Deep Learning For Time Series

Neural Networks

This section introduces some necessary background information on neural networks, much of it closely based on [1] and [2]. Suppose we have a collection of sample data

$$\mathbb{X} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n \times m}$$

Each column of $\mathbb{X}$ contains a different sample, which is a vector consisting of measurements of n features, i.e. $x^{(i)} \in \mathbb{R}^n$ for each of our m samples. Suppose we also have m corresponding output observations contained in a vector $y \in \mathbb{R}^m$. For our purposes, a neural network can be thought of as a supervised machine learning model that aims to approximate an assumed underlying relationship between the model inputs and outputs of the form $y = f^*(x)$. For a given input x, a trained network can provide an estimate, or forecast, $\hat{y} = f(x; \theta)$ of $f^*(x)$. Here θ is a vector of parameters to be learned by the model during training. As is usually the case for a supervised learning algorithm, these parameters will be learned through the minimisation of some loss function $\mathcal{L}(\hat{y}, y)$.

The above description applies to a wide variety of supervised learning algorithms. The key aspect that is unique to neural networks is the form of the function f: it is constructed through the use of 'layers' of 'neurons' that compute 'activation' functions on their inputs and pass the results on to the following layer. Such a network may consist of many layers, each of which possibly containing a different number of neurons. The first layer of the network will contain n neurons, or 'nodes', corresponding to the n features from each sample. The final layer of the network, or output layer, will tend to consist of one neuron containing the model's prediction of the output's value or class, depending upon whether we are working with a regression or classification problem. The layers in between these two are known as hidden since they are not visible external to the network. Figure 1 illustrates this for a network with four inputs, two hidden layers of three neurons each and one output. These types of networks are known as feedforward networks due to the way that information is propagated through the network.

With some basic definitions given, let us examine a network in more detail.

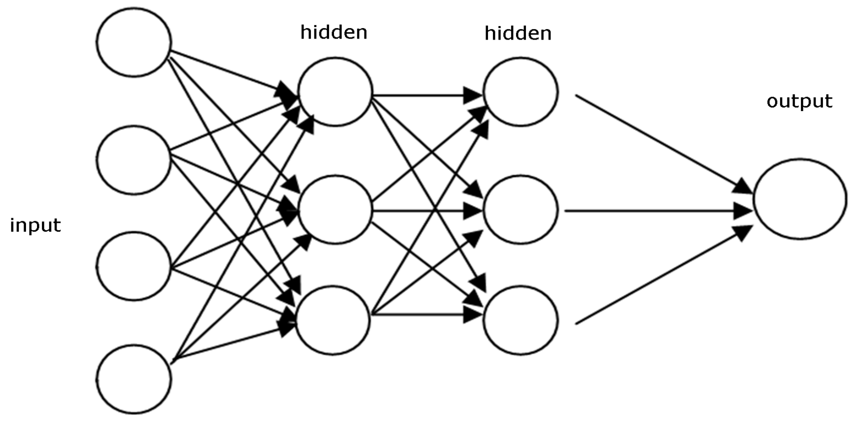

Figure 1: Feedforward Neural Network

Fix an input sample x. We may suppose that our network consists of L layers (not including the input layer) and that the lth of these layers contains n_l neurons. Starting at the first hidden layer and considering the first neuron for concreteness:

- The neuron has an associated weight vector, $w_1 \in \mathbb{R}^n$ and bias $b_1 \in \mathbb{R}$
- The affine linear combination $z_1 = \langle w_1, x \rangle + b_1$ is formed
- A function g is used to compute the neuron's activation: $a_1 = g(z_1)$
- The above is repeated for each neuron in the first hidden layer
- The vector of activations $a \in \mathbb{R}^{n_1}$ will be passed to each of the n_2 neurons in the second layer of the network
- The process continues until the final layer where a final 'activation' function produces an output

The function g above, usually known as an activation function, will be fixed within each layer but can vary across layers. Often the function g is chosen to be non-linear. Indeed, if g were linear in each layer then we would effectively be forming an affine linear combination of the inputs after passing through the whole network. This means our network would be redundant in the sense that the same result could be achieved by using a simpler model such as a linear or logistic regression, depending on certain features of the networks.

The most common choices of activation function are the ReLu, sigmoid and hyperbolic tangent (tanh), pictured in figure 2. Here are their definitions for reference:

$$\text{relu}(x) := \max(x, 0),\ \sigma(x) := 1/(1 + e^{-x}),\ \tanh(x) := (e^x - e^{-x})/(e^x + e^{-x})$$

ReLu's have been found to be very effective for feedforward networks. Sigmoid units - neurons whose activation function is σ - are often used if a probabilistic interpretation is required since their range is $(0,1)$. A well known example would be logistic regression. Tanh units will be encountered when we consider Long Short-Term Memory networks, or LSTMs, later.

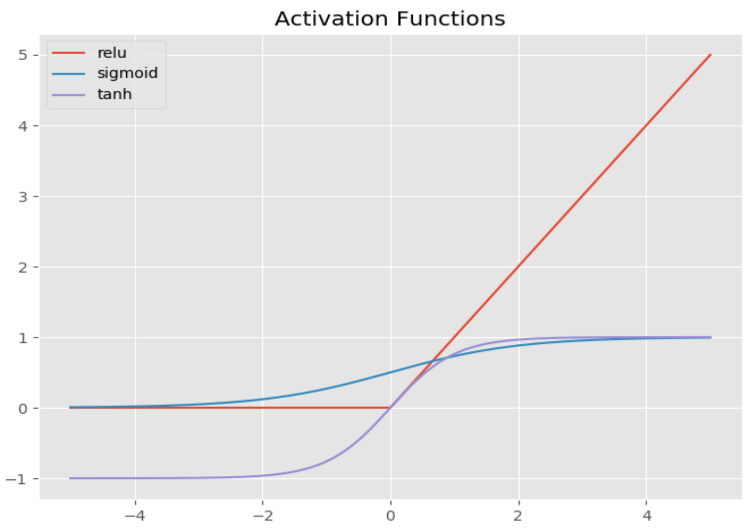

Figure 2: Activation Functions

To keep track of the many parameters contained within a model, which can number in the millions for networks with many layers of many neurons, it is convenient to introduce matrices and vectors for their representation. I will use the notation $W^{[l]}, b^{[l]}, a^{[l]}$ for the weights, bias and activation of the lth layer, respectively. Using the convention that the initial activation is the input, $a^{[0]} = x$, we can represent the process for any layer through the matrix equation:

$$a^{[l]} = g(W^{[l]}a^{[l-1]} + b^{[l]})$$

This representation is also helpful when considering the back-propagation algorithm used for updating the model parameters.

Recall that the model is trained through minimisation of an error functional, $\mathcal{L}$. Common forms of $\mathcal{L}$ might be:

- Average sum of squared errors in a regression problem:

$$\mathcal{L}(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$

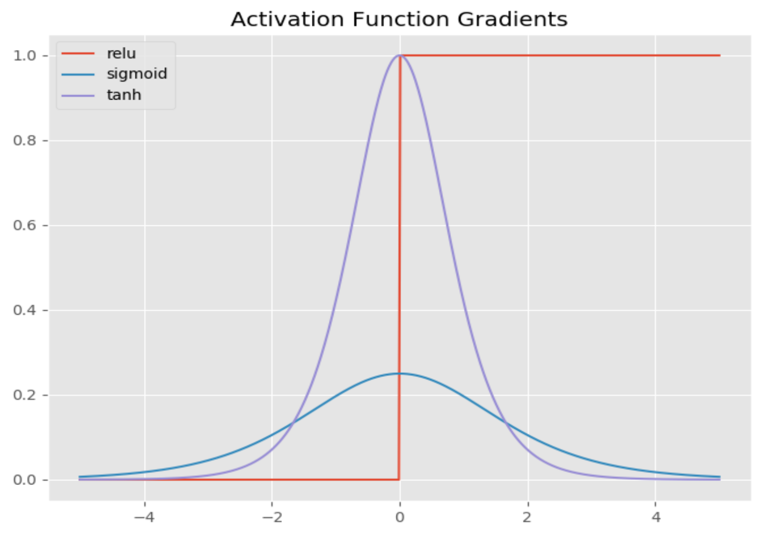

Figure 3: Activation Function Gradients

- Cross-Entropy for a classification problem with two classes:

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y_i}) + (1 - y_i)\log(1 - \hat{y_i})]$$

As an aside, note that the cross-entropy follows from a Maximum Likelihood Estimator (MLE) argument. Indeed, suppose we have a Bernoulli random variable with probability parameter p and we observe a collection of classifications y_i. To estimate p, the probability of belonging to the 'positive' or '1' class, we may maximise $\mathbb{P}(y_1, \ldots, y_m) = \prod_{i=1}^{m} p^{y_i}(1-p)^{1-y_i}$. Due to the monotonicity of log it is equivalent to minimise the negative logarithm, giving the above expression.

Gradient Descent

To minimise this loss function, which is a non-convex function of the model parameters, the method of gradient-descent is used. Recall that for a given function $f : \mathbb{R}^N \to \mathbb{R}$ and a vector $v \in \mathbb{R}^N$, the directional deriavtive of f at x in direction v is given by $D_v f(x) = \langle v, \nabla f(x)\rangle = ||v||_2 ||\nabla f(x)||_2 \cos(\phi)$. It follows that the direction in which f's gradient decreases most rapidly at the point x will be $-\nabla f(x)$. The gradient descent method seeks to minimise the non-negative loss as a function of the parameters $\theta = (W^{[l]}, b^{[l]})$ and hence training will proceed by making updates of the form:

$$W^{[l]} := W^{[l]} - \nabla_{W^{[l]}}\mathcal{L}, \quad b^{[l]} := b^{[l]} - \nabla_{b^{[l]}}\mathcal{L}$$

Practical considerations lead to so called mini-batch gradient descent generally being preferred over batch or stochastic gradient descent. These terms refer to the number of training samples used in each update. Batch gradient descent proceeds by using all m samples for each parameter update, i.e. the gradient, say $\nabla_{W^{[l]}}\mathcal{L}$, will be computed using all m samples in the loss: $\mathcal{L} = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}_i, y_i)$. These updates can be very time consuming when a large amount of training data is used. At the other extreme is Stochastic Gradient Descent (SGD), which will randomly choose just one sample to use for the gradient estimation each time parameters are updated. The Mini-batch approach computes each gradient from a sub-sample of m' data points at each update. This tends to allows local minima to be identified more quickly than with batch gradient descent while keeping the benefits of vectorisation that are abandoned with SGD. The size of each mini-batch, m', is usually chosen to be a power of 2 for practical reasons.

Recurrent Neural Networks (RNN)

This section follows [2] and [3]. We will be interested in applying neural network models to data in time series form. Recurrent neural networks were specifically designed for such tasks. For a classic feedforward network as discussed above, the activation of a neuron is only used by the network for the one sample in question. For time series data it can be very useful for the network to 'remember' a state triggered by what it has seen recently, helping it capture some of the temporal characteristics of the data. This intuition is materialised by allowing a neuron to feed its activation at time t, denoted $a^{<t>}$, into itself when predicting output at the next time step. In a recursive way, if the activation at time t contains information about the activation at time $t-1$, then so must the activation at time $t+1$.

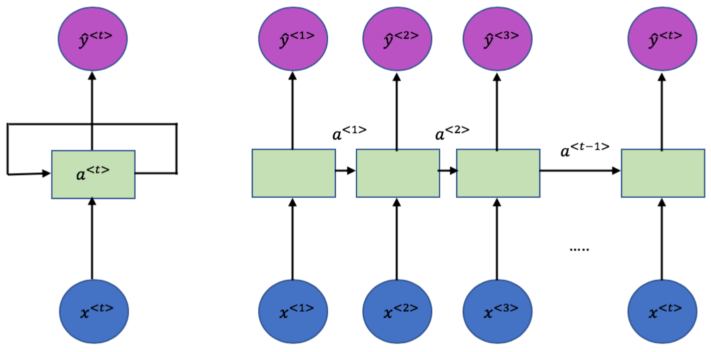

Figure 4: Recurrent Neural Network

Figure 4 shows two diagrammatic representations of this idea. Let $x^{<t>}, \hat{y}^{<t>}$ denote the input and predicted output at time t. The 'rolled up' figure on the left illustrates the loop of the activation in determining both the prediction value and the next activation value. The right side of the diagram is an 'unrolled' representation of the same behaviour, in which we can explicitly see how the input is passed in at time t and is used together with the activation from the previous time-step to produce the current activation of the neuron. This encapsulates the basic idea behind a recurrent neural network, although specific architectures can be much more complex.

To put the above idea into symbols, we will use the notation $W_\alpha = [W_{\alpha a}, W_{\alpha x}]$ for horizontally stacked matrices. For an arbitrary variable α, this allows us to introduce the shorthand

$$W_\alpha \left[a^{<t-1>}, x^{<t>}\right] = W_{\alpha a} a^{<t-1>} + W_{\alpha x} x^{<t>}$$

Then for a recurrent neural network with activation function g we can write

$$\begin{aligned}\hat{y}^{<t>} &= g\left(W_a \left[a^{<t-1>}, x^{<t>}\right] + b_a\right)\\ &= g\left(W_{\alpha a} a^{<t-1>} + W_{\alpha x} x^{<t>} + b_a\right)\end{aligned}$$

If we are trying to use a long temporal sequence for prediction, or predict a number of steps into the future, then RNNs can run into trouble. Such a case might be trying to use the last 200 feature observations in predicting the next output. Goodfellow, et al, in [1] examine the simplified setup in which a diagonalisable weight matrix W is repeatedly applied, i.e. as a simplified activation function. A similar situation could also be seen in a deep feedforward network. In such a case we know that $W = V\Lambda V^{-1}$ and hence $W^t = V\Lambda^t V^{-1}$, where Λ is the diagonal matrix of eigenvalues and the columns of V are W's eigenvectors. This tells us that, as t increases, any eigenvalue whose absolute value is less than 1 will approach 0 and any eigenvalue whose absolute value is bigger than 1 will blow-up to infinity. This argument gives some intuition behind the problems that occur with training such networks - gradients may blow up or tend to zero. The former case will cause instability in the learning algorithm, while the latter will make it difficult for the network to improve during training - it is not clear in which direction the parameters should be updated.

One reason behind the success of LSTM models is that they address this issue. Briefly, the idea is that the gradient can be controlled, and hence propagated for longer, by having cells recurrently connected to each other instead of the usual hidden units. These cells regulate the cells activation, in part by gating the information that flows in and out of the cell at each update. Figure 5, taken from [3], conveys the idea: the sigmoids are gates that can cause information from the cell's state to be forgotten or filtered, while the tanh functions are used to update the cell's state and the activation based on this.

In terms of symbols, let us denote the update, forget and output gates by Γ_u, Γ_f and Γ_o, respectively. Then for $\alpha = u, d, o$, we have

$$\Gamma_\alpha = \sigma\left(W_\alpha \left[a^{<t-1>}, x^{<t>}\right] + b_\alpha\right)$$

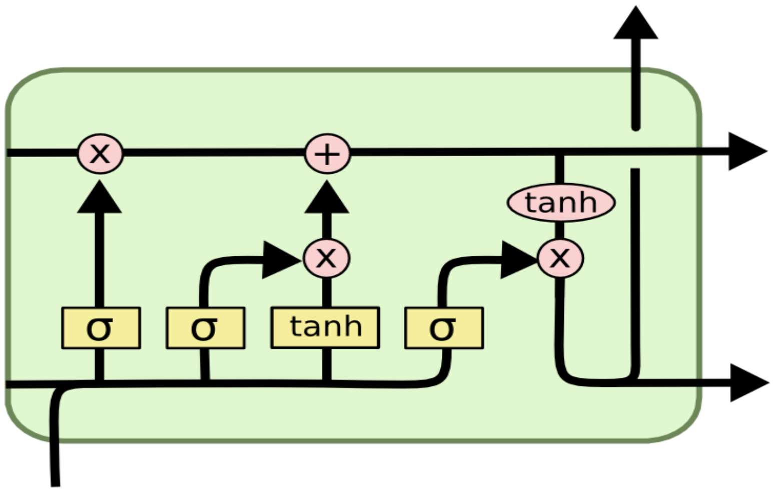

Figure 5: Long Short-Term Memory Cell

and the LSTM update equations can be written as

$$\begin{aligned}
c^{<t>} &= \Gamma_f \odot c^{<t-1>} + \Gamma_u \odot \tilde{c}^{<t>} \\
\tilde{c}^{<t>} &= \tanh\left(W_c\left[a^{<t-1>}, x^{<t>}\right] + b_c\right) \\
a^{<t>} &= \Gamma_o \odot \tanh(c^{<t>})
\end{aligned}$$

Data

Input Data

I have attempted to predict the direction of the price movements of the two Spanish listed financial stocks Banco Santander (SAN) and Banco Bilbao Vizcaya Argentaria (BBVA). For source data I have taken the below features from Bloomberg:

1. Prices: OPEN, CLOSE, HIGH, LOW
   Both stocks are quoted in EUR and hence so are all price fields.

2. Volumes: VOLUME
   Raw volume figures are shares traded per day, not dollar volumes.

3. CDS: 5Y SENIOR
   I have used 5y senior CDS quotes as they cover the most liquid product.

4. Fundamental: PTB RATIO, GTR INDEX
   To avoid dealing directly with dividends, which may need to be adjusted for historical corporate actions, I have chosen to measure a proxy by looking at the total return from an investment in the security.

All fields range from 14/09/2001 - 18/06/2020, i.e., almost 19 years of data.

Input Features

Model input features can be roughly separated into the 6 below categories. I use the notation $S = \{1, 2, 3, 4, 5, 10, 21, 50\}$, $T = \{20, 40, 60, 80, 100, 200\}$ and $kS = \{kx : x \in S\}$ when listing the periods considered to save space:

1. Basic functions of price

   i) Returns: $r_t^d = \log(p_t/p_{t-d}),\ d \in S$.

   ii) Price Momentum: $mtm_t^d = p_t - p_{t-d},\ d \in S$.

   iii) Simple Moving Average (SMA): $sma_t^d = \frac{1}{d}\sum_{s=0}^{d-1} p_{t-s},\ d \in 5S$.

   iv) Sign of Return: $\operatorname{sgn}(r_t^d) = 2 \times \mathbb{1}(p_t > p_{t-d}) - 1,\ d \in S$.

2. Volatility

   i) Close-Close (standard) Volatility: $\sigma_t^d = \sqrt{\frac{1}{d-1}\sum_{s=1}^{d-1}(r_{t-s}^1 - \hat{r}_t^1)^2}$, where $d \in T$ and $\hat{r}_t^1$ denotes the usual empirical mean of the (d day) period ending at t.

   ii) Yang - Zhang, or driftless, Volatility:

$$v^d(t) = \sqrt{V_o^d(t) + k\,V_c^d(t) + (1-k)V_{rs}^d(t)}$$

   This definition of volatility accounts for, and is independent of, both drift and opening jumps in a stochastic model for security prices. It is shown in [4] that v_t^d is an unbiased, minimum variance - necessarily multi-period - volatility estimate for such a model. Note that both drift and overnight jumps are not accounted for in the standard estimate above. For reference I include all the relevant notation below:

$$V_o^d(t) = \frac{1}{d-1}\sum_{s=0}^{d-1}(o_{t-s} - \hat{o}_t^d)^2, \quad V_c^d(t) = \frac{1}{d-1}\sum_{s=0}^{d-1}(c_{t-s} - \hat{c}_t^d)^2$$

$$V_{rs}^d(t) = \frac{1}{d}\sum_{s=0}^{d-1}[u_{t-s}(u_{t-s} - c_{t-s}) + d_{t-s}(d_{t-s} - c_{t-s})]$$

$$o_t = \log(O_t/C_{t-1}),\ \ u_t = \log(H_t/O_t),\ \ d_t = \log(L_t/O_t),\ \ c_t = \log(C_t/O_t),$$

   where O_t, C_t, H_t and L_t denote the open, close, high and low price of the underlying security on day t, respectively. $d \in T$.

3. Volume

   i) Median Volume: $\operatorname*{median}_{0\le s\le d-1} vol_{t-s},\ d \in T$.

   ii) Volume Momentum: $vol_t - vol_{t-d},\ d \in T$.

4. CDS

   i) CDS Return: $\log(cds_t/cds_{t-d}),\ d \in S$.

   ii) CDS Momentum: $cds_t - cds_{t-d},\ d \in S$.

5. Technical Indicators
   In some of the below definitions I have neglected multiplicative constants as all features will be normalised before being used in training.

   i) Stochastic Oscillator:
   For $d \in [5, 14]$, define rolling lows, highs and the SO indicator by

$$l_t^d = \min_{0\le s\le d-1} L_{t-s},\ h_t^d = \max_{0\le s\le d-1} H_{t-s},\ SO_t^d = (p_t - l_t^d)/(h_t^d - l_t^d)$$

   ii) RSI - Relative Strength Index
   $RSI = RS/(1 + RS)$, where $RS = sma_t^d(U)/sma_t^d(D)$ and U, D are the absolute positive and negative price increments over the time window in question. For example,

$$U = (\mathbb{1}(p_{t-s} > p_{t-s-1}) \times p_{t-s} - p_{t-s-1})_{0\le s\le d-1},$$

   except that zeroes are ignored from the calculation. $d = 14$.

   iii) MACD - MA Convergence/Divergence:
   The difference between a short and a long-term price EWMA:

$$macd_t^{d_s,d_l} = ewma_t^{d_s} - ewma_t^{d_l},\ d_s = 12,\ d_l = 26.$$

   iv) CCI - Commodity Channel Index:
   The typical price of a security on day t is defined to be

$$tp_t = (C_t + H_t + L_t)/3$$

   Letting $mad_t^d(tp)$ denote the mean absolute deviation of the typical price over a d-day window, the CCI is defined by

$$CCI_t = (tp_t - sma_t^d(tp))/mad_t^d(tp), d = 20$$

   v) ATR - Average True Range:
   Using the same notation as for driftless volatility, define

$$TR_t = \max(H_t - L_t, H_t - C_{t-1}, C_{t-1} - L_t)$$

   Then $ATR_t = (1/d)TR_t + (1 - 1/d)ATR_{t-1}$, an EWMA of the true range. $d = 14$.

vi) AD - Aggregation/Distribution:
Define the Close Location Value as a proxy for flow:

$$clv_t = ((C_t - L_t) - (H_t - C_t))/(H_t - L_t)$$

Then $AD_t = AD_{t-1} + vol_t \times clv_t$ is a kind of volume weighted indicator of order flow.

6. Fundamental

   i) Price to Book Ratio: ptb_t

   ii) Dividend Proxy: dvd_t
   Defined to be the difference between a gross total return investment in a security and its price return over the same period. This may be a very poor proxy.

Practicalities

I have considered the same two-class, one day estimation problem as seen in the previous assignment. In this instance, days on which the price of either security equals its value on the previous day were removed from the price series, for both securities, before other features were computed. This removal was propagated throughout all features, e.g., CDS quotes, volumes, etc. All NaNs were removed from the feature set, for instance those arising at the beginning of a rolling calculation window such as 100 day median volume. Following these adjustments, the resulting data set consists of 82 features per security ranging from 27/09/2002 to 17/06/2020.

Data Exploration

First Look

Table 1 shows the numbers of available data points and their composition in terms of the up and down classes, i.e., whether the close was higher or lower than that achieved on the day prior. Both stocks are fairly well balanced although a roughly 1% deviation from an equal split may justify using stratified sampling as part of cross validation.

	Count	Up Moves	Down Moves	Up Ratio	Down Ratio
SAN	4371	2227	2144	50.95%	49.05%
BBVA	4371	2147	2221	49.15%	50.85%

Table 1: Up and Down Moves

As was noted in the previous assignment, considering certain features over a range of data windows leads to some redundancy and the final model performance will likely be improved if we can restrict ourselves to a collection of features that are less interchangeable. To get a basic idea of the 'dimensionality' of our feature set we can consider a principal component analysis of the feature matrix. Figure 6 shows a plot of the cumulative contribution to variance of the eigenvectors for Santander. We see that around 20 features account for 90% of the overall variance. This contrasts with what was seen previously when considering only volatility and simple functions of price as features, i.e., including more data types and more complex 'signals', as in the case of the technical indicators, has given us information that we did not possess before.

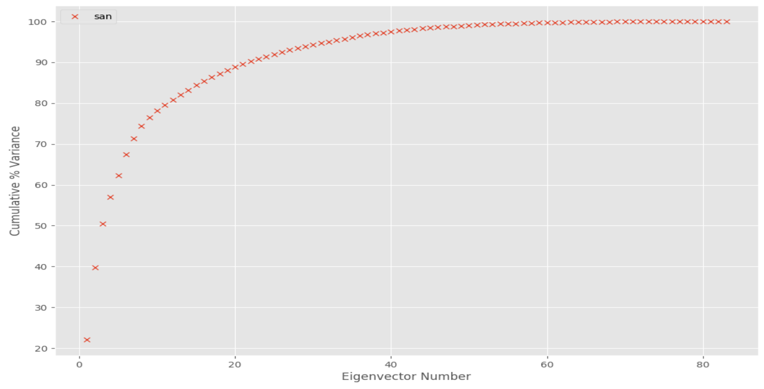

Figure 6: PCA Contributions to Variance - SAN

Feature Importance Through Decision Trees

To reduce the number of features entering the model, and hence reduce the chance of the model being overfit, I have trained decision tree classifiers and then extracted the feature importance information. This can serve as a guide to the most impactful features that should be used for the NN models. The Gini coefficient as opposed to mutual information was used for determining splits. Random forests were used to reduce model variance. The following hyper-parameters were considered:

1. Number of Trees in Forest: $[64, 128, 256]$
2. Number of Features to Select: $[5, 10, 15]$
3. Maximal Number of Features per Type: $[1, 2, 3, 5]$

The third point above was introduced as the initial results showed that the top features, in terms of importance, were heavily concentrated amongst volume momentum, CDS returns and CDS momentum. Figure 7 gives an example

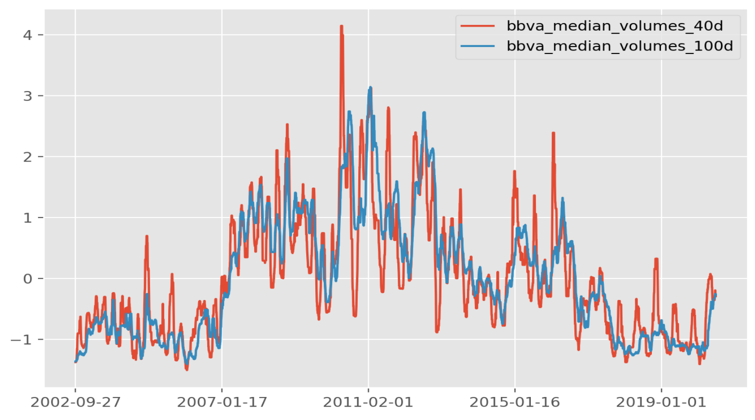

Figure 7: BBVA - Normalised Median Volumes

showing that there may be significant differences in these intra-feature families; in this instance showing that the median volume profile can vary quite widely with different window lengths. However, there are many examples of families of features, e.g. CDS returns, where correlations are fairly close for similar horizons and we would not want to select overly from such a feature family. Overall, I think such a cap on the maximum number of features that can be selected from any one family should increase model robustness.

The best performing sets of parameters and features for the two securities are displayed in tables 2 and 3 below. We see that the choice of top 10 features with a maximum of 2 features per family was the best performer for both securities. It must be stated, however, that the margin of victory was quite small. Indeed, less than 4 percentage points separate the very best from the worst model. The number of trees used to grow the random forest seemed to only have a minor impact on outcomes.

More robust than the accuracy levels themselves were the features chosen as being most important, at least in terms of the allotment per family. As mentioned above, volume momentum, CDS return and CDS momentum were all high on the list. Several of the technical indicators were also favoured - RSI and the stochastic oscillator for Santander. Note that no volatility features made it into the top 10. Similarly, no long term features were included, i.e. those with window greater than 10 days. This might be expected as we are dealing with 1 day prediction. Note also that the particular choice of window length is probably not particularly important - for SAN we see that two and four day volume momentum are chosen, whereas for BBVA it is the one and three day variants.

Note that data was normalised prior to training so less volatile features should not have been penalised.

	SAN	BBVA
Number of Features	10	10
Max Features per Family	2	2
Random Forest Trees	64	128

Table 2: Random Forest Feature Selection

To quantify the relative importance amongst the chosen features, we can re-train the random forests with the feature set restricted to what we know will be the outcome. The advantage of this approach is that importance is spread evenly between similar features and hence when the full array of data is used there is a corresponding dampening of the importance ascribed to any one feature that comes from a larger family. The final figures are shown under the headings '$< Ticker >$ Score' in 3. We still see a fairly even split amongst the features. Note that ordering can and does change when less vital features are removed from the process.

	SAN	SAN Score	BBVA	BBVA Score
1	Volumes Mtm. 4d	10.86%	Volumes Mtm. 1d	10.79%
2	Volumes Mtm. 2d	10.07%	Volumes Mtm. 3d	10.67%
3	RSI 14d	10.38%	Returns 1d	9.88%
4	Stochastic Osc. 5d	9.59%	RSI 14d	10.18%
5	Mtm. 10d	9.34$	CDS Returns 2d	9.75%
6	CDS Returns 2d	10.28%	Returns 10d	10.81%
7	Returns 10d	9.24%	CDS Mtm. 2d	9.61%
8	CDS Returns 5d	10.84%	Mtm. 1d	9.39%
9	Stochastic Osc. 14d	9.48%	CDS Returns 1d	9.51%
10	CDS Mtm. 1d	9.91%	CDS Mtm. 1d	9.40%

Table 3: Most Important Features

Neural Network Results

As was seen in table 1, the naive approach of always picking the most common direction will result in accuracies of 50.95% and 50.85% for SAN and BBVA, respectively. For the neural network approach to be worthwhile in this situation, these are the benchmarks that must be beaten.

Experiments

I compared results from the following scheme to get an idea of the effectiveness of neural networks for predicting 1 day forward price direction and identify which parameters, if any, could be tuned to improve performance:

1. Network Type - Feedforward, LSTM
   In the initial round of tests the feedforward architecture consisted of two dense layers of 16 neurons with ReLu activations and a sigmoidal output. The LSTM architecture consisted of one LSTM layer with 16 neurons, one 20% dropout layer and a sigmoidal output. The choice for number of neurons was somewhat arbitrary. The choice of adding a dropout layer to the LSTM was in response to some ad hoc trials that had shown strong overfitting for the LSTM.

2. Data Features - Full feature set, Top DT Features
   DT features were those determined through the use of Random Forest classifiers in the previous section. For both securities the chosen number was 10 while there were 82 raw features. I tested both datasets to see if the feature extraction had been successful.

3. Number of Epochs
   Recall that an epoch is an iteration over the entire training set. This notion is important since we are using mini-batches for optimisation.
   The majority of experiments covered $5, 10$ and 25 epochs. This was partially due to time constraints but also due to ad hoc tests that showed large numbers of epochs were accompanied by more overfitting and no improvement in accuracy.

4. Batch Size
   Sizes $8, 16, 32, 64$ and 128 were compared. A few tests examining larger batch sizes were also included.

5. Window Length (for LSTM)
   The number of previous data points to make use of during prediction. Figures considered were $1, 5, 10, 20, 60$.

Initial Findings

Several clear findings from these first tests were:

1. The best models were all for BBVA with the highest achieved test accuracy being 54.76% while that for SAN was 53.51%. These results are significantly better than the standard largest class benchmark, i.e., the result of always predicting the most common direction.

2. The top 75 results, in terms of accuracy, came from LSTMs indicating that the approach probably is more successful at such a time series prediction task than a network with no memory. We might have expected this.

3. Initial enthusiasm for high accuracy may have been misplaced. A closer look at the data shows that a large proportion of the models, particularly those performing well on the accuracy measure, were very one-sided in their predictions - almost always predicting down moves. Looking at the average prediction, with the classes being $\{0[down], 1[up]\}$, showed that all but one of the top fifty results had an average prediction below 50% and all but two were below 25%.

   There appears to be a relationship between batch size and average prediction, as displayed in table 4. This suggests that using larger batch sizes may improve future results, at least in the sense of making them more robust. The issue also seems to be more pronounced for Santander, with its average-average prediction being 13.18% versus 24.41% for BBVA. This bias seems likely related to the fact that the test period coincides roughly with the last two years of market data. Down moves for the securities in question have been notably more regular during this period.

4. In terms of epochs, 25 seemed a slightly more reasonable choice than the other values, especially for BBVA. However, the evidence to support this is not very strong. More epochs will likely be required for more complex network architectures.

Batch Size	Observations	Average Prediction
8	203	16.76%
32	192	15.00%
128	191	16.00%
512	98	35.77%

Table 4: Batch Size and Skewed Predictions

To further elaborate on point 2 above, restricting results to those models whose average prediction was between $40 - 60\%$, a rough 10% band around the true data averages, shows that the two securities were more evenly matched in terms of their predictability. The best model is still for BBVA with an accuracy of 53.96% while that for SAN had an accuracy of 52.78%.

With this filter in place, all of the top models are ones utilising the full features set. Thus it seems that my feature extraction process was not at all effective. Possible interesting follow up tests would be feature selection at the neural network level, perhaps through comparing training and validation metrics, i.e. like a form of cross validation.

Validation

A 20% validation set was held out during the above tests. This can be used to compare how reduction in training loss is translated into a reduction of loss on unseen data. Similarly, it can be used to test whether increases in training accuracy are being carried over into increases in accuracy on unseen data. Figures 8 and 9 show the loss and accuracy curves for highest scoring BBVA model.

We can clearly see that while the training loss is reducing at an accelerating rate, the validation loss is roughly constant throughout the training process, even increasing toward the end. The shapes of the accuracy curves are more encouraging, with both increasing as training progresses. However, it should be noted that the level of accuracy seen in these first few epochs is quite low and so the strong test performance is likely due to chance.

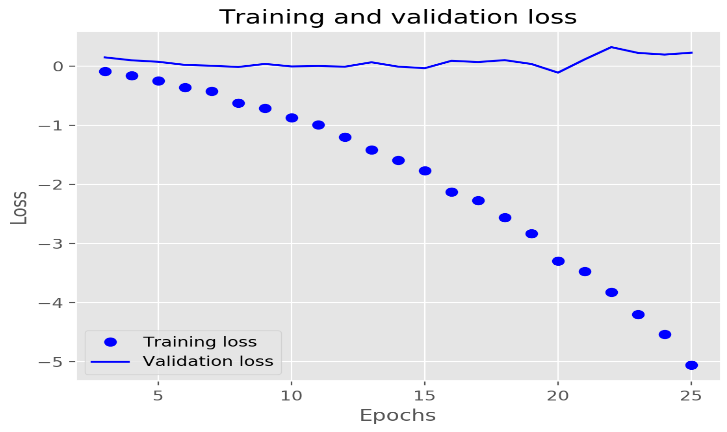

Figure 8: BBVA - Loss Function

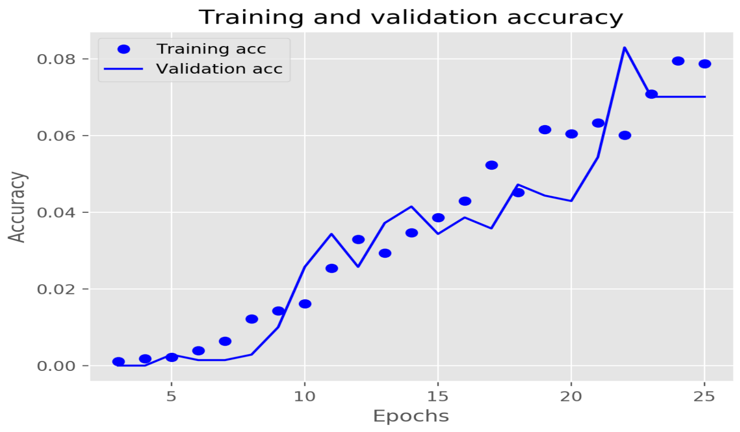

Figure 9: BBVA - Accuracy

Conclusion

First results suggest that LSTMs can be more accurate than feedforward networks and, indeed, any of the other supervised learning techniques considered in this project and the one before. However, I think much more testing would be required before a robust model is discovered. One major problem seems to be the choice of features. After this has been addressed, there are many particularities of architecture, learning rate and regularisation that might improve the predictions further.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[2] Andrew Ng. Deep learning specialization. `https://www.coursera.org/learn/nlp-sequence-models/home/welcome`, 2020.

[3] Christopher Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015.

[4] Dennis Yang and Qiang Zhang. Drift-independent volatility estimation based on high, low, open, and close prices. *The Journal of Business*, 73(3):477–492, 2000.