

Recurrent Neural Networks

Anahita Zarei, Ph.D.



Overview

- Recurrent Networks
 - LSTM
 - GRU
- 1D Convnets

Motivation for RNN's

- We previously saw that Convolutional Neural Networks (CNNs), form the base of many state-of-the-art **computer vision** systems. However, we do not understand the world around us with vision alone.
- **Sound**, for one, also plays an important role. As humans, we communicate and express ideas through sequences of symbolic reductions and abstract representations.
- Naturally we would want machines to understand this manner of **processing sequential information**, as it could help us to resolve many problems we face with such sequential tasks in the real world.

Examples of Sequences

- Visiting a foreign country and need to **order in a restaurant**.
- Want your car to perform a **sequence of movements** automatically so that it is able to park by itself.
- Want to understand how different sequences of adenine, guanine, thymine, and cytosine molecules in the **human genome** lead to differences in biological processes occurring in the human body.
- What's common between all these examples?
- All are related to sequence modeling tasks. In such tasks, the training examples (vectors of words, a set of car movements, or configuration of *A*, *G*, *T*, and *C* molecules) are multiple **time-dependent data points**.

Examples of Sequences

- *Don't judge a book by its ____.*
- How do you know that the next word?
- You consider the relative positions of words and (subconsciously) perform some form of Bayesian inference, leveraging the sentences you have previously seen and their apparent similarity to this example.
- i.e., you used your internal model of the English language to predict the most probable word to follow.
- Language model refers to the **probability of a particular configuration** of words occurring together in a given sequence.
- Such models are the **fundamental components** of modern speech recognition and machine translation systems.
- They rely on **modeling the likelihood of sequences of words**.

Why RNN's?

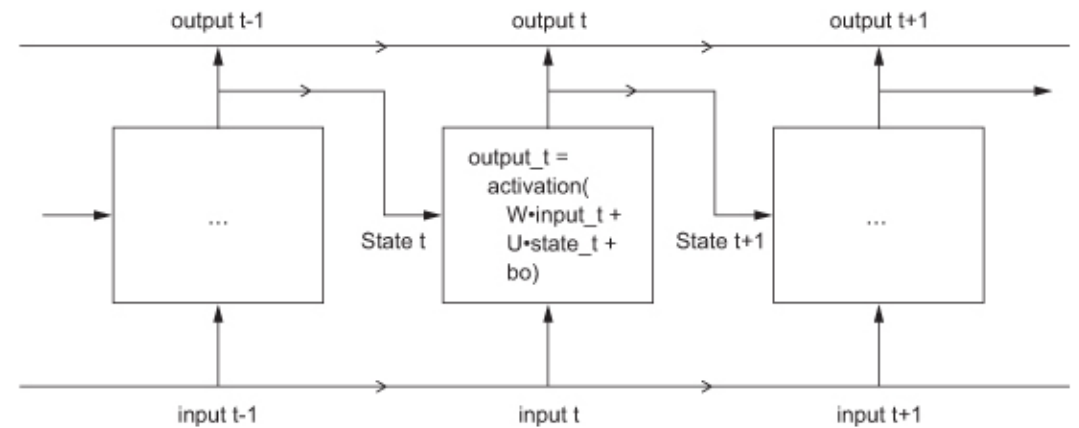
- A major characteristic of densely connected networks and convnets, is that they have **no memory**.
- Each input shown to them is **processed independently**, with no state kept in between inputs.
- For example, these networks would likely treat both “this movie is a bomb” and “this movie is the bomb” as being negative reviews, since they don’t consider inter-word relationships and sentence structure.
- Therefore, in order to process a sequence or a time series, you have to show the entire sequence to the network **at once**.
- For instance, in the IMDB example: an entire movie review was transformed into a single large vector and processed in one go. Such networks are called feedforward networks.

Why RNN's?

- The previous architectures **did not operate over a sequence** of vectors.
- This prohibits us from sharing any **time-dependent** information that may affect the likelihood of our predictions.
- In the case of image classification, the fact that NN saw the image of a cat at the last iteration does not help it classify the current image, because the class probabilities of these two instances are not temporally related. However, this approach may cause problem in other instances such as in sentiment analysis.

RNN Architecture

- An RNN processes sequences by iterating through the sequence elements and maintaining a state containing information relative to **what it has seen so far**.
- RNN has an internal loop; **they save relevant information in memory (also referred to as its state)** and use this information to perform predictions at subsequent time steps.
- The state of the RNN is reset between processing two different, independent sequences (such as two different IMDB reviews)
- So one sequence is still a single data point: a single input to the network.
- What changes is that this data point is **no longer processed in a single step**; rather, the network **internally loops over sequence elements**.



RNN Architecture

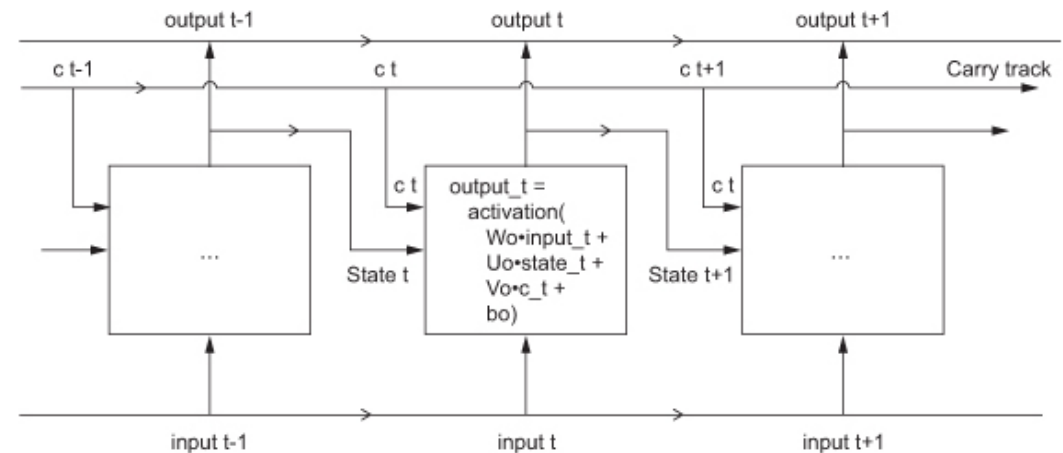
- In summary, an RNN is a for loop that reuses quantities computed during the previous iteration of the loop
- RNNs are characterized by their step function, such as the following function in this case

```
output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
```

- In practice, you'll always use a more elaborate model than the simple expression above.
- Simple RNN 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, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are **many layers deep**: as you keep adding layers to a network, the network eventually becomes untrainable.

LSTM (Long Short-Term Memory)

- LSTM adds a way to carry information across many timesteps.
- Imagine a conveyor belt running parallel to the sequence you're processing.
- Information from the sequence can jump onto the conveyor belt at any point, be transported to a later timestep, and jump off, intact, when you need it.
- This is essentially what LSTM does: it saves information for later, thus preventing older signals from gradually vanishing during processing.
- LSTM network provides a more complex solution to the problems of exploding and vanishing gradients.



GRU

- The GRU can be considered the younger sibling of the LSTM
- In essence, both leverage similar concepts to modeling **long-term dependencies**, such as remembering whether the subject of the sentence is plural, when generating following sequences.
- The underlying difference between GRUs and LSTMs is in the **computational complexity** they represent.
- Simply put, LSTMs are more complex architectures that, while computationally expensive and time-consuming to train, perform very well at breaking down the training data into meaningful and generalizable representations.
- GRUs, on the other hand, while computationally less intensive, are limited in their representational abilities compared to LSTM.
- However, not all tasks require heavyset 10-layer LSTMs!!

Example – Stock Market

- The goal of this exercise is to predict the movement of stock prices.
- We will use the S&P 500 dataset, and select a random stock to prepare for sequential modeling.
- The dataset comprises historical stock prices (opening, high, low, and closing prices) for all current S&P 500 large capital companies traded on the American stock market.
- We do acknowledge the stochasticity that lies embedded in market trends: the reality is that there is a lot of randomness that often escapes even the most predictive of models. Investor behavior is hard to foresee, as investors tend to capitalize for various motives.

Importing the Data

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
import tensorflow as tf

from keras.models import Sequential
from keras.layers import LSTM, GRU, Conv1D, Dense
from keras.layers import Dropout, Flatten, MaxPooling1D
from keras.callbacks import ModelCheckpoint, EarlyStopping

from sklearn.preprocessing import MinMaxScaler
```

```
df = pd.read_csv('all_stocks_5yr.csv')
df.head()
```

| | Date | Open | High | Low | Close | Volume | Name |
|---|------------|-------|-------|-------|-------|-----------|------|
| 0 | 2012-08-13 | 92.29 | 92.59 | 91.74 | 92.40 | 2075391.0 | MMM |
| 1 | 2012-08-14 | 92.36 | 92.50 | 92.01 | 92.30 | 1843476.0 | MMM |
| 2 | 2012-08-15 | 92.00 | 92.74 | 91.94 | 92.54 | 1983395.0 | MMM |
| 3 | 2012-08-16 | 92.75 | 93.87 | 92.21 | 93.74 | 3395145.0 | MMM |
| 4 | 2012-08-17 | 93.93 | 94.30 | 93.59 | 94.24 | 3069513.0 | MMM |

Visualizing the Data

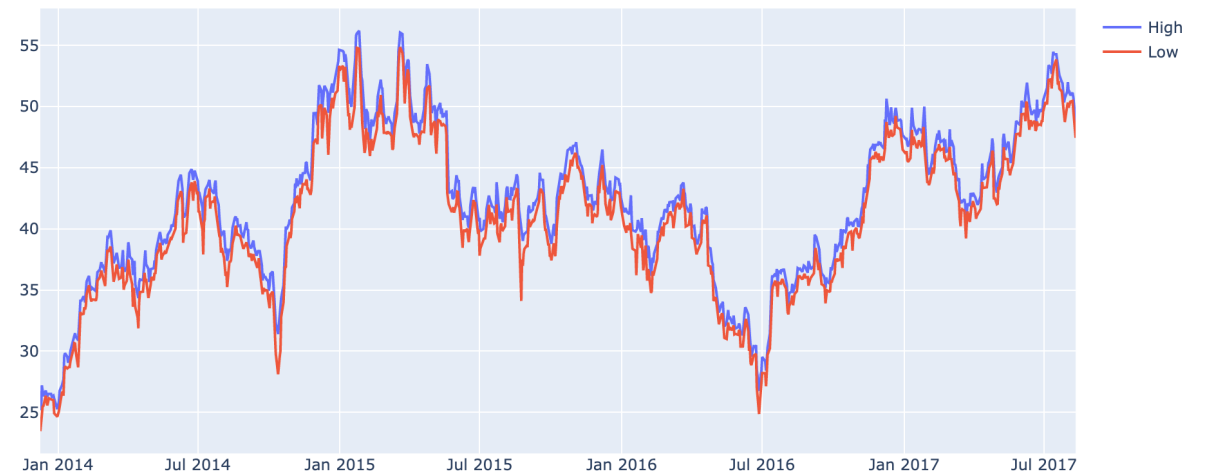
- We select a random stock (American airlines group, aal) out of the 505 different stocks in our dataset.
- Note that data is sorted by date, since we deal with a time series prediction problem where the order of the sequence is very important to our task.
- We then visually display our data by plotting out the high and low prices (on a given day) in sequential order of occurrence.
- We observe that, while slightly different from one another, the high and low prices both follow the same pattern.
- Hence, it would be redundant to use both these variables for predictive modeling, as they are highly correlated. We pick just the high values.

```
aal = df[df['Name'] == 'AAL']  
aal.shape
```

```
(926, 7)
```

```
1 fig = go.Figure()  
2 fig.add_trace(go.Scatter(x = aal['Date'], y = aal['High'], name = "High"))  
3 fig.add_trace(go.Scatter(x = aal['Date'], y = aal['Low'], name = 'Low'))  
4 fig.update_layout(title = "American Airline Stock Prices")
```

American Airline Stock Prices



Convert to Numpy Array

- We will convert the high price column on a given observation day into a NumPy array.
- We do so by calling values on that column, which returns its NumPy representation.

```
aal.loc[:, 'High']
```

```
38314    25.44  
38315    25.17  
38316    27.20  
38317    26.71  
38318    26.30  
38319    26.77
```

```
high_prices = aal.loc[:, 'High'].values  
high_prices
```

```
array([25.44, 25.17, 27.2 , 26.71, 26.3 , 26.77, 26.59, 26.23, 26.49,  
       26.49, 26.49, 26.26, 26.36, 26.1 , 25.25, 25.25, 25.82, 26.75,  
       27.2 , 27.4 , 27.68, 29.6 , 29.83, 29.53, 29.04, 29.44, 29.39,  
       30.02, 30.8 , 31.24, 31.46, 31.18, 30.87, 32.2 , 33.4 , 34.2 ,  
       34.08, 34.48, 34.38, 33.97, 34.92, 35.7 , 36.15, 35.64, 35. ,  
       35.04, 35.18, 34.95, 35.09, 35.67, 36.45, 36.66, 37.15, 37.28,  
       36.95, 36.98, 36.4 , 37.46, 38.03, 39.32, 39.29, 39.88, 39.26,  
       38.08, 37.99, 37.18, 37.77, 38.04, 37.63, 37.09, 36.82, 36.86,  
       38.2 , 38.26, 36.77, 36.5 , 36.87, 38.19, 38.9 , 38.44, 37.72,  
       37.48, 36.22, 37. , 37.38, 35.46, 34.77, 34.33, 35.59, 35.86,
```

Train, Validation, and Test Splits

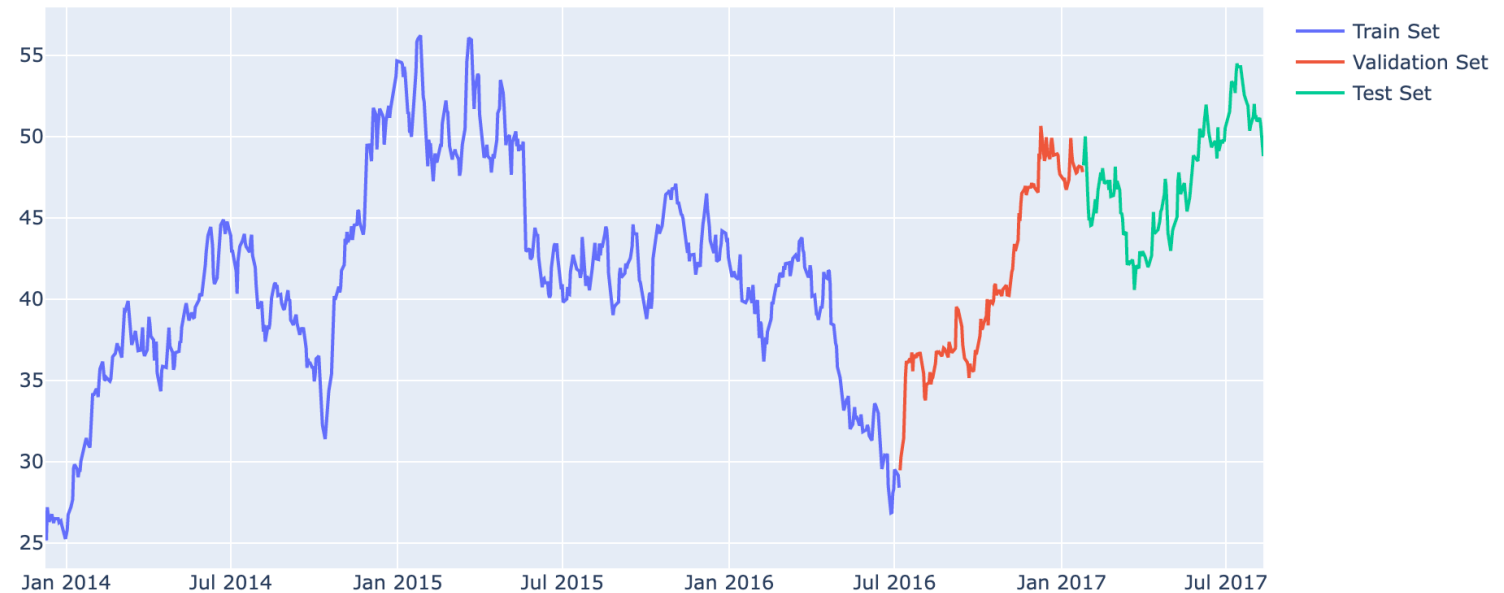
We use 70% of our data for training, 15% for validation, and 15% for test.

```
1 high_prices = aal['High'].values
2 trainPortion = round(high_prices.shape[0]*0.7)
3 valPortion = round(high_prices.shape[0]*0.15)
4 trainData = high_prices[:trainPortion]
5 valData = high_prices[trainPortion:trainPortion + valPortion]
6 testData = high_prices[trainPortion + valPortion:]
7 print('We have %d training, %d validation, and %d test points' %(len(trainData), len(valData), len(testData)))
8
```

We have 648 training, 139 validation, and 139 test data points

Visualizing the Data Subsets

- We visualize the unnormalized training, validation, and testing segments of the AAL stock data.
- Note that, the test data appears between the price range of \$40 to \$55 in the time frame of observations it represents, while training data appears in the range between \$25 to \$50+ in its respectively longer span of observation.



```
fig = go.Figure()
fig.add_trace(go.Scatter(x=aal[:trainPortion]['Date'], y = aal[:trainPortion]['High'], name = 'Train Set'))
fig.add_trace(go.Scatter(x=aal[trainPortion:trainPortion + valPortion]['Date'],
                        y = aal[trainPortion:trainPortion + valPortion]['High'],
                        name = 'Validation Set'))

fig.add_trace(go.Scatter(x=aal[trainPortion + valPortion:]['Date'],
                        y = aal[trainPortion + valPortion:]['High'],
                        name = 'Test Set'))
```

Normalizing the Data

- Recall that you need to normalize data for various machine learning tasks.

```
from sklearn.preprocessing import MinMaxScaler  
sc = MinMaxScaler(feature_range=(0, 1))
```

- You do need to reshape your data to a 2D array from a scalar array.

```
print(trainData.shape)  
trainData = trainData.reshape(-1, 1)  
print(trainData.shape)
```

(648,) →
(648, 1)

```
valData = valData.reshape(-1, 1)  
testData = testData.reshape(-1, 1)
```

Normalizing Data

Recall that we normalize data based on training parameters.

```
sc.fit(trainData)
trainNorm = sc.transform(trainData)
valNorm = sc.transform(valData)
testNorm = sc.transform(testData)
```

trainNorm

```
array([[0.00870126],
       [0.         ],
       [0.06542056],
       [0.04962939],
       [0.03641637],
       [0.051563   ],
       [0.04576217],
       [0.03416049],
       [0.04253948],
```

Creating sequences

- In order to train the RNN we need to organize our time series into segments of n consecutive values in a given sequence.
- The output for each training sequence will correspond to the stock price some timesteps into the future.
- We have two variables *look_back* and *foresight*:
 - *look_back* refers to the number of stock prices we keep in a given observation.
 - *foresight* refers to the number of steps between the last data point in the observed sequence, and the data point we aim to predict.

Example

- For example, if `look_back = 7`, the `x` sequence for would look like this:

```
1 valNorm
array([[0.13857557],
       [0.16467934],
       [0.20206252],
       [0.31324525],
       [0.326136   ],
       [0.35514019],
       [0.35095069],
       [0.35997422],
       [0.35159523],
       [0.37222043],
       [0.33483725],
       [0.36706413],
       [0.36190783],
       [0.37093136],
       [0.36835321],
       [0.36609733],
       [0.37286497],
       [0.34096036],
       [0.32968095],
```

```

var normX = [[0.13857557 0.16467934 0.20206252 0.31324525 0.326136 0.35514019
0.35095069]
[0.16467934 0.20206252 0.31324525 0.326136 0.35514019 0.35095069
0.35997422]
[0.20206252 0.31324525 0.326136 0.35514019 0.35095069 0.35997422
0.35159523]
[0.31324525 0.326136 0.35514019 0.35095069 0.35997422 0.35159523
0.37222043]
[0.326136 0.35514019 0.35095069 0.35997422 0.35159523 0.37222043
0.33483725]
[0.35514019 0.35095069 0.35997422 0.35159523 0.37222043 0.33483725
0.36706413]
[0.35095069 0.35997422 0.35159523 0.37222043 0.33483725 0.36706413
0.36190783]
[0.35997422 0.35159523 0.37222043 0.33483725 0.36706413 0.36190783
0.37093136]
[0.35159523 0.37222043 0.33483725 0.36706413 0.36190783 0.37093136
0.36835321]
[0.37222043 0.33483725 0.36706413 0.36190783 0.37093136 0.36835321

```

Example

- For example, if foresight= 6 (i.e. foresight + 1 values in the future), the y sequence for

would look like this:

```
1 valNorm
array([[0.13857557],
       [0.16467934],
       [0.20206252],
       [0.31324525],
       [0.326136   ],
       [0.35514019],
       [0.35095069],
       [0.35997422],
       [0.35159523],
       [0.37222043],
       [0.33483725],
       [0.36706413],
       [0.36190783],
       [0.37093136],
       [0.36835321],
       [0.36609733],
       [0.37286497],
       [0.34096036],
       [0.32968095],
```

```
valNormY = [0.37093136 0.36835321 0.36609733 0.37286497 0.34096036 0.32968095
0.28327425 0.27715114 0.30776668 0.31292298 0.33290364 0.30873348
0.31936835 0.32226877 0.35062842 0.37447631 0.37318724 0.37157589
0.37286497 0.36448598 0.36996455 0.38092169 0.37931034 0.37608766
0.3622301 0.39284563 0.38059942 0.37189816 0.37157589 0.38027715
0.46245569 0.45891073 0.45665485 0.42571705 0.42346117 0.38768933
0.36932001 0.36061876 0.35256204 0.34450532 0.32162423 0.34998389
0.33838221 0.33451499 0.35288431 0.37673219 0.36835321 0.38092169
0.40541412 0.4386078 0.41733806 0.42056075 0.42893974 0.44408637
0.47760232 0.42571705 0.46181115 0.47534644 0.46922333 0.47437963
0.50402836 0.50789558 0.48565904 0.49564937 0.49178215 0.48630358
0.48372543 0.49629391 0.50402836 0.50531743 0.48598131 0.48565904
0.4850145 0.53109894 0.53754431 0.5665485 0.5871737 0.57299388
0.59426362 0.6480825 0.6326136 0.66645182 0.68707702 0.69545601
0.70190139 0.68385433 0.69900097 0.6996455 0.70802449 0.69900097
0.70641315 0.69900097 0.68868837 0.68901063 0.76603287 0.75507573
0.82081856 0.77892362 0.75153078 0.75990976 0.79858202 0.77022237
0.75507573 0.78407992 0.79697067 0.78246858 0.76281018 0.76538833
0.76248791 0.73573961 0.72478247 0.71446987 0.71769256 0.70125685
0.6944892 0.71446987 0.75217531 0.79697067 0.76796648 0.74959716
0.72768289 0.72929423 0.73864003 0.74089591 0.73864003 0.72929423]
```

Example

| 1 | valNorm |
|---|--|
| | array([[0.13857557], [0.16467934], [0.20206252], [0.31324525], [0.326136], [0.35514019], [0.35095069], [0.35997422], [0.35159523], [0.37222043], [0.33483725], [0.36706413], [0.36190783], [0.37093136], [0.36835321], [0.36609733], [0.37286497], [0.34096036], [0.32968095], [0.33337435]] |

```
1 look_back = 7
2 foresight = 6
3 X, Y = [], []
4
5 X.append(valNorm[0:(0+look_back), 0])
6 Y.append(valNorm[0+look_back+foresight])
7 print(np.array(X)) # X & Y are lists
8 print(np.array(Y))
```

```
[[0.13857557 0.16467934 0.20206252 0.31324525 0.326136      0.35514019
  0.35095069]]
[[0.37093136]]
```

```
1 X.append(valNorm[1:(1+look_back), 0])
2 Y.append(valNorm[1+look_back+foresight])
3 print(np.array(X)) # X & Y are lists
4 print(np.array(Y))
```

```
[[0.13857557 0.16467934 0.20206252 0.31324525 0.326136      0.35514019
  0.35095069]
 [0.16467934 0.20206252 0.31324525 0.326136      0.35514019 0.35095069
  0.35997422]]
[[0.37093136]
 [0.36835321]]
```


Creating sequences

```
def createSeq(dataset, look_back, foresight):  
    X, Y = [], []  
    for i in range(len(dataset)-look_back-foresight):  
        obs = dataset[i:(i+look_back), 0] #Sequence of "look_back"  
        X.append(obs) #Append stock price value occurring "foresight+1"  
        Y.append(dataset[i + (look_back+foresight), 0])  
    return np.array(X), np.array(Y)
```


Sequences

```
trainNormX, trainNormY = createSeq(trainNorm, look_back = 7, foresight = 6)
```

- What will be the length of trainNorm after creating the sequence?

```
print(trainNormX.shape, trainNormY.shape)
```

- $648 - 7 - 6 = 635$

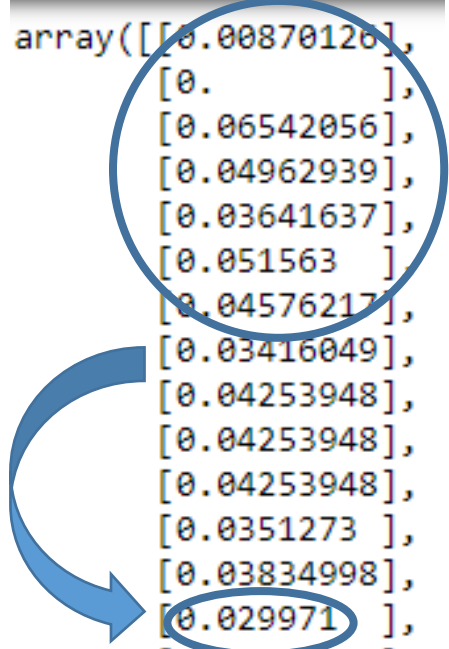
```
(635, 7) (635,)
```

```
print(trainNormX[0,:],trainNormY[0])
```

```
[0.00870126 0.          0.06542056 0.04962939 0.03641637 0.051563
 0.04576217] 0.029970995810505996
```

trainNorm

```
array([[0.00870126],
       [0.          ],
       [0.06542056],
       [0.04962939],
       [0.03641637],
       [0.051563   ],
       [0.04576217],
       [0.03416049],
       [0.04253948],
       [0.04253948],
       [0.04253948],
       [0.0351273 ],
       [0.03834998],
       [0.029971   ],
       [0.00257815],
       [0.0057015 ]])
```



Creating Sequences for Validation and Test

- You typically need to experiment with different values of look_back and foresight to assess how larger look_back and foresight values each affect the predictive prowess of your model.
- In practice, you will experience diminishing returns on either side for both values.

```
valNormX, valNormY = createSeq(valNorm, look_back = 7, foresight = 6)  
testNormX, testNormY = createSeq(testNorm, look_back = 7, foresight = 6)
```

Imports

```
from keras.models import Sequential
from keras.layers import LSTM, GRU, Dense
from keras.layers import Dropout, Flatten
from keras.callbacks import ModelCheckpoint, EarlyStopping
```

Simple LSTM

```
model = Sequential()  
model.add(LSTM(32, input_shape=(7,1), dropout=0.1, recurrent_dropout=0.1))  
model.add(Dense(1, activation='linear'))  
model.compile(loss='mae', optimizer='adam', metrics=['mean_absolute_error'])  
model.summary()
```

| Layer (type) | Output Shape | Param # |
|-------------------------|--------------|---------|
| ===== | | |
| lstm_1 (LSTM) | (None, 32) | 4352 |
| ===== | | |
| dense_2 (Dense) | (None, 1) | 33 |
| ===== | | |
| Total params: 4,385 | | |
| Trainable params: 4,385 | | |
| Non-trainable params: 0 | | |

Fitting Simple LSTM

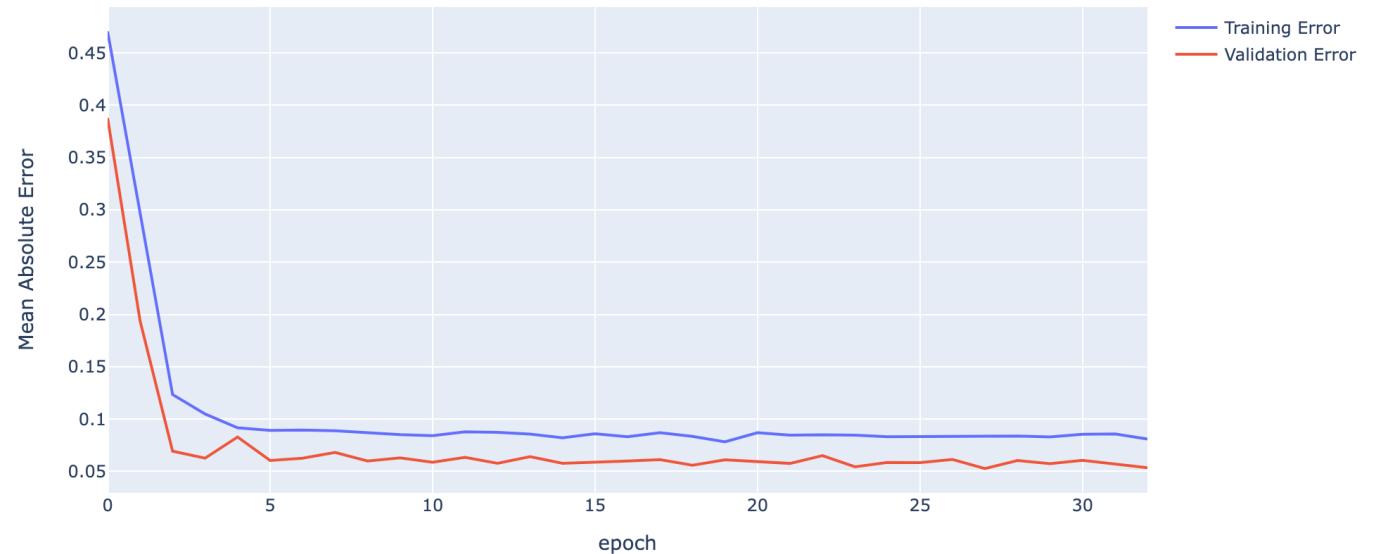
```
# network_name = 'lstm'
# filepath = network_name + "_epoch-{epoch:02d}-loss-{loss:.4f}-.hdf5"
#checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True, mode='min')
checkpoint = EarlyStopping(monitor='val_loss',patience=5, verbose=1, mode='auto', restore_best_weights=True)
callbacks_list = [checkpoint]
network = model.fit(trainNormX, trainNormY, validation_data=(valNormX, valNormY),
                    epochs=100, batch_size=64,callbacks=callbacks_list)
```

Error Plot for Simple LSTM

```
valMae = round(network.history['val_loss'][-1],2)
fig = go.Figure()
fig.add_trace(go.Scatter(y=network.history['loss'],
                        mode='lines',
                        name='Training Error'))
fig.add_trace(go.Scatter(y=network.history['val_loss'],
                        mode='lines',
                        name='Validation Error'))
fig.update_layout(yaxis_title = 'Mean Absolute Error',
                  xaxis_title = 'epoch',
                  title_text='Normalized MAE Validation = ' +
                        str(valMae))

fig.show()
```

Normalized MAE Validation = 0.05

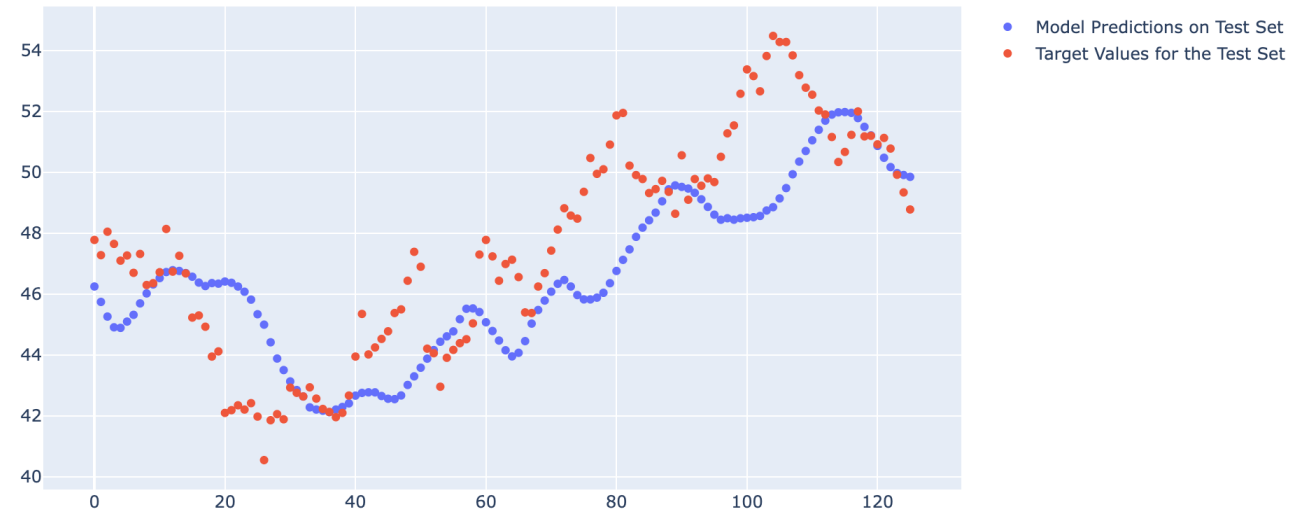


Simple LSTM Performance on Test Set

```
testNormPred= model.predict(testNormX)
testPred = sc.inverse_transform(testNormPred)
testY = sc.inverse_transform(testNormY.reshape(-1,1))
testMae = tf.keras.metrics.mean_absolute_error(testY, testPred)

fig = go.Figure()
fig.add_trace(go.Scatter(y=testPred.reshape(-1,),
                        mode='markers',
                        name='Model Predictions on Test Set'))
fig.add_trace(go.Scatter(y=testY.reshape(-1,),
                        mode='markers',
                        name='Target Values for the Test Set'))
fig.update_layout(title_text='Unnormalized MAE Test = '
                        + str(np.mean(testMae)))
fig.show()
```

Unnormalized MAE Test = 1.873413



Simple GRU

```
model = Sequential()  
model.add(GRU(32, input_shape=(7,1), dropout=0.1, recurrent_dropout=0.1))  
model.add(Dense(1, activation='linear'))  
model.compile(loss='mae', optimizer='adam', metrics=['mean_absolute_error'])  
model.summary()
```

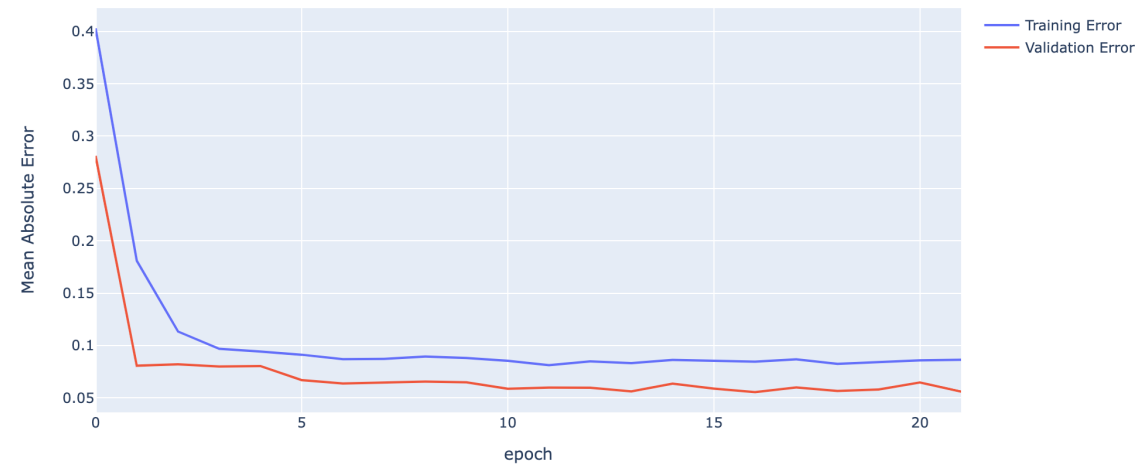

Fitting Simple GRU

```
# network_name = 'gru'
# filepath = network_name + "_epoch-{epoch:02d}-loss-{loss:.4f}-.hdf5"
#checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=0, save_best_only=True, mode='min')
checkpoint = EarlyStopping(monitor='val_loss',patience=5, verbose=1, mode='auto', restore_best_weights=True)
callbacks_list = [checkpoint]
network = model.fit(trainNormX, trainNormY, validation_data=(valNormX, valNormY),
                    epochs=100, batch_size=64,callbacks=callbacks_list)
```

Error Plot for Simple GRU

```
valMae = round(network.history['val_loss'][-1],2)
fig = go.Figure()
fig.add_trace(go.Scatter(y=network.history['loss'],
                        mode='lines',
                        name='Training Error'))
fig.add_trace(go.Scatter(y=network.history['val_loss'],
                        mode='lines',
                        name='Validation Error'))
fig.update_layout(yaxis_title = 'Mean Absolute Error',
                  xaxis_title = 'epoch',
                  title_text='Normalized MAE Validation = ' +
                        str(valMae))
fig.show()
```

Normalized MAE Validation = 0.06

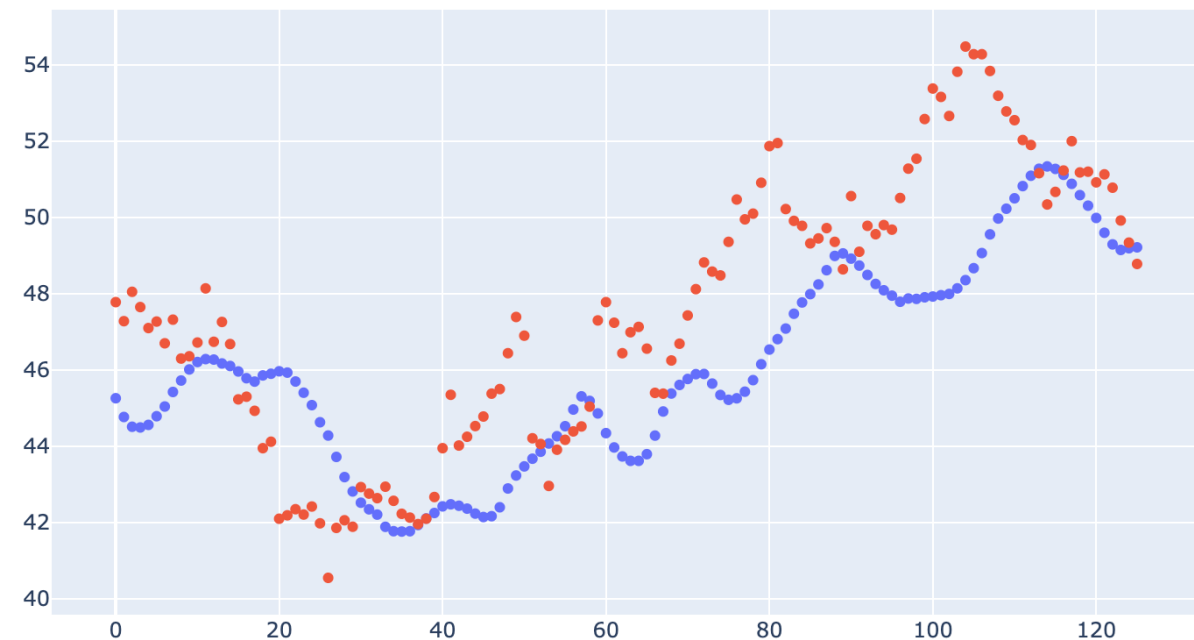


Simple GRU Performance on Test Set

```
testNormPred= model.predict(testNormX)
testPred = sc.inverse_transform(testNormPred)
testY = sc.inverse_transform(testNormY.reshape(-1,1))
testMae = tf.keras.metrics.mean_absolute_error(testY, testPred)

fig = go.Figure()
fig.add_trace(go.Scatter(y=testPred.reshape(-1,),
                        mode='markers',
                        name='Model Predictions on Test Set'))
fig.add_trace(go.Scatter(y=testY.reshape(-1,),
                        mode='markers',
                        name='Target Values for the Test Set'))
fig.update_layout(title_text='Unnormalized MAE Test = '
                        + str(np.mean(testMae)))
fig.show()
```

Unnormalized MAE Test = 2.102357

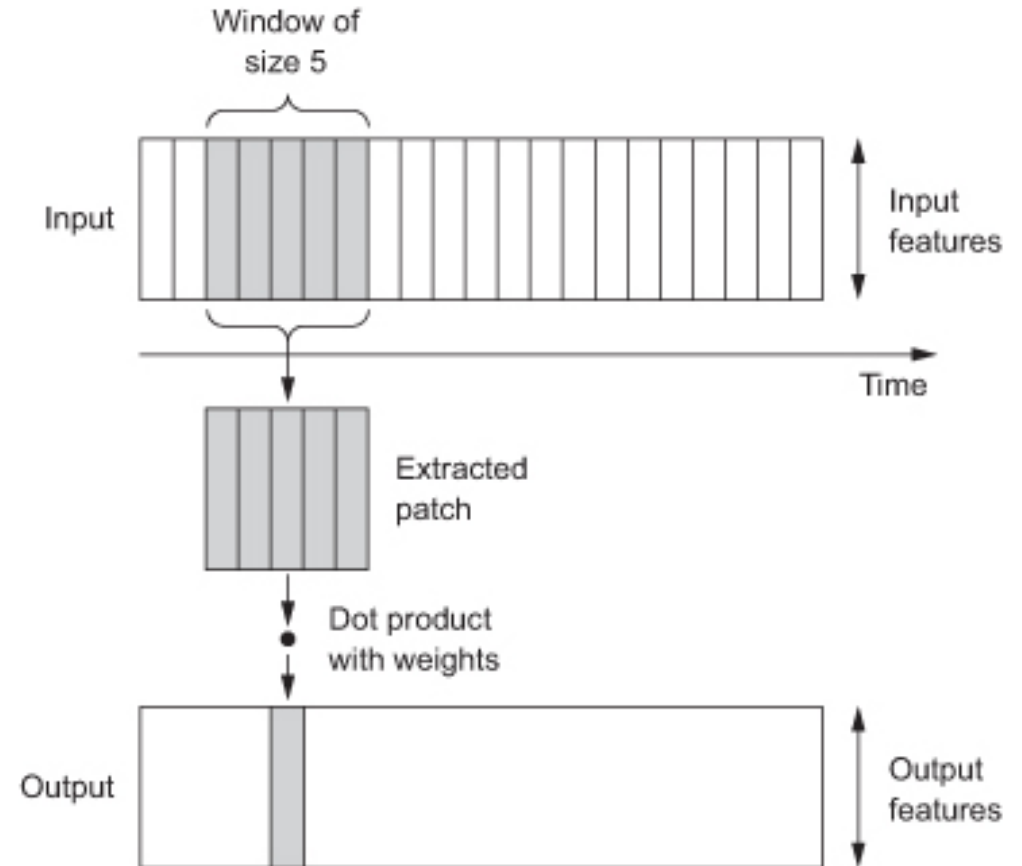


Sequence processing with convnets

- We saw that convnets perform particularly well on **computer vision** problems
- This is due to their ability to extract features from **local input patches** and allow for representation modularity and data efficiency.
- The same properties that make convnets excel at computer vision also make them highly relevant to sequence processing.
- **Time can be treated as a spatial dimension**, like the height or width of a 2D image.
- convnets can be competitive with RNNs on certain sequence-processing problems, usually at a considerably cheaper computational cost.

Understanding 1D convolution for Sequence Data

- The convolution layers introduced previously were 2D convolutions, extracting 2D patches from image tensors and applying an identical transformation to every patch.
- In the same way, you can use **1D convolutions**, extracting local 1D patches (subsequences) from sequences.
- Such 1D convolution layers can recognize local patterns in a sequence.
- Because the same input transformation is performed on every patch, a pattern learned at a certain position in a sentence can later be recognized at a different position, making 1D convnets translation invariant (for temporal translations).
- For instance, a 1D convnet processing sequences of characters using convolution windows of size 5 should be able to learn words length 5 or less, and recognize these words in any context in an input sequence.



1D pooling for sequence data

- In Keras, you use a 1D convnet via the Conv1D layer, which has an interface similar to Conv2D.
- It takes as input 3D tensors with shape (samples, time, features) and returns similarly shaped 3D tensors.
- The convolution window is a 1D window on the temporal axis: axis 1 in the input tensor.
- Keep in mind that here you can use larger convolution windows with 1D convnets.
- With a 2D convolution layer, a 3×3 convolution window contains $3 \times 3 = 9$ feature vectors; but with a 1D convolution layer, a convolution window of size 3 contains only 3 feature vectors. You can thus easily afford 1D convolution windows of size 7 or 9.

1D Convnet

```
1 model = Sequential()
2 model.add(Conv1D(64, kernel_size=5, input_shape=(7,1), activation='relu'))
3 model.add(MaxPooling1D(pool_size = 2))
4 model.add(Dense(1, activation='linear'))
5 model.compile(loss='mae', optimizer='adam', metrics = ['mean_absolute_error'])
6 model.summary()
```

Model: "sequential_38"

| Layer (type) | Output Shape | Param # |
|---------------------------------|---------------|---------|
| ===== | | |
| conv1d_32 (Conv1D) | (None, 3, 64) | 384 |
| max_pooling1d_25 (MaxPooling1D) | (None, 1, 64) | 0 |
| dense_26 (Dense) | (None, 1, 1) | 65 |
| ===== | | |
| Total params: 449 | | |
| Trainable params: 449 | | |
| Non-trainable params: 0 | | |

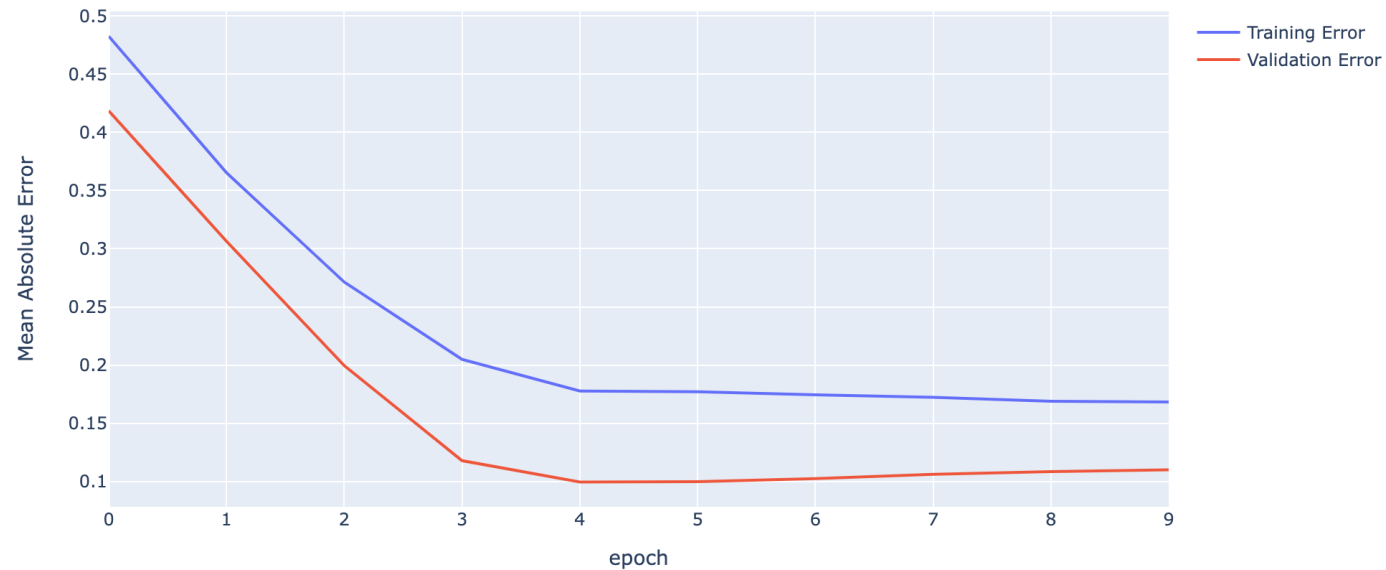
```
checkpoint = EarlyStopping(monitor='val_loss',patience=5, verbose=1, mode='auto', restore_best_weights=True)
callbacks_list = [checkpoint]
network = model.fit(trainNormX, trainNormY, validation_data=(valNormX, valNormY),
                    epochs=100, batch_size=64,callbacks=callbacks_list)
```

Error Plot for Conv1D

```
: 1 valMae = round(network.history['val_loss'][-1],2)
2 fig = go.Figure()
3 fig.add_trace(go.Scatter(y=network.history['loss'],
4                           mode='lines',
5                           name='Training Error'))
6 fig.add_trace(go.Scatter(y=network.history['val_loss'],
7                           mode='lines',
8                           name='Validation Error'))
9 fig.update_layout(yaxis_title = 'Mean Absolute Error',
10                  xaxis_title = 'epoch',
11                  title_text='Normalized MAE Validation = ' +
12                          str(valMae))
13 fig.show()
```



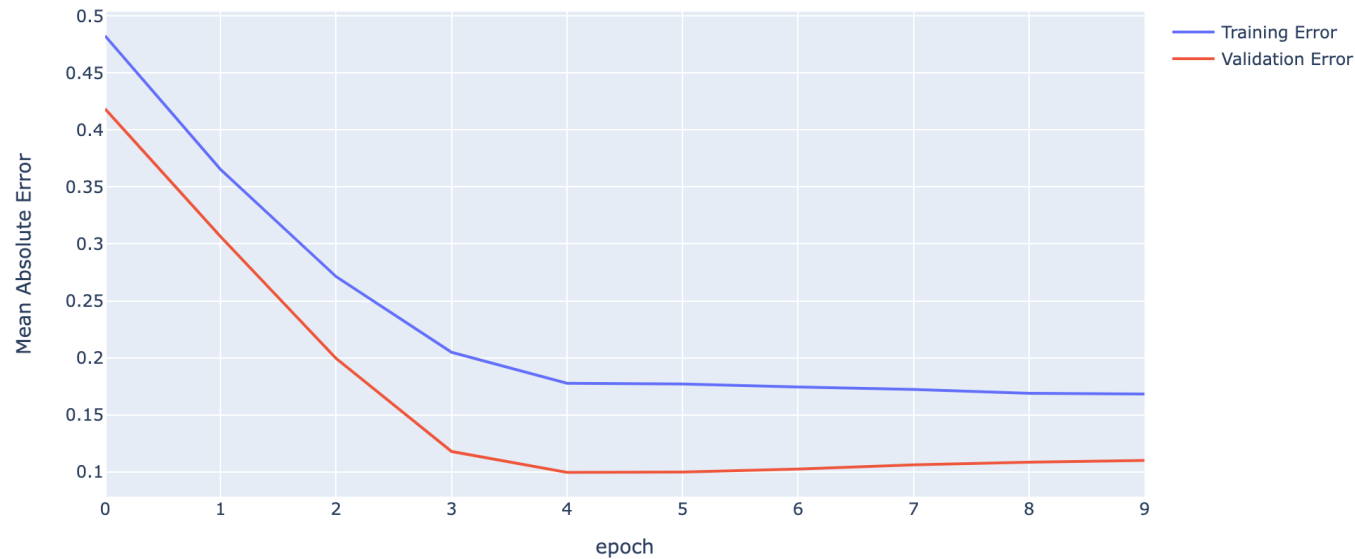
Normalized MAE Validation = 0.11



Conv1D Performance on Test Set

```
1 valMae = round(network.history['val_loss'][-1],2)
2 fig = go.Figure()
3 fig.add_trace(go.Scatter(y=network.history['loss'],
4                           mode='lines',
5                           name='Training Error'))
6 fig.add_trace(go.Scatter(y=network.history['val_loss'],
7                           mode='lines',
8                           name='Validation Error'))
9 fig.update_layout(yaxis_title = 'Mean Absolute Error',
10                  xaxis_title = 'epoch',
11                  title_text='Normalized MAE Validation = ' +
12                           str(valMae))
13 fig.show()
```

Normalized MAE Validation = 0.11



Advantages and Drawbacks of RNN's

| Advantages | Drawbacks |
|---|--|
| <ul style="list-style-type: none">• Possibility of processing input of any length• Model size not increasing with size of input• Computation takes into account historical information• Weights are shared across time | <ul style="list-style-type: none">• Computation being slow• Difficulty of accessing information from a long time ago• Cannot consider any future input for the current state |

References

- [Hands-on Neural Networks with Keras](#) by Niloy Purkait
- [Deep Learning with Python](#) by Chollet