning process. Time-series analysis can be very lucrative in the trading business and the crypto world is super attractive.

Models that perform the best are the most complicated ones, using LSTM and CNN. To optimize neural networks it would require a lot more data and training different neural networks to see which one will have robust performance and to avoid any overfitting.

References

[1] Tensorflow documentation (Link)

[2] Kannan Singaravelu (2021), Introduction to Deep Learning & Neural Networks

[3] Yves Hilpisch (2020), Artificial Intelligence in Finance