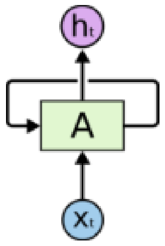
Implementation of RNN with LSTM

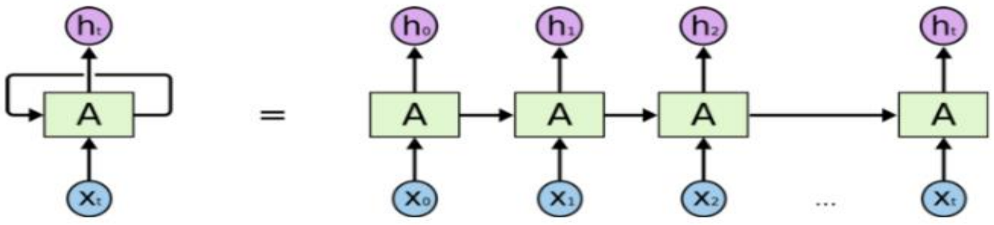
- **RNN(Recurrent Neural Networks)** are are a type of artificial neural network that are able to recognize and predict sequences of data such as text, genomes, handwriting, spoken word, or numerical time series data. They have loops that allow a consistent flow of information and can work on sequences of arbitrary lengths.
- Using an internal state (memory) to process a sequence of inputs, RNNs are already being used to solve a number of problems: Language translation, Speech recognition, Time series data
- RNN Structure:



Recurrent Neural Networks have loops.

A — Neural Network., Xt- Input., ht — Output.

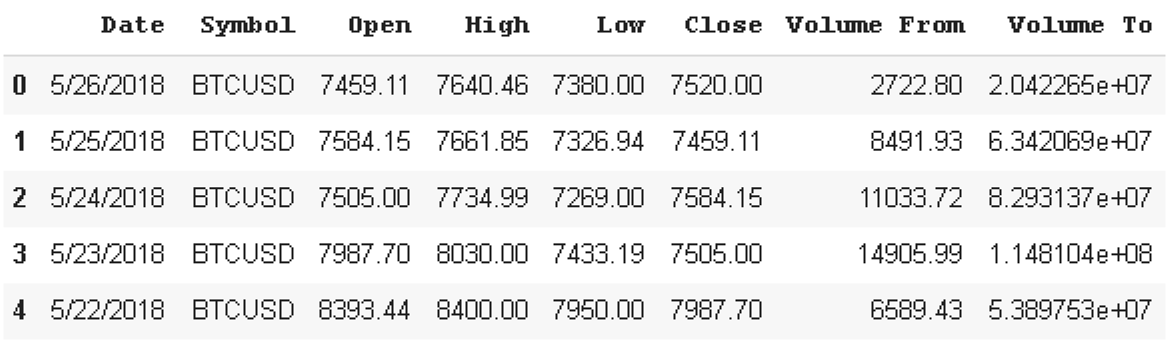
- Loops ensure a consistent flow of information. **A** (chunk of neural network) produces an output **ht** based on the input **Xt** .
- RNNs can also be viewed as multiple copies of the same network, each passing information to its successor.



when Xt comes in, the hidden state from Xt-1 will be concatenated with Xt and become the input for the Network at time t. This process will be repeated for every sample in a time-series.

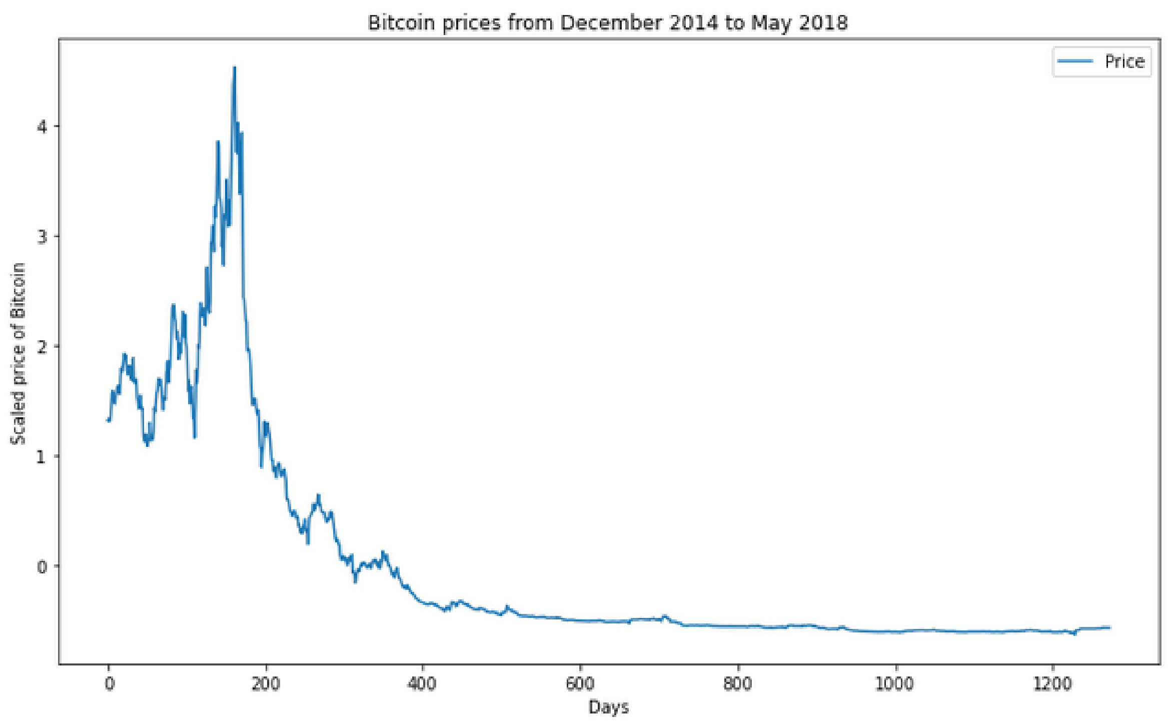
- RNNs have a major setback called vanishing gradient that is, they have difficulties in learning long-range dependencies (relationship between entities that are several steps apart).

- Imagine we have the price of bitcoin for December 2014, which was say $350, and we want to correctly predict the bitcoin price for the months of April and May 2018. Using RNNs, our model won't be able to predict the prices for these months accurately due to the long range memory deficiency. To solve this issue, a special kind of RNN called **Long Short-Term Memory cell (LSTM)** was developed.
- LSTM has special neuron for memorizing long-term dependencies. LSTM contains an internal state variable which is passed from one cell to the other and modified by **Operation Gates**
- Data for implementation: I have used time series analysis using the LSTMs to predict the prices of bitcoin from the Dec 2014 to May2018.
- I have used google Colabs development environment because of GPU.
- Before starting let me details about data, data has 1273 observation with 8 attributes



	Date	Symbol	Open	High	Low	Close	Volume From	Volume To
0	5/26/2018	BTCUSD	7459.11	7640.46	7380.00	7520.00	2722.80	2.042265e+07
1	5/25/2018	BTCUSD	7584.15	7661.85	7326.94	7459.11	8491.93	6.342069e+07
2	5/24/2018	BTCUSD	7505.00	7734.99	7269.00	7584.15	11033.72	8.293137e+07
3	5/23/2018	BTCUSD	7987.70	8030.00	7433.19	7505.00	14905.99	1.148104e+08
4	5/22/2018	BTCUSD	8393.44	8400.00	7950.00	7987.70	6589.43	5.389753e+07

- We are going to select the bitcoin closing price as our target variable to predict.
- Data preprocessing is done by Sklearn preprocessing module that allows us to scale our data and then fit in our model
- Lets take the scaled data over the given time period



Features and label dataset:

- This function is used to create the features and labels for our data set by windowing the data.
- **Input**: data — this is the dataset we are using .
- **Window_size** — how many data points we are going to use to predict the next datapoint in the sequence. (Example if window_size=7 we are going to use the previous 7 days to predict the bitcoin price for today).
- **Outputs**: X — features split into windows of data points(if windows_size=1, X=[len(data)-1,1]).
- y — labels — this is the next number in the sequence that we're trying to predict.

```
def window_data(data, window_size):
    X = []
    y = []

    i = 0
    while (i + window_size) <= len(data) - 1:
        X.append(data[i:i+window_size])
        y.append(data[i+window_size])

        i += 1
    assert len(X) ==  len(y)
    return X, y
#windowing the data with window_data function
X, y = window_data(scaled_data, 7)
```

- Splitting the data into training and test sets is crucial for getting a realistic estimate of our model’s performance. We have used 80% (1018) of the dataset as the training set and the remaining 20% (248) as the validation set.

```
#we now split the data into training and test set
import numpy as np
X_train  = np.array(X[:1018])
y_train = np.array(y[:1018])

X_test = np.array(X[1018:])
y_test = np.array(y[1018:])

print("X_train size: {}".format(X_train.shape))
print("y_train size: {}".format(y_train.shape))
print("X_test size: {}".format(X_test.shape))
print("y_test size: {}".format(y_test.shape))
```

- **Building the Network:** We need to specify hyperparameters which explain higher- level structural information of about the model

- **batch_size** — This is the number of windows of data we are passing at once.

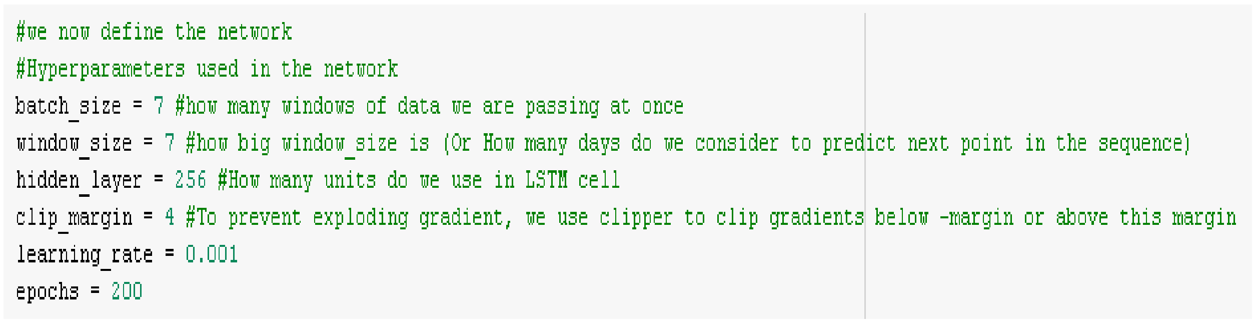
- **window_size** — The number of days we consider to predict the bitcoin price for our case.

- **hidden_layers** — This is the number of units we use in our LSTM cell.

- **clip_margin** — This is to prevent exploding the gradient — we use clipper to clip gradients below above this margin.

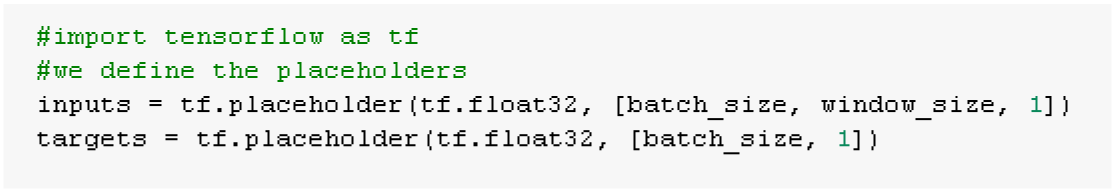
- **learning_rate** — This is a an optimization method that aims to reduce the loss function.

- **epochs** — This is the number of iterations (forward and back propagation) our model needs to make.



```
#we now define the network
#Hyperparameters used in the network
batch_size = 7 #how many windows of data we are passing at once
window_size = 7 #how big window_size is (Or How many days do we consider to predict next point in the sequence)
hidden_layer = 256 #How many units do we use in LSTM cell
clip_margin = 4 #To prevent exploding gradient, we use clipper to clip gradients below -margin or above this margin
learning_rate = 0.001
epochs = 200
```

- Placeholders allows us to send different data within our network with the tf.placeholder() command.



```
#import tensorflow as tf
#we define the placeholders
inputs = tf.placeholder(tf.float32, [batch_size, window_size, 1])
targets = tf.placeholder(tf.float32, [batch_size, 1])
```

- **LSTM Weights:** LSTM weights are determined by Operation Gates which include: Forget, Input and Output gates.
- **Forget Gate**

  $ft = \sigma(Wf[ht-1, Xt]+bf)$

  This is a sigmoid layer that takes the output at t-1 and the current input at time t and then combines them into a single tensor. It then applies linear transformation followed by a sigmoid.

  The output of the gate is between 0 and 1 due to the sigmoid. This number is then multiplied with the internal state, and that is why the gate is called forget gate. If ft =0 ,then the previous internal state is completely forgotten, while if ft =1, it will be passed unaltered.
- **Input Gate**

  $it=\sigma(Wi[ht-1,Xt]+bi)$

  This state takes the previous output together with the new input and passes them through another sigmoid layer. This gate returns a value between 0 and 1. The value of the input gate is then multiplied with the output of the candidate layer.

  $Ct=\tanh(Wi[ht-1,Xt]+bi)$

  This layer applies hyperbolic tangent to the mix of the input and previous output, returning the candidate vector. The candidate vector is then added to the internal state, which is updated with this rule:

  $Ct=ft *Ct-1+it*Ct$

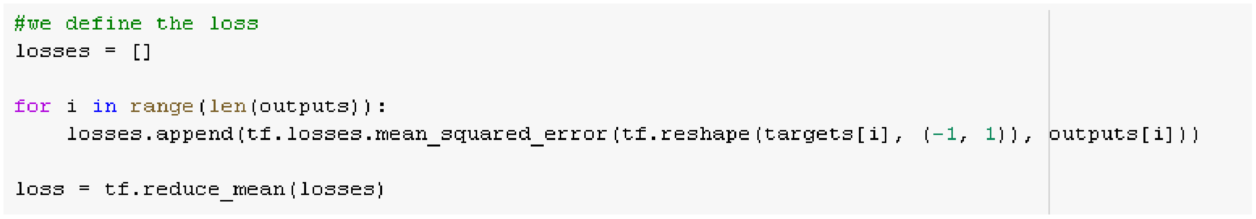
  The previous state is multiplied by the forget gate, and then added to the fraction of the new candidate allowed by the output gate.
- **Output Gate**

  $Ot=\sigma(Wo[ht-1,Xt]+bo)$

  $ht=Ot*\tanh Ct$

This gate controls how much of the internal state is passed to the output and works in a similar manner to the other gates.

- **Network Loop**: A loop for the network is created which iterates through every window in the batch creating the batch_states as all zeros .The output is the used for predicting the bitcoin price.
- **Defining Loss**: Here we will use the mean_squared_error function for the loss to minimize the errors.

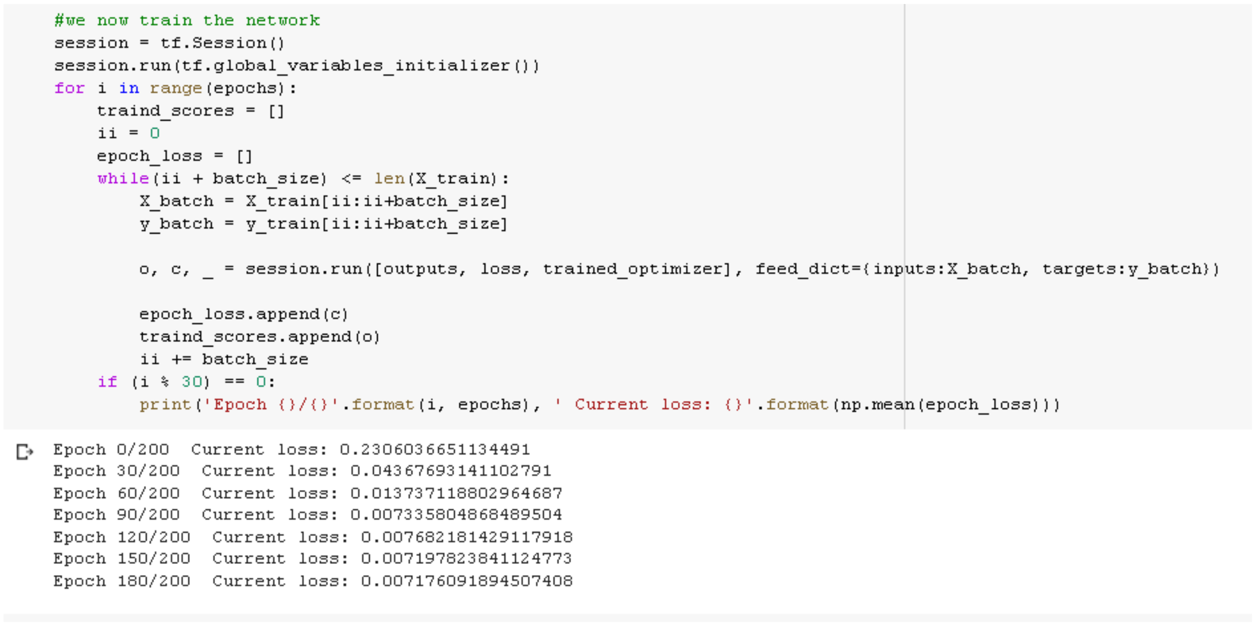


```
#we define the loss
losses = []

for i in range(len(outputs)):
    losses.append(tf.losses.mean_squared_error(tf.reshape(targets[i], (-1, 1)), outputs[i]))

loss = tf.reduce_mean(losses)
```

- **Training the network:** We now train the network with the number of epochs (200), which we had initialized, and then observe the change in our loss through time. The current loss decreases with the increase in the epochs as observed, increasing our model accuracy in predicting the bitcoin prices.



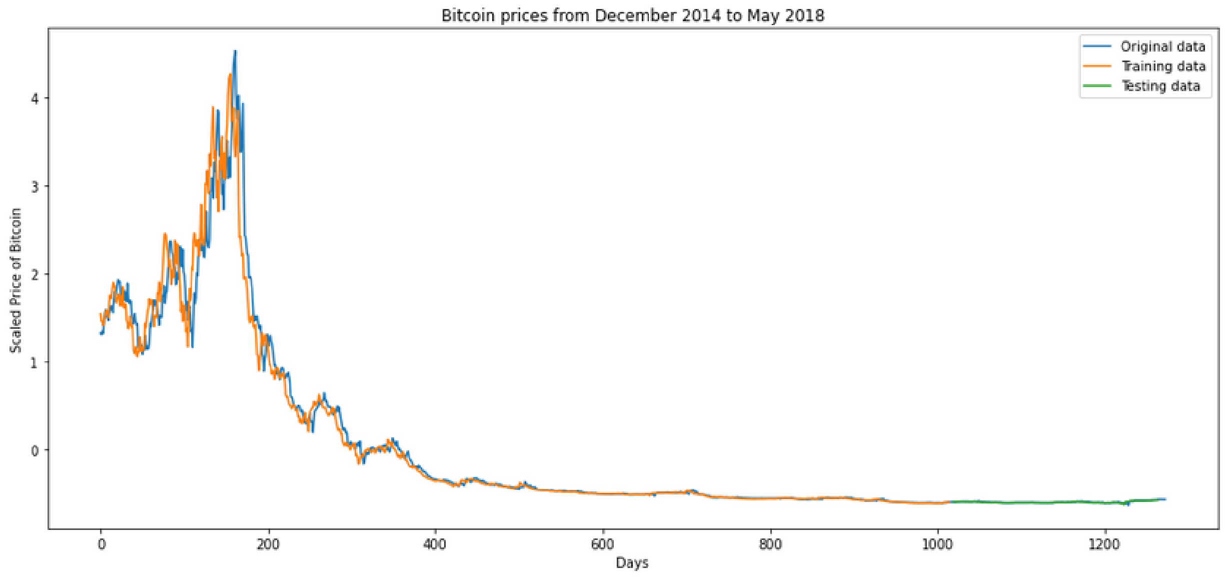
```
#we now train the network
session = tf.Session()
session.run(tf.global_variables_initializer())
for i in range(epochs):
    traind_scores = []
    ii = 0
    epoch_loss = []
    while(ii + batch_size) <= len(X_train):
        X_batch = X_train[ii:ii+batch_size]
        y_batch = y_train[ii:ii+batch_size]

        o, c, _ = session.run([outputs, loss, trained_optimizer], feed_dict={inputs:X_batch, targets:y_batch})

        epoch_loss.append(c)
        traind_scores.append(o)
        ii += batch_size
    if (i % 30) == 0:
        print('Epoch {}/{}'.format(i, epochs), ' Current loss: {}'.format(np.mean(epoch_loss)))
```

```
Epoch 0/200  Current loss: 0.2306036651134491
Epoch 30/200  Current loss: 0.043676931411102791
Epoch 60/200  Current loss: 0.013737118802964687
Epoch 90/200  Current loss: 0.007335804868489504
Epoch 120/200  Current loss: 0.007682181429117918
Epoch 150/200  Current loss: 0.007197823841124773
Epoch 180/200  Current loss: 0.007176091894507408
```

- **OUTPUT**:- Our model has been able to accurately predict the bitcoin prices based on the original data by implementing LSTMs cells. We could improve the model performance by reducing the window length from 7 days to 3 days. You can tweak the full code to optimize your model performance.


Bitcoin prices from December 2014 to May 2018
Original data
Training data
Testing data
Scaled Price of Bitcoin
Days