NEURAL NETS STRATEGY

Authors

Nicholas Wan | Tan Yi Hao | Yeo Zhang Yi

NUS FINTECH SOCIETY

nusfintech@gmail.com

Contents

Model Instructions

1. Initialization: Variable Declaration

To use the model, simply set the start and end date. Set the variables self.currency, which is the main currency to be traded, and self.currency2 and self.currency3 which are to be used as input features. Self.timeframe is the number of data points to be used in the training set. Self.features is the number of features to be used, in this case the 4 technical indicators and the 2 currency pairs.

```
def Initialize(self):
    self.SetStartDate(2018,11,7)    #Set Start Date
    self.SetEndDate(2018,11,21)    #Set End Date
    self.SetCash(100000)           #Set Strategy Cash
    self.SetBrokerageModel(BrokerageName.InteractiveBrokersBrokerage, AccountType.Cash)
    self.currency = "EURUSD"
    self.currency2 = "AUDUSD"
    self.currency3 = "GBPUSD"

    self.resolution = Resolution.Minute
    self.AddForex(self.currency,self.resolution)
    self.AddForex(self.currency2,self.resolution)
    self.AddForex(self.currency3,self.resolution)
    self.AddForex("EURUSD 8G",self.resolution)

    self.long_list = []
    self.short_list = []
    self.model =Sequential()
```

(Figure 1. Setting initialization parameters)

2. Varying Hyperparameters

Here, a grid search for the number of cells and epochs is conducted. For each parameter of cell and epoch value used, we calculate the mean squared error, tabulate it and compare across different parameter. These grids can be varied, or random search could be used instead.

```
        if self.start == 0:

            #USE TimeSeriesSplit to split data into n sequential splits
            tscv = TimeSeriesSplit(n_splits=2)

            # Make cells and epochs to be used in grid search.
            cells = [100,200]
            epochs  = [100,200]

            # creating a datframe to store final results of cross validation for different combination of cells and epochs
            df = pd.DataFrame(columns= ['cells','epoch','mse'])
```

(Figure 2. Tuning Hyperparameters)

3. Varying trade decisions

Based on our model predicted output, we make the decision to long and close long positions. If the output is positive, the price is expected to rise so we will long the currency or close our short position. Whereas if the output is negative, the price is expected to fall so we will short the currency or close our long position. We can vary the parameters to adjust the amount of trade to the predicted change in price, thus reducing our risk.

```
        ####Make decision for trading based on the output from LSTM and the current price.
        #Long when output exceeds a positive limit
        #Close long position when output becomes negative
        #Short when output falls below a negative limit
        #Close short position when output becomes positive

        upper_limit = 0.0000025
        lower_limit = upper_limit*-1
        upper_bound = 0.0025
        lower_bound = upper_bound*-1
```

(Figure 3. Trading Decisions)

Data Munging

1. Feature selection

Initially, we chose 7 features as inputs: 5, 4, 3 day Smooth moving averages (SMA), Exponential moving average (EMA), Moving average convergence divergence (MACD), Relative strength index (RSI) and Momentum indicator (MOM).

We choose these technical indicators as such because they represent different aspects of the price movements, such as trend, momentum, or volatility (Tanaka-Yamawaki & Tokuoka, 2007). We selected SMA and EMA to represent the trend, RSI for volatility, and MOM for momentum. We also input historical prices as well in the form of 5-day, 4-day, 3-day SMA. We also use 5-days EMA as recent data are more important and EMA gives more weight to recent data.

2. Finding correlation

Next, we tested for multicollinearity in the features. Research has shown that adding collinear features can cause a lower predictive accuracy (Howley, Madden, O'Connell, & Ryder, 2006).

	SMA5	SMA4	SMA3	EMA	MACD	RSI	MOM
	0	1	2	3	4	5	6
0	1.000000	0.995499	0.982370	0.995030	0.816139	-0.100890	0.070277
1	0.995499	1.000000	0.994278	0.995402	0.834975	-0.036842	0.149202
2	0.982370	0.994278	1.000000	0.990221	0.844753	0.039575	0.226541
3	0.995030	0.995402	0.990221	1.000000	0.809914	-0.037606	0.116214
4	0.816139	0.834975	0.844753	0.809914	1.000000	0.108403	0.316831
5	-0.100890	-0.036842	0.039575	-0.037606	0.108403	1.000000	0.823749
6	0.070277	0.149202	0.226541	0.116214	0.316831	0.823749	1.000000

(Figure 4. Correlation Matrix 1)

We can see that certain features have an extremely high correlation (features 0 - SMA5, 1- SMA4, 2- SMA3, 3- EMA) have a correlation of >0.98. Hence, It is sufficient just to have one of these features, as they do not provide more information to the model but only increase computational cost. Hence, we would look into removing the features with extremely high correlation, and add features that do not have that high a correlation (<0.9)

3. More feature selection

We explored using other technical indicators that can also be used to measure price trend, volatility, volume, and momentums to compare their effectiveness with existing technical indicators. We used one of each and included in BB and ROC.

Types of Technical Indicators	
Type	**Examples**
Trend	Moving Averages, MACD, Parabolic SAR
Momentum	Stochastics, CCI, Relative Strength Index
Volatility	Bollinger Bands, Average True Range, Standard Deviation
Volume	Chaikin oscillator, OBV, Rate of Change (ROCV)

(Figure 5. Types of Technical Indicators)

For the second multicollinearity test, we found that features 0 (EMA) has a high correlation with 4 (BB), feature 3 (MOM) has a high correlation with feature 5 (ROC). We can remove 2 redundant features, 0 (EMA) and 3 (MOM).

	EMA	MACD	RSI	MOM	BB	ROC
	0	1	2	3	4	5
0	1.000000	0.765907	0.124391	0.099519	0.991402	0.100716
1	0.765907	1.000000	0.170037	0.257934	0.780680	0.258447
2	0.124391	0.170037	1.000000	0.781341	0.231601	0.781151
3	0.099519	0.257934	0.781341	1.000000	0.206606	0.999993
4	0.991402	0.780680	0.231601	0.206606	1.000000	0.207768
5	0.100716	0.258447	0.781151	0.999993	0.207768	1.000000

(Figure 6. Correlation Matrix 2)

For the third multicollinearity test, features EMA and MOM were removed. There appears to be no other features that are strongly correlated (>0.9).

	MACD	RSI	BB	ROC
	0	1	2	3
0	1.000000	0.170037	0.780680	0.258447
1	0.170037	1.000000	0.231601	0.781151
2	0.780680	0.231601	1.000000	0.207768
3	0.258447	0.781151	0.207768	1.000000

(Figure 7. Correlation Matrix 3)

Other currency pairs were used as additional features because like "EURUSD", these currency pairs move against the USD. When the USD becomes strong, these currency pairs tend to move in the same direction and corresponds to the price movement, and likewise if the USD becomes weak. Research shows that AUD and GBP have a pairwise relation with EUR (Bekiros, & Diks, 2008). Thus, the respective currency pairs may have some relationship and will help predict the price movement of EURUSD. A test for multicollinearity shows that none of the feature inputs is extremely correlated.

4. Data scaling

Next, the training set is scaled using the indicator that we have defined. This same scaler will later be used to scale the input data for predictions.

```
#Scale and normalise x training data
#self.Debug("Length of training data: "+str(len(self.indicatorData)))
#self.Debug("X_data is " + str(self.indicatorData))
self.IndicatorScaler.fit(self.indicatorData)
X_data = self.IndicatorScaler.transform(self.indicatorData)
#self.Debug("______________________________________________________________")
#self.Debug("X_data after transform: "+str(X_data))
```

(Figure 8. Using MinMaxScaler() for data scaling and transformation)

Performance Measurements

1. Testing accuracy of the model

We apply time series windowing on data to train our models for different time frames. This is so to ensure our model is trained and able to predictions in time periods of different trends and seasonality. The function time series split is used to split the data into different time frames. Here, we set n_split = 2.

```
        if self.start == 0:

            #USE TimeSeriesSplit to split data into n sequential splits
            tscv = TimeSeriesSplit(n_splits=2)

            # Make cells and epochs to be used in grid search.
            cells = [100,200]
            epochs  = [100,200]

            # creating a datframe to store final results of cross validation for different combination of cells and epochs
            df = pd.DataFrame(columns= ['cells','epoch','mse'])
```

(Figure 9. Using time split series)

2. Retraining model

We also vary the parameters of cells and epochs from 100 to 200. For each parameter of cell and epoch value used, we calculate the mean squared error, tabulate it and compare across different parameter.

```
            #Loop for every combination of cells and epochs. In this setup, 4 combinations of cells and epochs [100, 100] [ 100,200] [200,100] [200,200]
            for i in cells:
                for j in epochs:
                    cvscores = []
                    # to store CV results
                    #Run the LSTM in loop for every combination of cells an epochs and every train/test split in order to get average mse for each combination.
                    for train_index, test_index in tscv.split(X_data):
                        #self.Debug("TRAIN:", train_index, "TEST:", test_index)
                        X_train, X_test = X_data[train_index], X_data[test_index]
                        Y_train, Y_test = Y_data[train_index], Y_data[test_index]
                        #self.Debug ( " X train [0] is " + str (X_train[0]))
                        #self.Debug ( " X train [1] is " + str (X_train[1]))

                        X_train= np.reshape(X_train, (X_train.shape[0],1,X_train.shape[1]))
                        #self.Debug("X input to LSTM :  " + str(X_train))
                        X_test= np.reshape(X_test, (X_test.shape[0],1,X_test.shape[1]))
                        #self.Debug("Y input to LSTM :  "+ str(Y_train)

                        #self.Debug("START: LSTM Model")
                        #self.Debug(i)
                        #self.Debug(j)
                        model = Sequential()
                        model.add(LSTM(i, input_shape = (1,7), return_sequences = True))
                        model.add(Dropout(0.10))
                        model.add(LSTM(i,return_sequences = True))
                        model.add(LSTM(i))
                        model.add(Dropout(0.10))
                        model.add(Dense(1))
                        model.compile(loss= 'mean_squared_error',optimizer = 'rmsprop', metrics = ['mean_squared_error'])
                        model.fit(X_train,Y_train,epochs=j,verbose=0)
                        #self.Debug("END: LSTM Model")
```

```
                scores = model.evaluate(X_test, Y_test, verbose=0)
                #self.Debug("%s: %f " % (model.metrics_names[1], scores[1]))
                cvscores.append(scores[1])

            MSE= np.mean(cvscores)
            #self.Debug("MSE" + str(MSE))

            #Create a dataframe to store output from each combination and append to final results dataframe df.
            df1 = pd.DataFrame({ 'cells': [i], 'epoch': [j], 'mse': [MSE]})
            #self.Debug("Individual run ouput DF1" + str(df1))
            #Appending individual ouputs to final dataframe for comparison
            df = df.append(df1)

        #self.Debug("Final table of DF"+ str(df))

        #Check the optimised values obtained from cross validation
        #This code gives the row which has minimum mse and store the values to O_values
        O_values = df[df['mse']==df['mse'].min()]

        # Extract the optimised  values of cells and epochs from above row (having min mse )
        O_cells = O_values.iloc[0][0]
        O_epochs = O_values.iloc[0][1]

        #self.Debug( "O_cells"  + str (O_cells))
        #self.Debug( "O_epochs" + str (O_epochs))
```

(Figure 10. Varying cells and epochs and mapping mean squared error)

Then the cells and epochs that minimize mean squared error is taken as the optimized parameter and is used in the final model. Self.start is set to 1 so that the model is only trained once.

```
            #self.Debug("Final table of DF"+ str(df))

            #Check the optimised values obtained from cross validation
            #This code gives the row which has minimum mse and store the values to O_values
            O_values = df[df['mse']==df['mse'].min()]

            # Extract the optimised  values of cells and epochs from above row (having min mse )
            O_cells = O_values.iloc[0][0]
            O_epochs = O_values.iloc[0][1]

            #self.Debug( "O_cells"  + str (O_cells))
            #self.Debug( "O_epochs" + str (O_epochs))

            X_data1= np.reshape(X_data, (X_data.shape[0],1,X_data.shape[1]))
            #self.Debug("START: Final_LSTM Model")

            self.model.add(LSTM(O_cells, input_shape = (1,7), return_sequences = True))
            self.model.add(Dropout(0.10))
            self.model.add(LSTM(O_cells,return_sequences = True))
            self.model.add(LSTM(O_cells))
            self.model.add(Dropout(0.10))
            self.model.add(Dense(1))
            self.model.compile(loss= 'mean_squared_error',optimizer = 'rmsprop', metrics = ['mean_squared_error'])
            self.model.fit(X_data1,Y_data,epochs=O_epochs,verbose=0)
            #self.Debug("END: Final_LSTM Model")

        self.start = 1
```

(Figure 11. Recreating LTSM model with optimized cells and epochs)

3. Avoid overfitting

To ensure that our model minimizes overfitting, we have applied dropout to our model. Dropout is a regularization method where input and recurrent connections to LSTM units are probabilistically excluded from activation and weight updates while training a network. This has the effect of reducing overfitting and improving model performance.

Risk Management

1. Making trade decisions

Based on our model predicted output, we make the decision to long and close long positions. If the output is positive, the price is expected to rise so we will long the currency or close our short position. Whereas if the output is negative, the price is expected to fall so we will short the currency or close our long position.

```
#Close long position if output becomes negative and holding long position
#Long when output exceeds limit and not holding any position
if output>0 and (self.currency not in self.long_list):
    self.SetHoldings(self.currency, 1)
    self.long_list.append(self.currency)
    self.Debug("Make a long position")

if self.currency in self.long_list and output < 0:
    self.SetHoldings(self.currency, 0)
    self.long_list.remove(self.currency)
    self.Debug("Close long position")

#self.Debug("END: Ondata")
```

(Figure 12. Trading Decisions)

2. Risk management strategies

Given that there are transaction fees involved, we decide to set benchmark to avoid making trades on extremely small price differences. As such we set a limit value of 0.0005 and only make long and short trades if output exceeds the limit. We only make trades when we obtain outputs of large enough value that predicts that trade is profitable enough to cover transaction fees.

```python
#Make decision for trading based on the output from LSTM and the current price.
#Long when output exceeds a positive limit
#Close long position when output becomes negative
#Short when output falls below a negative limit
#Close short position when output becomes positive

limit = 0.0005

#Long when output exceeds limit and not holding any position
if output > limit and self.currency not in self.long_list and self.currency not in self.short_list:
    self.SetHoldings(self.currency, 1)
    self.long_list.append(self.currency)
    self.Debug("Make a long position")

#Short when output falls below limit and not holding any position
if output < limit*-1 and self.currency not in self.long_list and self.currency not in self.short_list:
    self.SetHoldings(self.currency, -1)
    self.short_list.append(self.currency)
    self.Debug("Make a short position")

#Close long position if output becomes negative and holding long position
if self.currency in self.long_list and output < 0:
    self.Liquidate(self.currency)
    #self.SetHoldings(self.currency, 0)
    self.long_list.remove(self.currency)
    self.Debug("Close long position")

if self.currency in self.short_list and output < 0:
    self.Liquidate(self.currency)
    self.short_list.remove(self.currency)
    self.Debug("Close short position")
#self.Debug("END: Ondata")
```

(Figure 13. Improving trading decisions)

The profitability of our model can be further improved if we implement in 'risk and reward' rules. Hence, we size our positions according to the amount of risk and returns we would expect. In our model, we will make our trades proportional to output values. A greater output value would indicate a higher upward price movement, hence a higher profitability so we will increase our stakes in the long position.

```python
#Make decision for trading based on the output from LSTM and the current price.
#Long when output exceeds a positive limit
#Close long position when output becomes negative
#Short when output falls below a negative limit
#Close short position when output becomes positive


upper_limit = 0.0001
lower_limit = upper_limit*-1
upper_bound = 0.001
lower_bound = upper_bound*-1
size = 0

#Model predicts that price will rise by a lot. Full long
if output > upper_bound:
    size = 1

#Model predicts that price will rise, but not by a lot. Partial long
elif output>upper_limit and output <upper_bound:
    size = output/upper_bound * 100

#Model predicts price will drop by a lot. Full short
elif output < lower_bound:
    size = 1

#Model predicts price will drop, but not by a lot. Partial short
elif output < lower_limit and output > lower_bound:
    #Go partial short
    size = output/lower_bound * 100
```

(Figure 14. Resizing position algorithm)