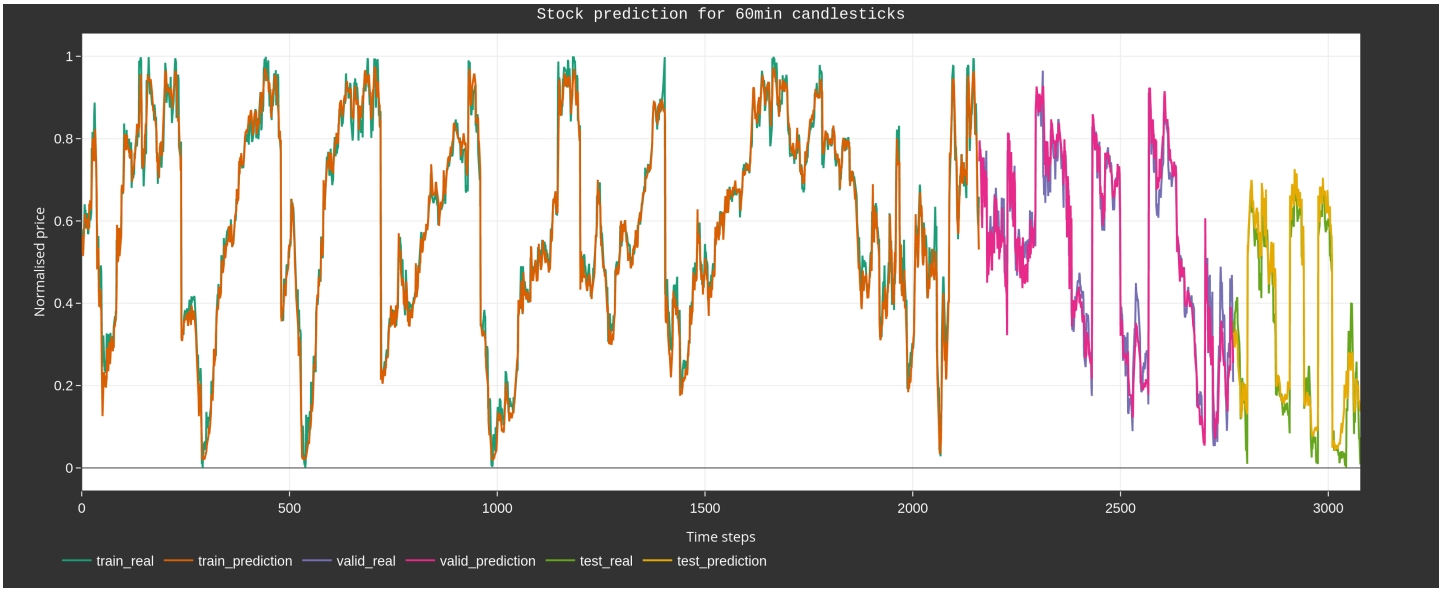


Normalized stock price predictions for train, validation and test datasets. Don't be fooled!

TRADING WITH AI

Stock prediction using recurrent neural networks

Predicting gradients for given shares

Joshua Wyatt Smith Aug 21, 2019 · 12 min read

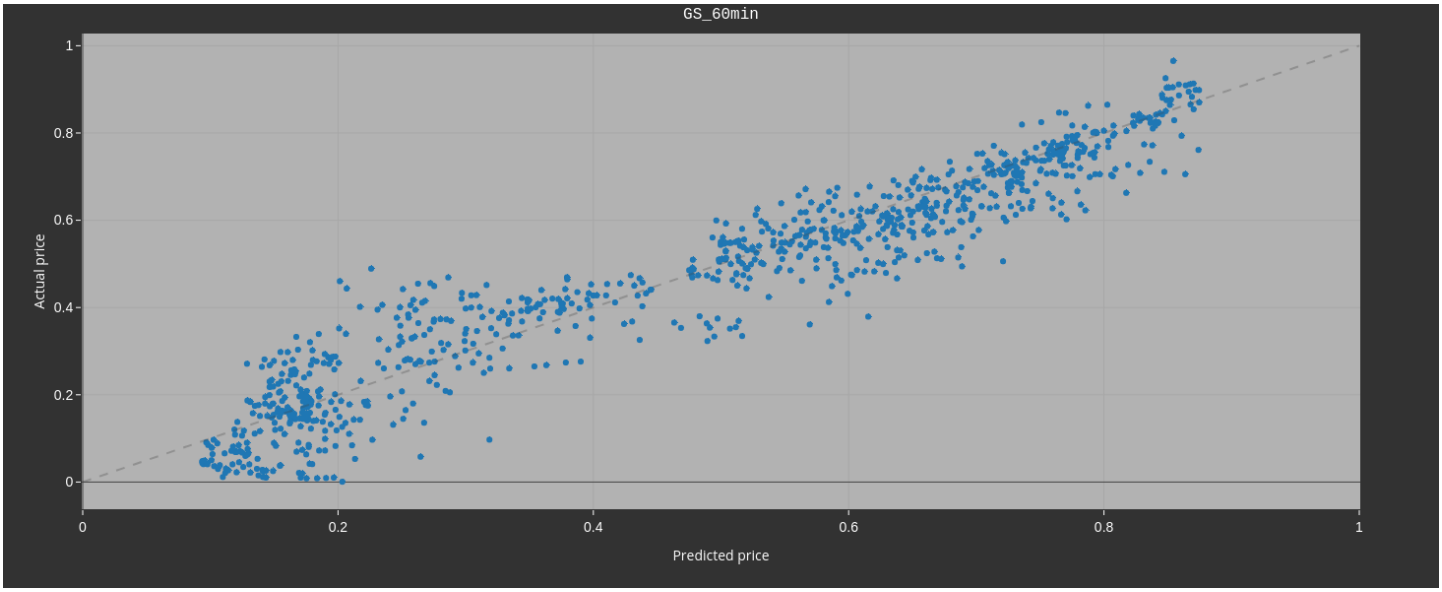
This type of post has been written quite a few times, yet many leave me unsatisfied. Recently, I read *Using the latest advancements in deep learning to predict stock price movements*, which, I think was overall a very interesting article. It covers many topics and even gave me some ideas (it also nudged me into writing my first article ☺). But it doesn't actually say how well the network performed. My gut feeling says "not well" given that this is usually the case, but maybe/hopefully I'm wrong!

This covers:

1. **The challenge**
2. **Data**
3. **Building datasets**
4. **Training**
5. **Results**
6. **Final Remarks**

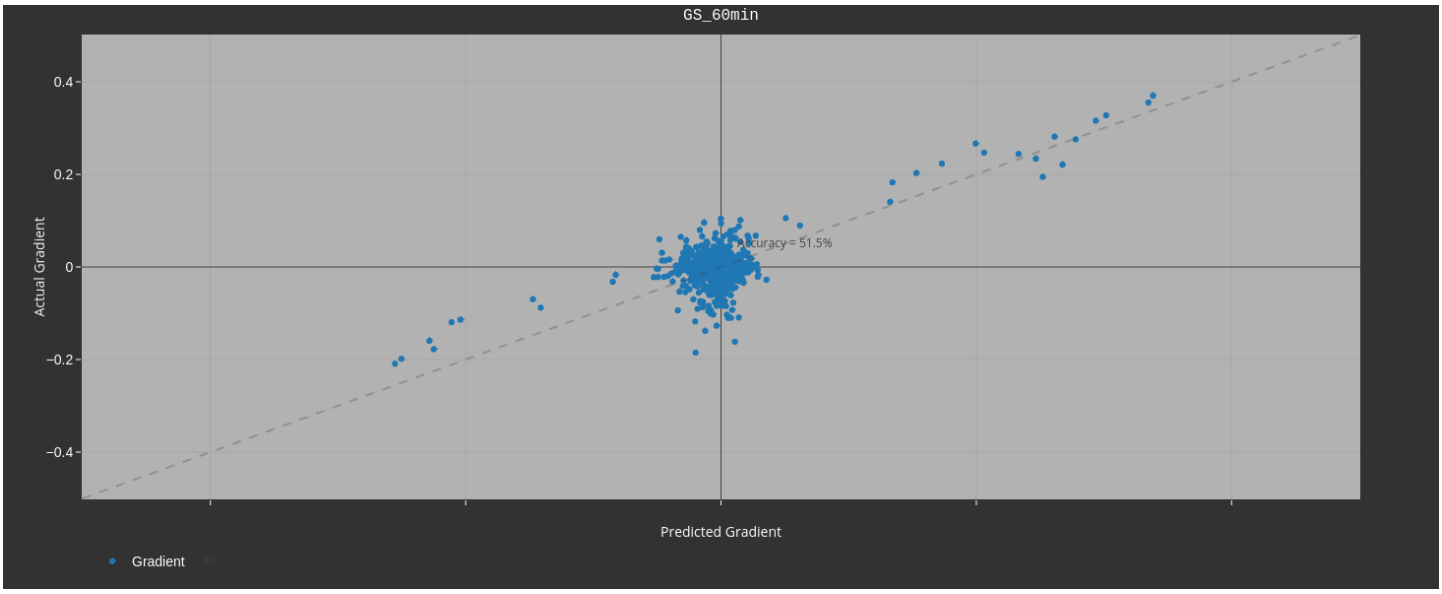
The challenge

The overall challenge is to determine the gradient difference between one Close price and the next. *Not* the actual stock price. Why? It's easy to fool yourself into thinking you have a viable model when you are trying to predict something that could fluctuate marginally ($|<0.01\%|$) and/or largely ($|>5\%|$). The plot below gives an example of this. A basic model (nothing special) was trained to predict the (normalized) price of Goldman Sachs:



Actual vs predicted (normalized) prices for the validation dataset.

developed from this. But what happens if we plot the gradient between two consecutive points?



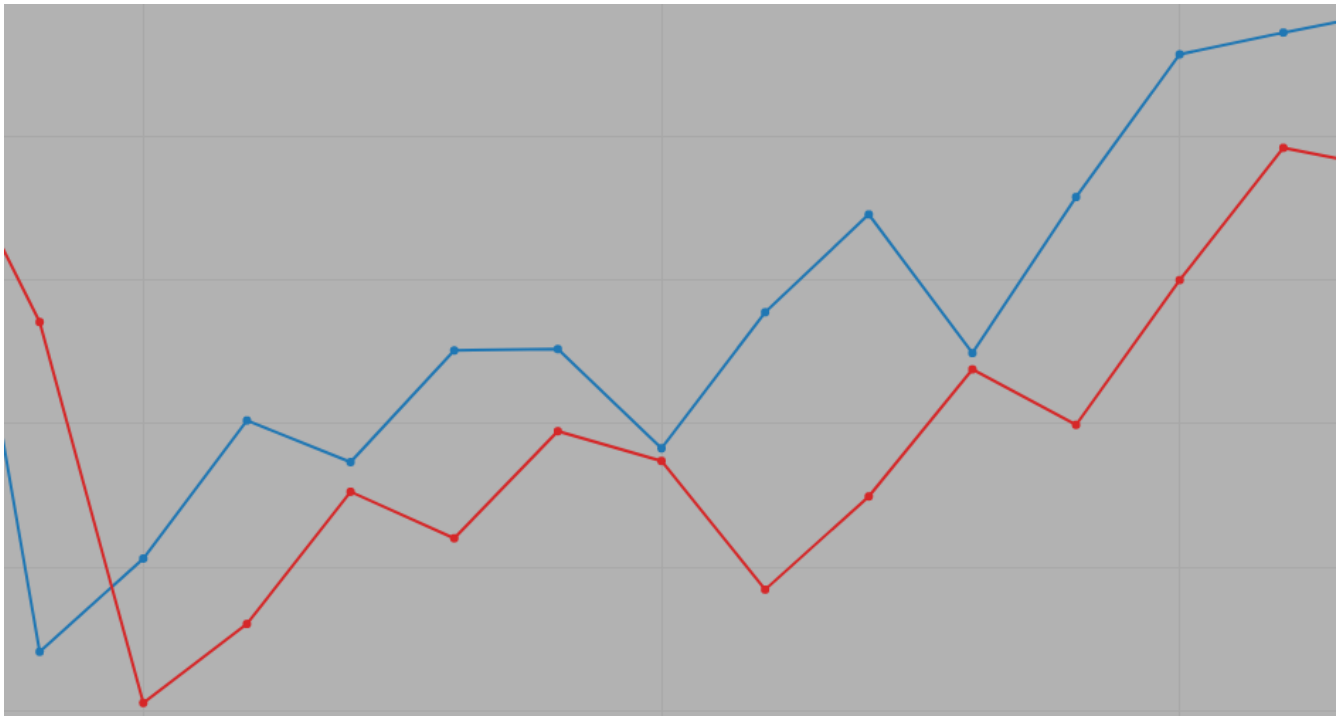
The actual vs predicted gradient for the validation+test datasets. Ideally, it would be a diagonal line.

Uh oh. For predicting whether the price will go up or down for the next candlestick (the definition of gradient here), our model is essentially no better then guessing. That's a pretty large fundamental issue. The accuracy here (51.5%) is calculated by summing the values in the correct quadrants (top right and bottom left) and dividing by all points.

Instead of telling you why this is a difficult problem (you probably already know), I'll mention two personal struggles I faced here.

1. The data. **The quality of the data determines the outcome of your model. Obviously**. Clean and process your data, understand it, play with it, plot it, cuddle it. Make sure you explore every aspect of it. For example; I use news stories. They get published in different time-zones. The stock price data comes from another time-zone. Make sure you are syncing correctly and not cheating yourself by using future information. That's just one such example. Another one: For when asking for hourly candlesticks from my broker, the first bar is a 30min window. Without checks for catching this, you're going to be scratching your head for a while.

2. Building a naive estimator. By that I mean your model's prediction is largely based on the previous point. This seems to be the most common problem in stock



A naive estimator. The red line (the prediction) follows the blue line (the actual price) with a lag of 1 data point.

Right, so in a nutshell:

> *Can we train a model that accurately predicts the next gradient change, while mitigating the naive estimator effect?*

Spoiler alert: Yes, we can! (I think).

Data

Stock price information

Most of the time spent on this project was making sure the data was in the correct format, or aligned properly, or not too sparse etc. (Well, that and the GUI I built around this tool, but that's a different issue entirely 😳).

My data comes from *Interactive Brokers* (IB). After signing up and depositing some minimum amount, you can then subscribe to various feeds. At present I pay ~15 euros

My function that makes use of *their API* to download stock prices can be seen *in this gist*.

What's important there:

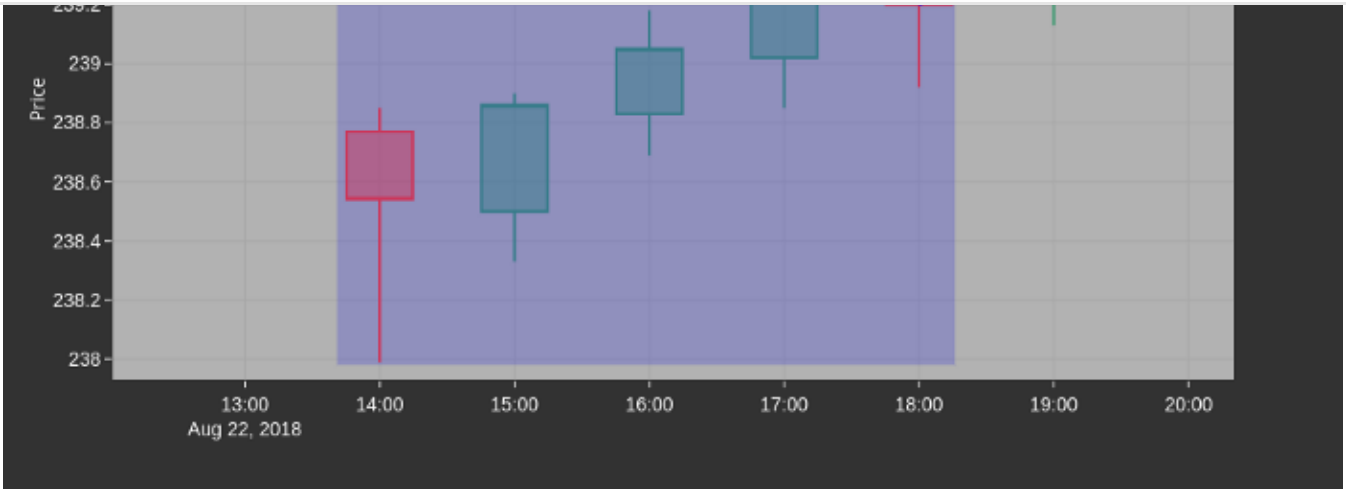
1) Connect to IB
2) Create a "contract"
3) Request historical bars using that contract.

All of this is put on a patched async loop (hence the package *nest_asyncio*), due to my code already being on a thread. The *Usage* in the above gist gives an example of how one would call this function.

I now have a pandas dataframe of 1 hour candlesticks. From there it's easy to make plots:

60 minute candlesticks for Goldman Sachs

I use the pretty awesome *Plotly* library. The slightly more involved syntax is a sacrifice for interactive plots (although not interactive for this article). By zooming in on a section, the goal can be better highlighted:



What is the gradient (i.e. sign change) from one Close price to the next? To make this prediction, everything in the shaded box (among other things) is taken into account. More on variables later. This shows a sequence of 5 candles used to predict the 6th.

I will try predict the gradient from the latest Close price that I have, to the incoming Close price. This can be used to formulate strategies for trading. At a later stage the size of the gradient could also potentially be taken into account.

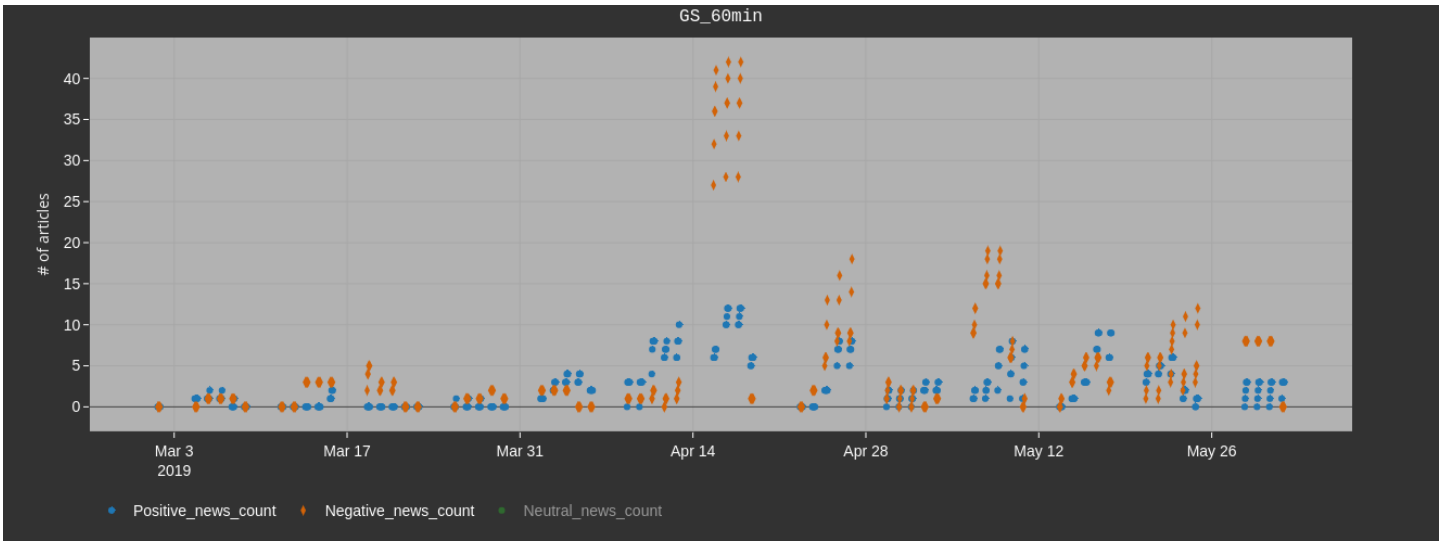
News

The hypothesis is that news has a very large impact on how stock prices evolve. There are a couple of sources for news out there, *newsapi.org* for one, IB also has some options, Thomson Reuters etc.
As for my sources, I'm not quite ready to share them yet ☺.

I currently use news sentiment in the most basic form: I count the number of positive/negative and neutral stories for a given time period and use them as features. I use my own home-rolled semi-supervised news classifier, but one could also use *BERT* or any other pre-trained library.
There are other ways to include sentiment, such as injecting the embeddings directly into the network for example.

For each stock, I chose certain keywords and retrieve the associated news articles. One hyper-parameter is "lag". A lag of 0 means that if I'm predicting the Close price at 2pm, only stories before 2pm *on the same day* are used. A lag of 1 means to include news for an extra day back, and so on. The question here is: how long does it take for news to propagate through society and trading algorithms, and how long does it's effect have on the stock?

reporting *mixed first quarter results*.



The number of positive and negative news articles with lag=2 for a given date range

Building datasets

Variables and features

One problem with predicting stock prices is that there really is just a finite amount of data. Also, I don't want to go *too* far back as I believe the nature of trading has completely changed from say 2013 till now. I can train on many or few stocks concatenated together, with others used as features. By concatenating stocks I increase the number of data, as well as potentially learn new insights. The pseudocode for my dataset builder looks like this:

```
# Specify stocks to concatenate and specify those to use as
features.
Training_assets=[...] # i.e. currencies, or other stocks
Feature_assets[...] # i.e. related stocks
For stock in Training_assets:
  Download/update stock from IB
  Merge in News sentiments
  Add extra variables like MACD, Boilinger Bands, etc.
  Download Feature_assets and append to stock
  Normalize
  Concatenate with the previous training stock
```

During training, I normalize each feature and save the parameters to a scalar file. Then, when inferring, I read the file and apply the parameters to the variable. This speeds up the process of inferring when I can just ask for the latest point from my broker. A gist of how to normalize can be seen here. An interesting parameter is the `norm_window_size`. This specifies how many points in the feature should be normalized together. A window that is too big means your resolution is not fine grained enough. A larger variety of external factors that haven't been taken into account will play a bigger role. A window too small will essentially just look like noise. It's an interesting parameter to play with.

Correlations

The correlations between each variable are shown below. Remember, in the most broad sense, two highly correlated variables means that if one increases, so will the other. For anti-correlation, if one variable decreases, the other will increase.
Higher, positive correlations are darker colors, lighter for lower/negative correlations.
Currently, I do use Open, High, Low as features. They are extremely highly correlated with the Close price, but I have all that information at inference time, so hey, why not. Future training might see me remove those to see what happens. In general, it's not good to have "repetitive" variables.
Other features that seem redundant are the indicators built with the Close price. But they give me an easy way to change the sequence length for just those variables so I left them in for now.

But what "external" sources are there (i.e., not derived from the stock(s) we're trying to infer on)? These are the most important variables.
High correlations between features such as the currencies, the indexes and an anti-correlation with the VIX are very promising.
Some currencies could be eliminated to reduce the overall network size (i.e., the USD and South African Rand don't seem like they should influence each other, but who knows), and further trainings over different variables could eliminate some of them.

It is important to remember that "…correlation is not the same thing as a trading prediction." as pointed out by Daniel Shapiro in *Data Science For Algorithmic Trading*, i.e. correlation is not causation. And so one filtering technique on the to-do list is to look at how correlations evolve over time for individual variables vs the Close price of a

Variable correlations. Currencies are technically “midpoints” instead of “close” prices.

Sliding window algorithm

At this point `pandas.head()` gives:

which shows 5 time steps, with 7 normalized features (for brevity).

Then, we create the training, validation and testing datasets.
Since this is a sequence prediction problem, we use a sliding window algorithm. The premise is shown in the figure below. X number of points (4 in the image) are used, with X+1 taken as the label and forming a new array. The window is then moved 1

Sliding window algorithm of sequence length 4, for data (X) and corresponding labels (Y).

From here, and after splitting into train, validation and test sizes (80%,15%, 5%), the data can be fed into a neural network.

Architectures

I played around with a variety of architectures (including GANs), until finally settling on a simple recurrent neural network (RNN). And so *Occam can rest in peace*. In theory, an LSTM (a type of RNN) should be better, something I need to play with again. Christopher Olah provides *a very nice article* about RNN's and LSTMs.

My model in Tensorflow (1.12) looks a little something like this (namespaces and histograms etc. removed):

```
def gru_cell(state_size):
  cell = tf.contrib.rnn.GRUCell(state_size)
  return cell

# Inputs
inputs = tf.placeholder(tf.float32, [None, seq_len, len(variables)])
labels = tf.placeholder(tf.float32, [None ,n_outputs])

# Placeholder for dropout to switch on and off for
training/inference
keep_prob = tf.placeholder(tf.float32)

# Run the data through the RNN layers
batch_size = tf.shape(inputs)[0]
```

```
range(num_layers)], state_is_tuple=False)

outputs, final_state = tf.nn.dynamic_rnn(cell, inputs,
initial_state=initial_state)

# Then feed into a dropout layer
dense_layer = tf.contrib.layers.dropout(outputs[:,-1],
keep_prob=keep_prob)

# ... and a dense layer
dense_layer = tf.layers.dense(dense_layer, dense_units,
activation=tf.nn.selu)

# ... followed by a single node dense layer
final_predictions =
tf.layers.dense(dense_layer,n_outputs,activation=tf.sigmoid)
```

The graph looks like this:

Tensorboard graph visualization (The multi_rnn namespace is connected to accuracy by a variable placeholder, batch size).

dropout percentage all are optimized during training.

The "Accuracy" node is long convoluted set of TF operations that convert a prediction from the dense network into a binary gradient movement. As an experiment, this accuracy is actually currently used in my cost function as:

```
cost = (1-Accuracy) + tf.losses.mean_squared_error(labels,
final_predictions)
```

where the *label* is the normalized price, and *final_predictions* are the normalized actual price predictions. I use the *AdamOptimiser* with a cyclic function learning rate. Is this the optimal/best way to do it? I'm not entirely sure yet! ☺

Training

I too made use of *Bayesian Optimization* (BO) during the training stage. I think it's a fantastic library, BUT, am I convinced that it works well for this type of problem and actually saves a huge amount of time? Not really. I'd like to create some plots to see how the training progresses and what the function(s) looks like. But with this many parameters it's tough. However, maybe it provides a slightly biased random number generator. With that being said, the parameters used for the results in this article are:

Final parameters used for training for the purpose of this article

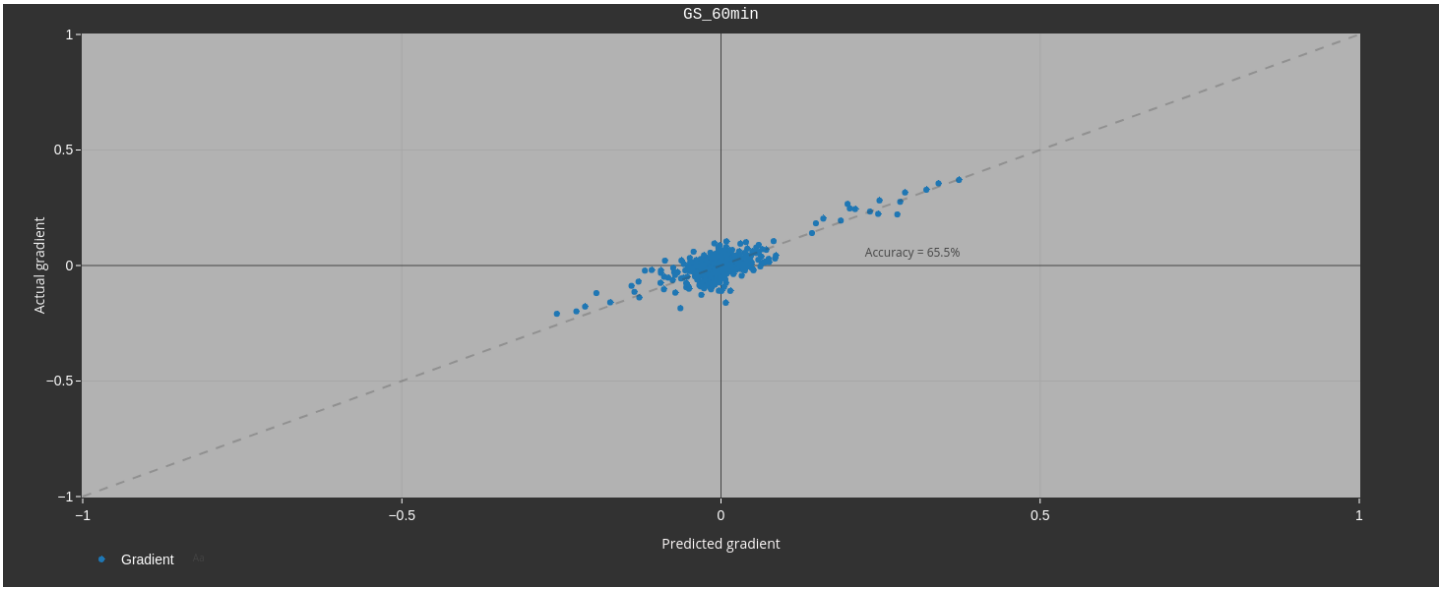
Here's an interesting read on scaled exponential linear units (selus).

The loss curve for train (orange) and validation (blue) data sets is shown below. The lines are very jumpy, and maybe using a larger batch size could help with that. There's also a some difference between the two datasets. This is not too surprising. Remember, I've concatenated (and shuffled) multiple stocks with Goldman Sachs, and so what we're actually training is a model for a given "sector", or whatever you want to call it. In theory, it's more generalizable, and so more difficult to train (the trade-off to get more data). Thus, it could hint at some over-training; something to be further checked.

Loss function for 4k iterations. Only the best model is saved.

Results

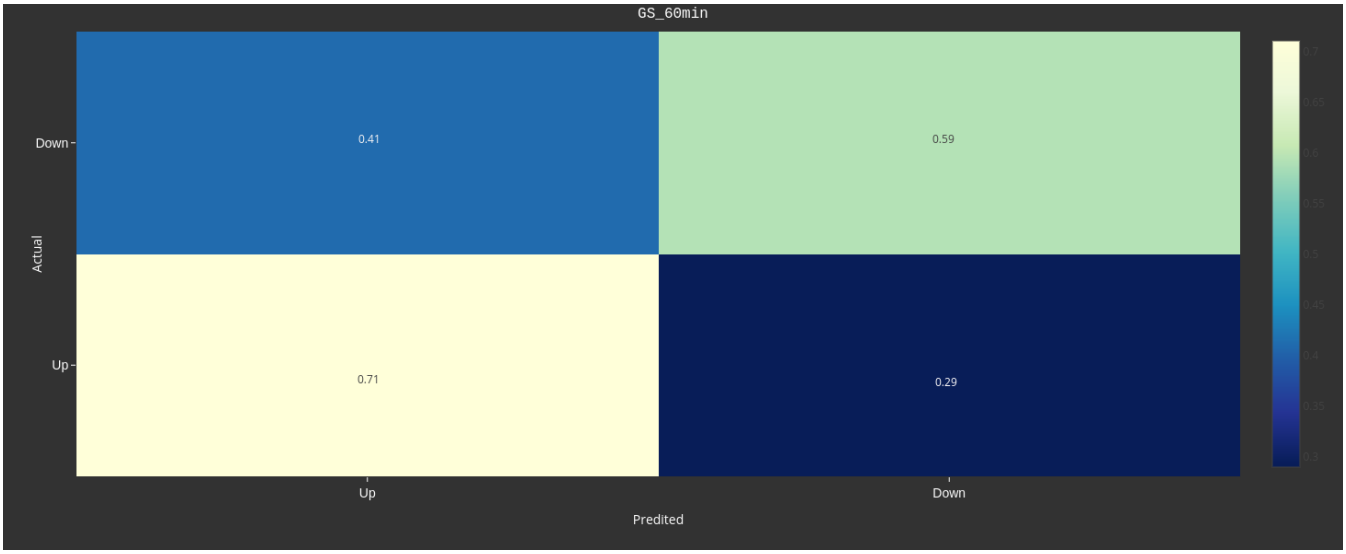
How does this latest model perform? Below is the actual gradient vs the predicted gradient. 65% accuracy (with the same definition used before) isn't too bad.



The actual vs predicted gradient for the validation+test dataset.

The figure below shows a confusion matrix for the actual gradient vs the predicted gradient. It shows that 59% of the time we correctly predict a negative gradient, while 71% of the time we correctly predict a positive gradient. This imbalance could come from the nature of the dataset and the model, i.e., maybe three small positive gradients

trading strategies.

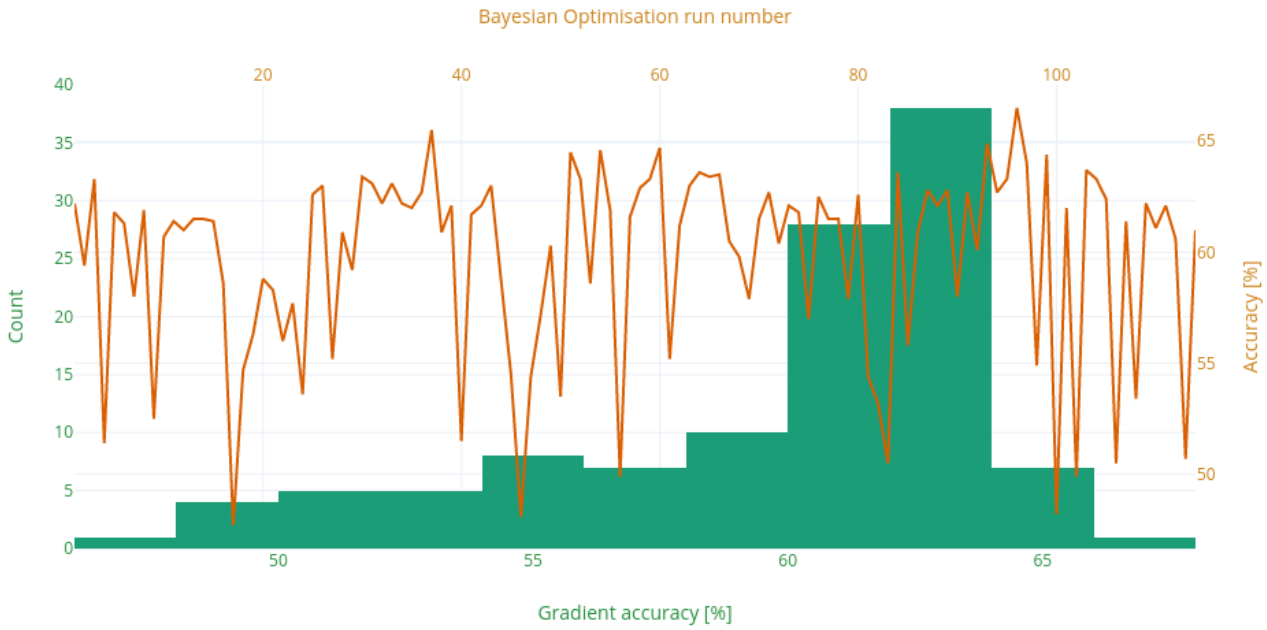


Confusion matrix showing accuracy for up and down predictions.

The cover plot is shown again, focusing on just the validation and test datasets. Let's be honest, it's not *that* sexy. But there are times when trends of gradient changes are indeed followed. To me, anytime a plot like this is shown with seemingly perfect overlap, an alarm bell should go off inside the reader's head.
Remember, the validation dataset is only used in the training steps to determine when to stop training (i.e. no improvement after x epochs, stop). The test dataset is not used *anywhere*. That means this plot shows around 600 hours of "semi-unseen" data, and just under 300 hours of completely unseen data.

Predictions for the validation and test datasets.

shown below (the histogram in green). The accuracy of each training session is plotted against run number in orange. This confirms my suspicion that BO isn't working too well here, but maybe it just needs more iterations and/or parameters tweaked.



It turns out there was actually a better result I could have used. Whoops 😀. (The better result has a more even distribution for up/up vs down/down in the confusion matrix, which is nice).

The take-away from the green histogram is that we are learning *something*. Also, it's good to see that there are some results that are no better than guessing, meaning we aren't *always* learning something when playing with parameters. Some models just suck. And if no models sucked that would be an alarm bell.

Final remarks

In this article I highlighted my means of building a RNN that is able to predict the correct gradient difference between 2 Close prices around 65% of the time. I believe with more playing around and some tweaking this number can be improved. Also, plenty more checks and studies can be performed.

Will it actually make money when backtesting? How about when trading live?

I hope you found the article interesting… I had fun writing it!

Joshua Wyatt Smith

Trading Quant Recurrent Neural Network Machine Learning Stock Market