# Appendix B

## B.1 Building RNN Architectures in Keras framework

The Keras framework's Sequential() API is well-known for rapid prototyping and its easy of use. Neural network-based models are built by stacking layers on top of each other from input to output. A simple 3-layer(1 hidden) neural network may be built as follows:

```
from tensorflow.keras.layers import Sequential, Dense, InputLayer

model = Sequential()                                  # Initialize Model
model.add(InputLayer(input_shape=<iN>))               # Define Input Layer
model.add(Dense(units=<hN>, activation='relu'))       # Define Hidden Layer
model.add(Dense(units=<oN>, activation='linear'))     # Define Output Layer
```

Listing B.1: 3-Layer Neural Network

For a neural network, the arguments for the `Dense` layer are as given in the above example code. The `<iN>` represents the number of input neurons which should be equal to the number of features in the input matrix($X$). The input matrix $X$ should be of the dimensions $m$ by $n$, where $m$ = number of samples in the data and $n$ = number of features. The `<hN>` represents the number of neurons in the hidden layer and `<oN>` represents the number of neurons in the output layer. The activation function for each `Dense` layer is passed as the $2^{nd}$ argument to the function.

To build recurrent neural networks we use similar code structure by replacing the initial `Dense` layers with either `GRU` layer or `LSTM` layer.

```
from tensorflow.keras.layers import Sequential, Dense, InputLayer, GRU

model = Sequential()
model.add(InputLayer(input_shape=(timsteps,features)))
model.add(GRU(units=<hS>, activation='relu', return_sequences=False))
model.add(Dense(units=<oN>, activation='linear'))
```

Listing B.2: Simple GRU Network

Here, the key change is now in the `input_shape` argument for the `InputLayer`. The `input_shape` argument is now a tuple of `(timesteps, features)` where the `timesteps` represents number of lags and the `features` represents the number of features at *each* of those `timesteps`. The `return_sequences` argument is set to `False` as it is used to stack multiple GRU or LSTM layers on top of each other and for this vanilla GRU network following layer is the final `Dense`(output) layer. The `<hS>` is the dimensions of the hidden state vector in a GRU or LSTM cell. This is **not** equivalent to the number of hidden 'cells' of a RNN because the number of temporal cells that a RNN will spawn will always be equal to the number of `timesteps` used in modeling.

After converting the time-series into a supervised learning problem, the input matrix is 2-dimensional where the number of columns indicate the number of lags being used for that given model. Thus, that is equal to the `timesteps` mentioned above. However, for each of these timesteps we have only 1 single scalar value for the given quantity, when we only have a single time-series. This indicates that the number of features for *each* of those `timesteps` for uni-variate time-series is equal to 1. Thus, for uni-variate time-series modeling, the `input_shape` will be equal to `(lags, 1)`. This is key to running any RNN model in Keras framework. Now, because the each input sample is of dimension `(timesteps, features)` and we $m$ such samples in our training data, the input matrix($X$) needs to reshaped into a 3-D matrix of dimensions `(`$m$`, lags, 1)` for uni-variate time-series. Note that, upon conversion of time-series to supervised learning problem, we already have the input matrix($X$) in the dimensions `(`$m$`, lags)`. A simple call to the numpy library's `reshape` function and passing in the arguments `(`$m$`, lags, 1)` would convert the 2-D matrix to the required 3-D matrix for uni-variate time-series modeling.

### B.1.1 Stacked Architectures

To build a multi-layer RNN with stacked GRU or LSTM layers, we need to define each layer after the other in the Sequential() API. In order to ensure that the layers are stacked correctly and the follow of information is accurate, we define `return_sequences` equal to `True` as shown in the below example.

```
from tensorflow.keras.layers import Sequential, Dense, InputLayer, GRU

model = Sequential()
model.add(InputLayer(input_shape=(timsteps,features)))
model.add(GRU(units=<hS1>, activation='relu', return_sequences=True))
model.add(GRU(units=<hS2>, activation='relu', return_sequences=True))
model.add(GRU(units=<hS3>, activation='relu', return_sequences=False))
model.add(Dense(units=<oN>, activation='linear'))
```

Listing B.3: Stacked GRU Network

The above network now has 3 `GRU` layers stacked on top of each other and a final `Dense` layer to give the required multihorizon output. This can potentially allow the different GRU layer to model separate pieces of information and can collective add up to give more modeling powers compared to the Simple GRU Network. Stacked networks can especially be beneficial when Simple GRU or LSTM Networks may not be enough to model the information in the given data.

### B.1.2 Autoencoder Architectures

Instead of passing the outputs from one stacked RNN layer to another, we can choose to set `return_sequences` equal to `False` while connecting the intermediate layers using fixed length vector sequences. To do this we define a `RepeatVector` layer in-between the stacked RNN layers as shown in example below.

```
from tensorflow.keras.layers import Sequential, Dense, InputLayer, GRU,
    RepeatVector

model = Sequential()
model.add(InputLayer(input_shape=(timsteps,features)))
model.add(GRU(units=<hS1>, activation='relu', return_sequences=False))
model.add(RepeatVector(<nRepeats>))
model.add(GRU(units=<hS2>, activation='relu', return_sequences=False))
model.add(Dense(units=<oN>, activation='linear'))
```

Listing B.4: GRU Autoencoder

The above network now is called the "GRU-based Autoencoder". The idea behind this is that the output of the first `GRU` is encoded in a fixed length sequence of size *<nRepeats>*, which is then passed as input to the next `GRU` layer which may act as decoder to model the information encoded by the first layer so as to offer better modeling capabilities.

### B.1.3 Multi-Level Autoencoders

It is also possible to define multi-level autoencoders which may have multiple GRU or LSTM layers each of which will have encoded fixed length vector between them as shown in the example below.

```
from tensorflow.keras.layers import Sequential, Dense, InputLayer, GRU,
    RepeatVector

model = Sequential()
model.add(InputLayer(input_shape=(timsteps,features)))
model.add(GRU(units=<hS1>, activation='relu', return_sequences=False))
model.add(RepeatVector(<nRepeats>))
model.add(GRU(units=<hS2>, activation='relu', return_sequences=False))
model.add(RepeatVector(<nRepeats>))
model.add(GRU(units=<hS3>, activation='relu', return_sequences=False))
model.add(RepeatVector(<nRepeats>))
model.add(GRU(units=<hS4>, activation='relu', return_sequences=False))
model.add(Dense(units=<oN>, activation='linear'))
```

Listing B.5: Multi-Level GRU Autoencoder

This can potentially offer more modeling powers to the autoencoders. The size of each repeat vector does not have to be the same, it is shown as an example in the above code snippet.

Note: All the above code snippet examples use `GRU` layer but the same can be replaced with `LSTM` layer without requiring any additional changes to accommodate for the LSTM networks.

## B.2 Rolling Validation in Python

The below method explains the logic involved in computing the multi-horizon rolling validation forecasts on the test data.

```
1  from tensorflow.keras import callbacks
2  def rollingValidate(model,x,y,kt,nEpoch,nBatch,plot=True):
3      TR_RATIO = 0.6
4      m = y.shape[0]
5      tr_size = int(m * TR_RATIO)
6      te_size = int(m - tr_size)
7      te = int(tr_size)
8      if kt < 0: kt = te_size
9      (x_e, y_e, x_r, y_r) = chopr (x, y, te, te_size, tr_size)
10     forecasts = []
11     for i in range(y_e.shape[0]):
12         if i%kt == 0:
13             if i > 0:
14                 xe_next = x_e.iloc[i-kt:i]
15                 ye_next = y_e.iloc[i-kt:i]
16                 x_r, y_r = shift((x_r,y_r),((xe_next, ye_next)))
17             model = buildModel(<modelName>) #re-define model
18             callb = callbacks.EarlyStopping(monitor='loss',
   restore_best_weights=True, patience=20,verbose=1)
19             fit = model.fit(x_r, y_r,validation_split=0.0,shuffle=True,
   epochs=nEpoch,batch_size=nBatch,verbose=0, callbacks=callb)
20             if plot: plotHistory(fit)
21         forecasts.append(model.predict(x_e.iloc[i,:].values.reshape(1,x_e.
   shape[1])).squeeze())
22     rSq   = [x for x in range(y.shape[1])]
23     mse   = [x for x in range(y.shape[1])]
24     sse   = [x for x in range(y.shape[1])]
25     sst   = [x for x in range(y.shape[1])]
26     rmse  = [x for x in range(y.shape[1])]
27     mape  = [x for x in range(y.shape[1])]
28     smape = [x for x in range(y.shape[1])]
29     forecasts = pd.DataFrame(np.array(forecasts).reshape(y_e.shape))
30     for h in range(1,y.shape[1]+1):
31         yf = forecasts.iloc[:,h-1].values
32         rSq[h-1], mape[h-1], smape[h-1], mse[h-1], rmse[h-1], sse[h-1],
   sst[h-1] = eval(y_e.values[:,h-1],yf)
33     return (rSq, mape, smape, mse, rmse, sse, sst, forecasts, y_e)
```

Listing B.6: Rolling Validation

Lines 3-8 define the configuration parameters for the rolling validation. `TR_RATIO` defines what portion of the input time-series data `x` and `y` will be used as the training data. The argument `kt` decides the frequency of retraining while generating rolling forecasts into the test set.

Once the configuration parameters for the method are set, the `chopr` method, as given in Listing B.7, is used to split the data into training(`x_r`, `y_r`) and test(`x_e`, `y_e`) data. Line 10 defines an empty array to collect the rolling out-of-sample forecasts.

```
1 def chopr(x,y,te,te_size,tr_size):
2     te2 = te + te_size
3     tr = te - tr_size
4     x_e = x[te:te2]
5     y_e = y[te:te2]
6     x_r = x[tr:te]
7     y_r = y[tr:te]
8     return(x_e, y_e, x_r, y_r)
```

Listing B.7: chopr method

We then start iterating through the length of the test set in line 11 and generate the forecasts 1 sample at a time. Line 12 checks if the current iteration has reached the frequency of retraining, in which case model retraining needs to occur else the program jumps to Line 21 for forecasting 1 sample from the test data and appending it to the forecasts array for accumulating the rolling forecasts.

Upon satisfying the retraining condition on line 12, a few additional things need to happen before we roll forward to forecast the next values in the test data. If it is not the first iteration of the test data, then we do not need to shift the test set into the training set so we jump to line 17 to build the model using the `buildModel` method. The `buildModel` method may be a generic model building method that accepts the `<modelName>` as an arugment to generate and return the model object based on the name of the architecture. Line 18 defines the Early Stopping rule for models in Keras. We then fit the model, on line 19, using the training(`x_r`, `y_r`) data for number of epochs and batch size, given by `nEpoch` and `nBatch` arguments, respectively. We ensure that validation does not occur during training using `validation_split=0.0` arugment for the fit() method.

If the condition on line 12 is satisfied and we are not in the first iteration of the rolling forecasts, as given by the condition on line 13, then this means that we are predicted `kt` number of samples into the test data and we now need to shift the next `kt` number of samples from the test data to the training data and remove the earliest `kt` samples from the training data. This is handled by the `shift` method call on line 16. Line 14 and 15 collect the next `kt` samples from the test(`x_e`, `y_e`) data. The `shift` method is given in listing B.8.

```
1 def shift_rm(xy1,xy2):
2     d1 = xy1[1].shape[0]
3     d2 = xy2[1].shape[0]
4     gap = d1 - d2
5     x = pd.DataFrame(np.random.randint(10, size=(d1, xy1[0].shape[1])))
6     y = pd.DataFrame(np.random.randint(10, size=(d1, xy1[1].shape[1])))
7     for i in range(y.shape[0]):
8         if i<gap:
9             for j in range(x.shape[1]): x.iloc[i,j] = xy1[0].iloc[i+d2,j].
    copy()
10            y.iloc[i,:] = xy1[1].iloc[i+d2].copy()
```

```
        else:
            for j in range(x.shape[1]): x.iloc[i,j] = xy2[0].iloc[i-gap,j
    ].copy()
            y.iloc[i,:] = xy2[1].iloc[i-gap].copy()
     x.columns = xy1[0].columns
     y.columns = xy1[1].columns
     return(x,y)
```

Listing B.8: shift method

Once the model is trained if the `plot` argument is true, we use the `plotHistory` method, as given in Listing B.9, to plot the loss vs epochs curve and give the minimum training loss and the epoch number at which occurs.

```
def plotHistory(fit_history,ls="MAPE"):
    loss = fit_history.history['loss']
    epoch_count = range(1, len(loss) + 1)
    plt.figure(num=0,figsize=(6,3))
    plt.plot(epoch_count, loss)
    plt.legend(['Training Loss - '+str(ls), 'Validation Loss - '+str(ls)])
    plt.xlabel('Epochs')
    plt.ylabel(ls)
    plt.title("Loss vs Epochs")
    plt.show()
    print("Min Training Loss = " + str(np.min(loss)) + " ",end='')
    print("at Epoch " + str(np.argmin(loss) + 1))
```

Listing B.9: plotHistory method

This repeats until we iterate through the test data and finally, when all the forecasts are collected we proceed to generate the performance metrics for each horizon. To enable collection of performance metrics, line 22-28 defines their respective arrays to store the values. Line 29 converts the collected forecasts into a matrix of dimensions `te_size` $\times$horizons.

We then iterate through the number horizons($h$) on Line 30, to collect the metrics for each horizon. Line 31 extracts the forecast vector($y_f$) for each horizon from the `forecasts` matrix. Line 32 calls the `eval` method, as given in Listing B.10, to generate and return all the required performance metrics and other statistics which are collected in their respective arrays.

```
def eval(y,yp):
    import math
    m = y.shape[0]
    e = y - yp
    yt = y - y.mean()
    sse = e.dot(e)
    sst = yt.dot(yt)
    rSq = round((1 - ((sse)/(sst))),roundTo)
    mape = round(((np.absolute(e)/np.absolute(y)).sum())*100/m,roundTo)
    smape = round(((np.absolute(e)/(np.absolute(y)+np.absolute(yp))).sum()
   )*200/m,roundTo)
```

```
11      mse = round((sse/m),roundTo)
12      rmse = round(math.sqrt(mse),roundTo)
13      return(format(rSq,'.4f'),format(mape,'.2f'),format(smape,'.2f'),format
       (mse,'.2f'),format(rmse,'.2f'),format(sse,'.2f'),format(sst,'.2f'))
```

Listing B.10: eval method

Once we collect the metrics for all horizons the method returns the metrics along with the out-of-sample forecasts and the true values on Line 33.

This template for rolling validation is particularly suited for Keras-based models. However, with slight modifications the same template/logic should be usable on models generated through the other machine learning or statistics frameworks.