**3.2 Python packages – Explanation of the Python packages that will be used to implement Neural Nets**

**3.2.1 Implementation – Code and issues surrounding implementation**

The implementation of the RNNs was carried out using the Python library Keras. Keras runs on top of numerical computation and large-scale machine learning libraries TensorFlow, CNTK or Theano.

Gathering the data was done by accessing Quandl APIs, for price data and accounting metric data. Price data needed to be manipulated using the Pandas library into firstly monthly price from a daily pricing dataset then changed into monthly returns. Returns being the change (positive or negative) in the monthly share price.

Factor data (accounting metrics) was also returned from a Quandl API. As not all factor data is available on a monthly basis quarterly data was maintained until new data became available. Returns data is then added to the factor data. After running a correlation matrix and factors with a correlation of > 0.95 are dropped to reduce model complexity.

As Keras only accepts data in the form of arrays the Pandas dataframe is amended to a Numpy array. As the factor data is composed of different scales these values are normalised to that they take small values on a similar scale. The SKlearn module MinMax scalar is used to pass a feature range between 0 and 1 for the data.

3.2.2 Training the RNN models

A neural network is based around four objects. Layers (combined into a network or model), input data and corresponding targets, loss function that defines the feedback signal used for learning and the optimiser which determines how learning proceeds. A layer is a data-processing module that takes as input one or more tensors and outputs the same. Some layers are stateless but more often that have a state: the layer’s weights, one or several tensors learned with stochastic gradient descent, which together contain the network’s knowledge.

Different layers are appropriate for different tensor formats and different types of data processing. For instance, simple vector data, stored in 2D tensors of shape (samples, features), is often processed by densely connected layers, also called fully connected or dense layers (the Dense class in Keras). Sequence data, stored in 3D tensors of shape (samples, timesteps, features), is typically processed by recurrent layers such as an LSTM layer. Image data, stored in 4D tensors, is usually processed by 2D convolution layers (Conv2D).

The dataset is a timeseries dataset therefore a 3D tensor shape is required for the project.

After the architecture is defined there are still choices to be made with regard to

the loss function and the optimiser. As this is fundamentally a regression the chosen loss function used can be mean absolute error (mae) or mean squared error (mse). Mae being the absolute value of the difference between the predictions and the targets and mse being the square of the difference. This is a widely used loss function for regression problems. The default in the keras documentation is mse and this was used at initiation.

The dataset will be data for one stock ‘A’ for the time period 2009-13-31 to 2019-01-01 a matrix (36, 72) in shape. Train data will be the first 32 instances of the data set. Initially a very small network was used consisting of one hidden layer with 32 units. The design of neural networks is often described as more of an art than a science, a process of trial and error but also adopting know best practices was the approach taken in the design stage. Small networks are often seen as a way of mitigating overfitting. The network ends with a single unit and no activation, a linear layer. This will single layer will be maintained for all architectures used, as this is a regression problem and we are trying to predict a single continuous value. Applying an activation function would constrain the range the output could take. The optimiser is ‘rmsprop’, is the default optimiser in the keras documentation. The implementation also uses the mean absolute error as the metric to be monitored by the model. To validate the network K-fold cross validation is used splitting the data into K partitions, instantiating the K identical models and training each one on K-1 partitions while evaluation is based on the remaining partition. Taking the average of the scores returns a single measurement. The epochs, the length of time the model is trained for, is set to 100 in and attempt to restrain processor requirements. Parameters that will be adjusted are the number and size of the hidden layers. After the parameters have been amended a model is trained on all the training data and its performance evaluated on the test data. A simple model will be used as a baseline for comparison with more complicated models.

The next step in the implementation process was to move from just a densely connected neural networks to recurring neural networks, adding memory to the network. RNNs process sequences by iterating through the sequence elements and maintaining a state containing information relative to what it has seen so far. Keras has a SimpleRNN layer which processes batches of sequences and takes the inputs of shape (batch\_size, timesteps, input\_features), rather than just (timesteps, input\_features.)

A SimpleRNN layer was used in the next testing with changes to the numbers of layers and the units the parameters that could be amended. As is captured in the literature simple RNNs are generally too simplistic to be of real use. SimpleRNN has a major issue: although it should theoretically be able to retain at time*t* information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn. This is due to the vanishing gradient problem

 LSTM and GRU layers are designed to solve this problem, and these are applied next. A single layer GRU is added with a single dense output layer, keras has an implementation for this, again the parameters are the number of layers and the units. Next a single layer LSTM is applied in the same way.

**4 TESTING AND RESULTS**

Evaluation of all models will be 4-fold cross validation with 100 epochs with mean absolute error (mae) as the validation metric, the mean for these folds is calculated. The mae is then calculated on the entire test data set providing a final test mae score.

To determine a baseline for testing, a simple small densely connected network was created.

|  |  |
| --- | --- |
| **Model Layer** | Dense |
| **Number of Layers** | 1 |
| **Number of Units** | 32 |
| **Optimiser** | rmsprop |
| **Loss** | mse |
| **Metrics** | mae |
| **Validation** |  |
| **Fold** |  |
| **1** | 0.343737632 |
| **2** | 0.134157419 |
| **3** | 0.175569564 |
| **4** | 0.177201182 |
| **Validation Mean** | 0.207666449 |
| **Test Score** | **0.108901992** |

The results for the Simple RNN

|  |  |
| --- | --- |
| **Model Layer** | Simple RNN |
| **Number of Layers** | 1 |
| **Number of Units** | 32 |
| **Optimiser** | rmsprop |
| **Loss** | mse |
| **Metrics** | mae |
| **Validation** |  |
| **Fold** |  |
| **1** | 0.287335515 |
| **2** | 0.236223578 |
| **3** | 0.156718925 |
| **4** | 0.206536397 |
| **Validation Mean** | 0.221703604 |
| **Test Score** | **0.302181363** |

The Simple RNN results are a large degradation on the baseline.

GRU results improve on the baseline slightly.

|  |  |
| --- | --- |
| **Model Layer** | GRU |
| **Number of Layers** | 1 |
| **Number of Units** | 32 |
| **Optimiser** | rmsprop |
| **Loss** | mse |
| **Metrics** | mae |
| **Validation** |  |
| **Fold** |  |
| **1** | 0.284213245 |
| **2** | 0.147503138 |
| **3** | 0.100005403 |
| **4** | 0.151451439 |
| **Validation Mean** | 0.170793306 |
| **Test Score** | **0.102192424** |

LSTM score is a further improvement

|  |  |
| --- | --- |
| **Model Layer** | LSTM |
| **Number of Layers** | 1 |
| **Number of Units** | 32 |
| **Optimiser** | rmsprop |
| **Loss** | mse |
| **Metrics** | mae |
| **Validation** |  |
| **Fold** |  |
| **1** | 0.282652259 |
| **2** | 0.131771699 |
| **3** | 0.098661974 |
| **4** | 0.125670373 |
| **Validation Mean** | 0.159689076 |
| **Test Score** | **0.096809559** |

Plotting this we can see that the baseline simple model is difficult to beat although some gains can be made with models that have the concept of ‘memory.’

Parameters that are configurable are number of layers and number of units. Changing the layers to two (excluding the output layer) had the following effects on the test scores, retaining all other model metrics.

|  |  |
| --- | --- |
| Model | MAE Score |
| Baseline | 0.10890199 |
| Simple RNN (1) | 0.30218136 |
| Simple RNN (2) | 0.1022668 |
| GRU (1) | 0.10219242 |
| GRU (2) | 0.13110155 |
| LSTM (1) | 0.09680956 |
| LSTM (2) | 0.10668621 |

Adding another layer only improved the Simple RNN model and although this was an improvement on the baseline score it did not better the one-layer GRU or LSTM models. Adding layers did not improve accuracy so the single layer was retained when adjusting the number of units.

Maintaining the single layer on the GRU and LSTM model the number of units can be adjusted. The results of amending the layers are shown below.

Units of 32 do appear to be the optimal number for both GRU and LSTM layers with LSTM(32) fairing the best of all with a MAE

With the major parameters reviewed testing now looked to amending the loss function. The other loss functions available for regression problems are mean squared log error (msle) and mean absolute error (mae). Msle has the effect of relaxing the punishing effect of large differences in large predicted values. When predicting a large value, you may not want to punish a model as heavily as mean squared error. Mae is a loss function that is generally more robust to outliers, large or small values far from the mean value.  When using these for GRU and LSTM single layer models with 32 units the results for the MAE were as below.

A change to the loss functions did not improve the score from the mse

**5 ANALYSIS AND DISCUSSION**