Pattern Recognition Report Paper Regarding Analysis of Recurrent Neural Network (RNN) and Long Short Term Memory (LSTM)

Irvan Ariyanto
Informatics, Institut Teknologi Bandung, Indonesia
235019028@std.stei.itb.ac.id

Abstract:

This writing is trying to analyze the problem using Recurrent Neural Network (RNN) and Long Short Term Memory (LSTM). The methods are explained continuously from one process to another which entirely correlated to illustrate the results. This writing reports how the program is running by more elaboration with forward and backward methods on Recurrent Neural Network and Long Short Term Memory. The furthermore analysis will be explained comprehensively by showing the programmer's data results and discussion regarding how the system works.

1 Introduction

In everyday life, we often gather a number of sequential data, such as news text data, weather forecasts, sensors, traffic videos, etc. Recurrent Neural Networks (RNN) is a form of Artificial Neural Networks (ANN) architecture specifically designed to process sequential data. RNN is usually utilized to complete tasks related to time-series data, such as weather forecast data. For example, today's weather can depend on the previous day's weather, if the previous day was cloudy, then it is likely to rain today.

RNN does not just throw away information from the past in the learning process. This is what distinguishes RNN from ordinary ANN. RNN is able to store memory/feedback (feedback loop) that allows it to recognize patterns of data smoothly, then utilize it to create accurate predictions. The way in which RNN is able to store the past information is by using a loop within its architecture, which automatically keeps information from the past stored.

2 Theoretical Framework

2.1 Recurrent Neural Network (RNN)

RNN (Recurrent Neural Network) is one of the types of ANN (Artificial Neural Network) which in the "core" (or so called as the cell) loop. This means that the output of this cell will also become the input.[1] The idea behind RNNs is to make use of sequential information. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that's a very bad idea. If you want to predict the next word in a sentence you better know which words came before it. RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far. In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps (more on this later).[2]

2.2 Long Short Term Memory (LSTM)

Long Short Term Memory (LSTM) is very powerful in sequence prediction problems because they're able to store past information This is important in our case because the previous price of a stock is crucial in predicting its future price. Long Short Term Memory Neural Network (LSTM) is one type of RNN. LSTM stores information about patterns in the data. LSTM can learn which data will be stored and which data will be discarded, because each neuron LSTM has several gates that regulate the memory of each neuron

---

[1] https://www.academia.edu/34438387/PENGENALAN_LSTM_LONG_SHORT_TERM_MEMORY_
[2] http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

itself.[3] What distinguishes LSTM from RNN is only the cell contents that are more complex, the rest are the same as the principle.

3 Analytical Methods

3.1 RNN Handson

- *Packages*

  Import all the packages that will need during this assignment.

- *RNN Cell Forward*

  RNN cell forward is utilized to calculate the hidden state by using the formulas below:

  Next formula activation state :

  $$a^{(t)} = \tanh\left(W_{ax}x^{(t)} + W_{aa}a^{(t-1)} + b_a\right)$$

  Prediction formula at timestep :

  $$\hat{y}^{(t)} = softmax(W_{ya}a^{(t)} + b_y)$$

  There are few parameters that should be defined by the programmer before running the RNN Cell Forward function including:

  - **xt** : your input data at timestep "t", numpy array of shape (n_x, m).
  - **a_prev** : Hidden state at timestep "t-1",numpy array of shape (n_a, m)
  - **parameters** : python dictionary containing:
  - **Wax** : Weight matrix multiplying the input, numpy array of shape (n_a, n_x)
  - **Waa** : Weight matrix multiplying the hidden state, numpy array of shape (n_a, n_a)
  - **Wya** : Weight matrix relating the hidden-state to the output, numpy array of shape (n_y, n_a)
  - **ba** : Bias, numpy array of shape (n_a, 1)
  - **by** : Bias relating the hidden-state to the output, numpy array of shape (n_y, 1)

  Therefore, the program will results the next hidden step and prediction at time step after accomplishing the overall calculation with the given formulas.

  Returns:

  - **a_next** : next hidden state, of shape (n_a, m)
  - **yt_pred** : prediction at timestep "t", numpy array of shape (n_y, m)
  - **cache** : tuple of values needed for the backward pass, contains (a_next, a_prev, xt, parameters)

- *RNN Forward*

  In this stage, each cell takes as input the hidden state from the previous cell ($a\langle t-1\rangle$) and the current time-step's input data ($x\langle t\rangle$). It outputs a hidden state ($a\langle t\rangle$) and a prediction ($y\langle t\rangle$) for this time-step and this process is continuously repeated to produce the output.

- *LSTM Cell Forward*

  LSTM-cell. This tracks and updates a "cell state" or memory variable $c\langle t\rangle$ at every time-step, which can be different from $a\langle t\rangle$

  Following are the arguments and parameters that must be defined by the programmer before running the LSTM Cell Forward function including:

  Arguments:

  - **xt** : Input data at timestep "t", numpy array of shape (n_x, m).
  - **a_prev** : Hidden state at timestep "t-1", numpy array of shape (n_a, m)
  - **c_prev** : Memory state at timestep "t-1", numpy array of shape (n_a, m)

  Parameters: python dictionary containing:

  - **Wf** : Weight matrix of the forget gate, numpy array of shape (n_a, n_a + n_x)

[3] Putra, Muhammad Wildan. "Analisis Dan Implementasi Long Short Term Memory Neural Network Untuk Prediksi Harga Bitcoin." e-Proceeding of Engineering, August 2, 2018.

- **bf** : Bias of the forget gate, numpy array of shape (n_a, 1)
- **Wi** : Weight matrix of the update gate, numpy array of shape (n_a, n_a + n_x)
- **bi** : Bias of the update gate, numpy array of shape (n_a, 1)
- **Wc** : Weight matrix of the first "tanh", numpy array of shape (n_a, n_a + n_x)
- **bc** : Bias of the first "tanh", numpy array of shape (n_a, 1)
- **Wo** : Weight matrix of the output gate, numpy array of shape (n_a, n_a + n_x)
- **bo** : Bias of the output gate, numpy array of shape (n_a, 1)
- **Wy** : Weight matrix relating the hidden-state to the output, numpy array of shape

So it shows the following results:
Returns:

- **a_next** : next hidden state, of shape (n_a, m)
- **c_next** : next memory state, of shape (n_a, m)
- **yt_pred** : prediction at timestep "t", numpy array of shape (n_y, m)
- **cache** : tuple of values needed for the backward pass, contains (a_next, c_next, a_prev, c_prev, xt, parameters)

- *LSTM Forward*
  Now that you have implemented one step of an LSTM, you can now iterate this over this using a for-loop to process a sequence of TxTx inputs.
- *RNN Cell Backward*
  Compute the gradient from the loss function towards all parameters by looking for partial derivative from the given function for each cell.
  Following are the arguments that must be defined by the programmer:
  Arguments:
  - **da_next** : Gradient of loss with respect to next hidden state
  - **cache** : python dictionary containing useful values (output of rnn_cell_forward())

  So it shows the following results:
  Returns:
  - **gradients** : python dictionary containing:
  - **dx** : Gradients of input data, of shape (n_x, m)
  - **da_prev** : Gradients of previous hidden state, of shape (n_a, m)
  - **dWax** : Gradients of input-to-hidden weights, of shape (n_a, n_x)
  - **dWaa** : Gradients of hidden-to-hidden weights, of shape (n_a, n_a)
  - **Dba** : Gradients of bias vector, of shape (n_a,1)
- RNN Backward
  - Computing the gradients of the cost with respect to a⟨t⟩a⟨t⟩ at every time-step tt is useful because it is what helps the gradient backpropagate to the previous RNN-cell.
  - To do so, you need to iterate through all the time steps starting at the end, and at each step, you increment the overall dba, dWaa, dWax and you store dxd.
- LSTM Cell Backward
  Implement the backward pass for the LSTM-cell (single time-step).
  Following are the arguments that should be defined by the programmer:
  Arguments:
  - **da_next** : Gradients of next hidden state, of shape (n_a, m)
  - **dc_next** : Gradients of next cell state, of shape (n_a, m)
  - **cache** : cache storing information from the forward pass
  - **dbo** : Gradient w.r.t. biases of the output gate, of shape (n_a, 1)

  Thus, it shows the following results:
  gradients : python dictionary containing:
  - **dxt** : Gradient of input data at time-step t, of shape (n_x, m)
  - **da_prev** : Gradient w.r.t. the previous hidden state, numpy array of shape (n_a, m)
  - **dc_prev** :Gradient w.r.t the previous memory state, of shape (n_a, m, T_x)
  - **dWf** : the weight matrix of the forget gate, numpy array shape of (n_a, n_a + n_x)
  - **dWi** : the weight matrix of the update gate, numpy array shape of (n_a, n_a + n_x)

- **dWc** : the weight matrix of the memory gate, numpy array shape of (n_a, n_a + n_x)
- **dWo** : the weight matrix of the output gate, numpy array shape of (n_a, n_a + n_x)
- **bbf** : Gradient w.r.t biases of the forget gate, numpy array shape of (n_a, 1)
- **bbi** : Gradient w.r.t biases of the update gate, numpy array shape of (n_a, 1)
- **bbc** : Gradient w.r.t biases of the memory gate, numpy array shape of (n_a, 1)
- **bbo** : Gradient w.r.t biases of the output gate, numpy array shape of (n_a, 1)

3.2 Character Level Language Model

- Problem Statement:
  Data Set and Preprocessing
  Read the dataset of dinosaur names, create a list of unique characters (such as a-z), and compute the dataset and vocabulary size. The characters are a-z (26 characters) plus the "\n" (or newline character), which in this assignment plays a role similar to the <EOS> (or "End of sentence") token we had discussed in lecture, only here it indicates the end of the dinosaur name rather than the end of a sentence. We create a python dictionary (i.e., a hash table) to map each character to an index from 0-26. We also create a second python dictionary that maps each index back to the corresponding character character. This will help to figure out what index corresponds to what character in the probability distribution output of the softmax layer. Below, char_to_ix and ix_to_char are the python dictionaries.
  Overview of the model
  model will have the following structure:
  - Initialize parameters
  - Run the optimization loop
    - Forward propagation to compute the loss function
    - Backward propagation to compute the gradients with respect to the loss function
    - Clip the gradients to avoid exploding gradients
    - Using the gradients, update your parameter with the gradient descent update rule.
  - Return the learned parameters

- *Building Blocks of the Model:*
  build two important blocks of the overall model:
  - Gradient clipping: to avoid exploding gradients
  - Sampling: a technique used to generate characters

  then apply these two functions to build the model.
  Clipping The Gradients in the Optimization Loop
  In this section you will implement the clip function that you will call inside of your optimization loop. Recall that your overall loop structure usually consists of a forward pass, a cost computation, a backward pass, and a parameter update. Before updating the parameters, you will perform gradient clipping when needed to make sure that your gradients are not "exploding," meaning taking on overly large values.
  In the exercise below, you will implement a function clip that takes in a dictionary of gradients and returns a clipped version of gradients if needed. There are different ways to clip gradients; we will use a simple element-wise clipping procedure, in which every element of the gradient vector is clipped to lie between some range [-N, N]. More generally, you will provide a maxValue (say 10). In this example, if any component of the gradient vector is greater than 10, it would be set to 10; and if any component of the gradient vector is less than -10, it would be set to -10. If it is between -10 and 10, it is left alone.

  Sampling
  Step 1: Pass the network the first "dummy" input $X^{(1)} = \vec{0}$ (the vector of zeros). This is the default input before we've generated any characters. We also set $X^{(1)} = \vec{0}$
  Step 2: Run one step of forward propagation to get $a^{(1)}$and $\hat{y}^{(1)}$
  Step 3: Carry out sampling: Pick the next character's index according to the probability distribution $\hat{y}^{(t+1)}$

Step 4: The last step to implement in sample() is to overwrite the variable x , which currently stores $x^{(t)}$, with the value of $x^{(t+1)}$.

- *Building the Language Model:*
  It is time to build the character-level language model for text generation.
  Gradient Design,
  In this section you will implement a function performing one step of stochastic gradient descent (with clipped gradients). You will go through the training examples one at a time, so the optimization algorithm will be stochastic gradient descent. As a reminder, here are the steps of a common optimization loop for an RNN:
  - Forward propagate through the RNN to compute the loss
  - Backward propagate through time to compute the gradients of the loss with respect to the parameters
  - Clip the gradients if necessary
  - Update your parameters using gradient descent

  Training the Model
  Given the dataset of dinosaur names, we use each line of the dataset (one name) as one training example. Every 100 steps of stochastic gradient descent, you will sample 10 randomly chosen names to see how the algorithm is doing. Remember to shuffle the dataset, so that stochastic gradient descent visits the examples in random order.

3.3 Stock Market Prediction

- Loading Dataset
  - The dataset that will be used in this tutorial is "NSE-TATAGLOBAL.csv"
  - The next step is to load in our training dataset and select the Open and High columns that we'll use in our modeling.
  - We check the head of our dataset to give us a glimpse into the kind of dataset we're working with.
  - The Open column is the starting price while the Close column is the final price of a stock on a particular trading day. The High and Low columns represent the highest and lowest prices for a certain day.

- Feature Scaling
  - From the experience with deep learning models, we know that we must scale our data for optimal performance.
  - We will use Scikit- Learn's MinMaxScaler and scale our dataset to numbers between zero and one.
- Creating Data with Timestep
  LSTMs expect our data to be in a specific format, usually a 3D array. • We start by creating data in 60 timesteps and converting it into an array using NumPy. • Next, we convert the data into a 3D dimension array with X_train samples , 60 timestamps, and one feature at each step.
- Building the LSTM
  In order to build the LSTM, we need to import a couple of modules from Keras:
  - Sequential for initializing the neural network
  - Dense for adding a densely connected neural network layer
  - LSTM for adding the Long Short-Term Memory layer
  - Dropout for adding dropout layers that prevent overfitting
  - We add the LSTM layer and later add a few Dropout layers to prevent overfitting.
  - We add the LSTM layer with the following arguments:
    - 50 units which is the dimensionality of the output space
    - return_sequences=True which determines whether to return the last output in the output sequence, or the full sequence
    - input_shape as the shape of our training set.

- When defining the Dropout layers, we specify 0.2, meaning that 20% of the layers will be dropped.
- Thereafter, we add the Dense layer that specifies the output of 1 unit.
- After this, we compile our model using the popular Adam optimizer and set the loss as the mean_squarred_error. This will compute the mean of the squared errors.
- Next, we fit the model to run on 100 epochs with a batch size of 32.

- Predicting Future Stock
- After building the model, it is time to test its performance:
- In order to predict future stock prices we need to do a couple of things after loading in the test set:
  - Merge the training set and the test set on the 0 axis.
  - Set the time step as 60 (as seen previously)
  - Use MinMaxScaler to transform the new dataset
  - Reshape the dataset as done previously
  - After making the predictions we use inverse_transform to get back the stock prices in normal readable format.
- Plotting the Result
  - we use Matplotlib to visualize the result of the predicted stock price and the real stock price. we can see that the prediction align with the movement of the real stock price. This clearly shows how powerful LSTMs are for analyzing time series and sequential data.
  - we use Matplotlib to visualize the result of the predicted stock price. we can see that the prediction align with the movement of the real stock price. This clearly shows how powerful LSTMs are for analyzing time series and sequential data.

4 Result and Discussion

4.1 RNN Handson

- Packages

In [1]:
```
import numpy as np
from rnn_utils import *
```

- RNN Cell Forward

In [2]:
```
def rnn_cell_forward(xt, a_prev, parameters):
    Wax = parameters["Wax"]
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
    by = parameters["by"]

    a_next = np.tanh(np.dot(Wax,xt)+np.dot(Waa,a_prev)+ba)
    yt_pred = softmax(np.dot(Wya,a_next)+by)
    cache = (a_next, a_prev, xt, parameters)
    return a_next, yt_pred, cache
```

In [3]:
```
np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
Waa = np.random.randn(5,5)
Wax = np.random.randn(5,3)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)

parameters = {"Waa": Waa, "Wax": Wax, "Wya": Wya, "ba": ba, "by": by}

a_next, yt_pred, cache = rnn_cell_forward(xt, a_prev, parameters)
print("a_next[4]", a_next[4])
print("a_next.shape", a_next.shape)
print("yt_pred[1]", yt_pred[1])
print("yt_pred.shape", yt_pred.shape)
```

Result:

```
a_next[4] [ 0.59584544  0.18141802  0.61311866  0.99808218  0.85016201  0.99980978
 -0.18887155  0.99815551  0.6531151   0.82872037]
a_next.shape (5, 10)
yt_pred[1] [0.9888161  0.01682021 0.21140899 0.36817467 0.98988387 0.88945212
 0.36920224 0.9966312  0.9982559  0.17746526]
yt_pred.shape (2, 10)
```

- RNN Forward

In [4]:
```python
def rnn_forward(x, a0, parameters):
    caches = []
    n_x, m, T_x = x.shape
    n_y, n_a = parameters["Wya"].shape

    a = np.zeros((n_a, m, T_x))
    y_pred = np.zeros((n_y, m, T_x))

    a_next = a0

    for t in range(0, T_x):
        a_next, yt_pred, cache = rnn_cell_forward(x[:,:,t], a_next, parameters)
        a[:,:,t] = a_next
        y_pred[:,:,t] = yt_pred
        caches.append(cache)
    caches = (caches, x)
    return a, y_pred, caches
```

In [5]:
```python
np.random.seed(1)
x = np.random.randn(3, 10, 4)
a0 = np.random.randn(5, 10)
Waa = np.random.randn(5, 5)
Wax = np.random.randn(5, 3)
Wya = np.random.randn(2, 5)
ba = np.random.randn(5, 1)
by = np.random.randn(2, 1)
parameters = {"Waa":Waa, "Wax":Wax, "Wya":Wya, "ba":ba, "by":by}

a, y_pred, caches = rnn_forward(x, a0, parameters)
print("a[4][1]", a[4][1])
print("a.shape", a.shape)
print("y_pred[1][3]", y_pred[1][3])
print("y_pred.shape", y_pred.shape)
print("caches[1][1][3]", caches[1][1][3])
print("len(caches) = ", len(caches))
```

Result:

```
a[4][1] [-0.99999375  0.77911235 -0.99861469 -0.99833267]
a.shape (5, 10, 4)
y_pred[1][3] [0.79560373 0.86224861 0.11118257 0.81515947]
y_pred.shape (2, 10, 4)
caches[1][1][3] [-1.1425182  -0.34934272 -0.20889423  0.58662319]
len(caches) =  2
```

- LSTM Cell Forward

In [6]:
```python
def lstm_cell_forward(xt, a_prev, c_prev, parameters):
    Wf = parameters["Wf"]
    bf = parameters["bf"]
    Wi = parameters["Wi"]
    bi = parameters["bi"]
    Wc = parameters["Wc"]
    bc = parameters["bc"]
    Wo = parameters["Wo"]
    bo = parameters["bo"]
    Wy = parameters["Wy"]
    by = parameters["by"]

    n_x, m = xt.shape
    n_y, n_a = Wy.shape

    concat = np.zeros((n_x + n_a, m))
    concat[: n_a, :] = a_prev
    concat[n_a :, :] = xt

    ft = sigmoid(np.dot(Wf,concat)+bf)
    it = sigmoid(np.dot(Wi,concat)+bi)
    cct = np.tanh(np.dot(Wc,concat)+bc)
    c_next = c_prev*ft + it*cct
    ot = sigmoid(np.dot(Wo,concat)+bo)
    a_next = ot*np.tanh(c_next)

    yt_pred = softmax(np.dot(Wy, a_next)+by)

    cache = (a_next, c_next, a_prev, c_prev, ft, it, cct, ot, xt, parameters)
    return a_next, c_next, yt_pred, cache
```

In [7]:
```
np.random.seed(1)
xt = np.random.randn(3, 10)
a_prev = np.random.randn(5, 10)
c_prev = np.random.randn(5, 10)
Wf = np.random.randn(5, 5+3)
bf = np.random.randn(5, 1)
Wi = np.random.randn(5, 5+3)
bi = np.random.randn(5, 1)
Wo = np.random.randn(5, 5+3)
bo = np.random.randn(5, 1)
Wc = np.random.randn(5, 5+3)
bc = np.random.randn(5, 1)
Wy = np.random.randn(2, 5)
by = np.random.randn(2, 1)

parameters = {"Wf": Wf, "Wi": Wi, "Wo":Wo, "Wc": Wc, "Wy":Wy, "bf":bf, "bi":bi, "bo":bo, "bc": bc, "by":by}

a_next, c_next, yt, cache = lstm_cell_forward(xt, a_prev, c_prev, parameters)
print("a_next[4]", a_next[4])
print("a_next.shape", a_next.shape)
print("c_next[2]", c_next[2])
print("c_next.shape", c_next.shape)
print("yt[1]", yt[1])
print("yt.shape", yt.shape)
print("cache[1][3]", cache[1][3])
print("len(cache) = ", len(cache))
```

Result:

```
a_next[4] [-0.66408471  0.0036921   0.02088357  0.22834167 -0.85575339  0.00138482
  0.76566531  0.34631421 -0.00215674  0.43827275]
a_next.shape (5, 10)
c_next[2] [ 0.63267805  1.00570849  0.35504474  0.20690913 -1.64566718  0.11832942
  0.76449811 -0.0981561  -0.74348425 -0.26810932]
c_next.shape (5, 10)
yt[1] [0.79913913 0.15986619 0.22412122 0.15606108 0.97057211 0.31146381
 0.00943007 0.12666353 0.39380172 0.07828381]
yt.shape (2, 10)
cache[1][3] [-0.16263996  1.03729328  0.72938082 -0.54101719  0.02752074 -0.30821874
  0.07651101 -1.03752894  1.41219977 -0.37647422]
len(cache) =  10
```

- LSTM Forward

In [8]:
```
def lstm_forward(x, a0, parameters):
    caches = []

    n_x, m, T_x = x.shape
    n_y, n_a = parameters["Wy"].shape

    a = np.zeros((n_a, m, T_x))
    c = np.zeros((n_a, m, T_x))
    y = np.zeros((n_y, m, T_x))

    a_next = a0
    c_next = np.zeros((n_a, m))

    for t in range(0, T_x):
        a_next, c_next, yt, cache = lstm_cell_forward(x[:,:,t], a_next, c_next, parameters)
        a[:,:,t] = a_next
        y[:,:,t] = yt
        c[:,:,t] = c_next
        caches.append(cache)

    caches = (caches, x)

    return a, y, c, caches
```

In [9]:
```
np.random.seed(1)
x = np.random.randn(3, 10, 7)
a0 = np.random.randn(5, 10)
Wf = np.random.randn(5, 5+3)
bf = np.random.randn(5, 1)
Wi = np.random.randn(5, 5+3)
bi = np.random.randn(5, 1)
Wo = np.random.randn(5, 5+3)
bo = np.random.randn(5, 1)
Wc = np.random.randn(5, 5+3)
bc = np.random.randn(5, 1)
Wy = np.random.randn(2, 5)
by = np.random.randn(2, 1)

parameters = {"Wf": Wf, "Wi": Wi, "Wo":Wo, "Wc": Wc, "Wy":Wy, "bf":bf, "bi":bi, "bo":bo, "bc": bc, "by":by}

a, y, c, caches = lstm_forward(x, a0, parameters)
print("a[4][3][6]", a[4][3][6])
print("a.shape", a.shape)
print("y[1][4][3]", y[1][4][3])
print("y.shape", y.shape)
print("caches[1][1][1]", caches[1][1][1])
print("c[1][2][1]", c[1][2][1])
print("len(caches) = ", len(caches))
```

Result:

```
a[4][3][6] 0.17211776753291672
a.shape (5, 10, 7)
y[1][4][3] 0.9508734618501101
y.shape (2, 10, 7)
caches[1][1][1] [ 0.82797464  0.23009474  0.76201118 -0.22232814 -0.20075807  0.18656139
  0.41005165]
c[1][2][1] -0.8555449167181982
len(caches) =  2
```

- RNN Cell Backward

In [10]:
```
def rnn_cell_backward(da_next, cache):
    (a_next, a_prev, xt, parameters) = cache

    Wax = parameters["Wax"]
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
    by = parameters["by"]

    dtanh = 1 - np.power(a_next, 2)
    dxt = np.dot(Wax.T, da_next*dtanh)
    dWax = np.dot(da_next*dtanh, xt.T)

    da_prev = np.dot(Waa.T, da_next*dtanh)
    dWaa = np.dot(da_next*dtanh, a_prev.T)

    dba = np.sum(da_next*dtanh, axis=1, keepdims=True)
    gradients = {"dxt":dxt, "da_prev":da_prev, "dWax": dWax, "dWaa": dWaa, "dba":dba}

    return gradients
```

In [11]:
```
np.random.seed(1)
xt = np.random.randn(3, 10)
a_prev = np.random.randn(5, 10)
Wax = np.random.randn(5, 3)
Waa = np.random.randn(5, 5)
Wya = np.random.randn(2, 5)
b = np.random.randn(5, 1)
by = np.random.randn(2, 1)

parameters = {"Wax": Wax, "Waa": Waa, "Wya":Wya, "ba":ba, "by":by}

a_next, yt, cache = rnn_cell_forward(xt, a_prev, parameters)

da_next = np.random.randn(5,10)
gradients = rnn_cell_backward(da_next, cache)

print("gradients[\"dxt\"][1][2]", gradients["dxt"][1][2])
print("gradients[\"dxt\"].shape", gradients["dxt"].shape)
print("gradients[\"da_prev\"][2][3]", gradients["da_prev"][2][3])
print("gradients[\"da_prev\"].shape", gradients["da_prev"].shape)
print("gradients[\"dWax\"][3][1]", gradients["dWax"][3][1])
print("gradients[\"dWax\"].shape", gradients["dWax"].shape)
print("gradients[\"dWaa\"][1][2]", gradients["dWaa"][1][2])
print("gradients[\"dWaa\"].shape", gradients["dWaa"].shape)
print("gradients[\"dba\"][4]", gradients["dba"][4])
print("gradients[\"dba\"].shape", gradients["dba"].shape)
```

Result:

```
gradients["dxt"][1][2] -0.4605641030588796
gradients["dxt"].shape (3, 10)
gradients["da_prev"][2][3] 0.08429686538067718
gradients["da_prev"].shape (5, 10)
gradients["dWax"][3][1] 0.3930818739219304
gradients["dWax"].shape (5, 3)
gradients["dWaa"][1][2] -0.2848395578696067
gradients["dWaa"].shape (5, 5)
gradients["dba"][4] [0.80517166]
gradients["dba"].shape (5, 1)
```

- RNN Backward

In [12]:
```
def rnn_backward(da, caches):
    (caches, x) = caches
    (a1, a0, x1, parameters) = caches[1]
    n_a, m, T_x = da.shape
    n_x, m = x1.shape

    dx = np.zeros((n_x, m, T_x))
    dWax = np.zeros((n_a, n_x))
    dWaa = np.zeros((n_a, n_a))
    dba = np.zeros((n_a, 1))
    da0 = np.zeros((n_a, m))
    da_prevt = np.zeros((n_a, m))

    for t in reversed(range(0, T_x)):
        gradients = rnn_cell_backward(da[:,:,t] + da_prevt, caches[t])
        dxt, da_prevt, dWaxt, dWaat, dbat = gradients['dxt'], gradients['da_prev'], gradients['dWax'], gradients['dW
        dx[:, :, t] = dxt
        dWax += dWaxt
        dWaa += dWaat
        dba += dbat

    da0 = da_prevt

    gradients = {"dx": dx, "da0": da0, "dWax": dWax, "dWaa": dWaa, "dba": dba}

    return gradients
```

In [13]:
```
np.random.seed(1)
x = np.random.randn(3, 10, 4)
a0 = np.random.randn(5, 10)
Wax = np.random.randn(5, 3)
Waa = np.random.randn(5, 5)
Wya = np.random.randn(2, 5)
ba = np.random.randn(5, 1)
by = np.random.randn(2, 1)

parameters = {"Wax": Wax, "Waa": Waa, "Wya":Wya, "ba":ba, "by":by}

a, y, caches = rnn_forward(x, a0, parameters)

da = np.random.randn(5, 10, 4)
gradients = rnn_backward(da, caches)

print("gradients[\"dx\"][1][2]", gradients["dx"][1][2])
print("gradients[\"dx\"].shape", gradients["dx"].shape)
print("gradients[\"da0\"][2][3]", gradients["da0"][2][3])
print("gradients[\"da0\"].shape", gradients["da0"].shape)
print("gradients[\"dWax\"][3][1]", gradients["dWax"][3][1])
print("gradients[\"dWax\"].shape", gradients["dWax"].shape)
print("gradients[\"dWaa\"][1][2]", gradients["dWaa"][1][2])
print("gradients[\"dWaa\"].shape", gradients["dWaa"].shape)
print("gradients[\"dba\"][4]", gradients["dba"][4])
print("gradients[\"dba\"].shape", gradients["dba"].shape)
```

Result:

```
gradients["dx"][1][2] [-2.07101689 -0.59255627  0.02466855  0.01483317]
gradients["dx"].shape (3, 10, 4)
gradients["da0"][2][3] -0.31494237512664996
gradients["da0"].shape (5, 10)
gradients["dWax"][3][1] 11.264104496527777
gradients["dWax"].shape (5, 3)
gradients["dWaa"][1][2] 2.303333126579893
gradients["dWaa"].shape (5, 5)
gradients["dba"][4] [-0.74747722]
gradients["dba"].shape (5, 1)
```

- LSTM Cell Backward

In [16]:
```
def lstm_cell_backward(da_next, dc_next, cache):
    (a_next, c_next, a_prev, c_prev, ft, it, cct, ot, xt, parameters) = cache

    n_x, m = xt.shape
    n_a, m = a_next.shape

    dot = da_next*np.tanh(c_next)
    dcct = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*it
    dit = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*cct
    dft = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*c_prev

    dit = dit*it*(1-it)
    dft = dft*ft*(1-ft)
    dot = dot*ot*(1-ot)
    dcct = dcct*(1-np.power(cct, 2))

    concat = np.zeros((n_x + n_a, m))
    concat[: n_a, :] = a_prev
    concat[n_a :, :] = xt

    dWf = np.dot(dft, concat.T)
    dWi = np.dot(dit, concat.T)
    dWc = np.dot(dcct, concat.T)
    dWo = np.dot(dot, concat.T)
    dbf = np.sum(dft, axis=1, keepdims=True)
    dbi = np.sum(dit, axis=1, keepdims=True)
    dbc = np.sum(dcct, axis=1, keepdims=True)
    dbo = np.sum(dot, axis=1, keepdims=True)

    da_prevx = np.dot(parameters['Wf'].T, dft) + np.dot(parameters['Wo'].T, dft) + np.dot(parameters['Wi'].T, dft)
    + np.dot(parameters['Wc'].T, dft)
    da_prev = da_prevx[: n_a, :]
    dc_prev = (da_next*ot*(1-np.power(np.tanh(c_next), 2))+dc_next)*ft
    dxt = da_prevx[n_a :, :]

    gradients = {"dxt": dxt, "da_prev": da_prev, "dc_prev": dc_prev,
                "dWf": dWf, "dbf": dbf, "dWi": dWi, "dbi": dbi,
                "dWc": dWc, "dbc": dbc, "dWo": dWo, "dbo": dbo}

    return gradients
```

In [17]:
```
np.random.seed(1)
xt = np.random.randn(3, 10)
a_prev = np.random.randn(5, 10)
c_prev = np.random.randn(5, 10)
Wf = np.random.randn(5, 5+3)
bf = np.random.randn(5, 1)
Wi = np.random.randn(5, 5+3)
bi = np.random.randn(5, 1)
Wo = np.random.randn(5, 5+3)
bo = np.random.randn(5, 1)
Wc = np.random.randn(5, 5+3)
bc = np.random.randn(5, 1)
Wy = np.random.randn(2, 5)
by = np.random.randn(2, 1)


parameters = {"Wf": Wf, "Wi": Wi, "Wo":Wo, "Wc":Wc, "Wy":Wy,
              "bf": bf, "bi": bi, "bo": bo, "bc": bc, "by": by}

a_next, c_next, yt, cache = lstm_cell_forward(xt, a_prev, c_prev, parameters)

da_next = np.random.randn(5, 10)
dc_next = np.random.randn(5, 10)

gradients = lstm_cell_backward(da_next, dc_next, cache)

print("gradients[\"dxt\"][1][2]", gradients["dxt"][1][2])
print("gradients[\"dxt\"].shape", gradients["dxt"].shape)
print("gradients[\"da_prev\"][2][3]", gradients["da_prev"][2][3])
print("gradients[\"da_prev\"].shape", gradients["da_prev"].shape)
print("gradients[\"dc_prev\"][2][3]", gradients["dc_prev"][2][3])
print("gradients[\"dc_prev\"].shape", gradients["dc_prev"].shape)
print("gradients[\"dWf\"][3][1]", gradients["dWf"][3][1])
print("gradients[\"dWf\"].shape", gradients["dWf"].shape)
print("gradients[\"dWi\"][1][2]", gradients["dWi"][1][2])
print("gradients[\"dWi\"].shape", gradients["dWi"].shape)
print("gradients[\"dWc\"][3][1]", gradients["dWc"][3][1])
print("gradients[\"dWc\"].shape", gradients["dWc"].shape)
print("gradients[\"dWo\"][1][2]", gradients["dWo"][1][2])
print("gradients[\"dWo\"].shape", gradients["dWo"].shape)
print("gradients[\"dbf\"][4]", gradients["dbf"][4])
print("gradients[\"dbf\"].shape", gradients["dbf"].shape)
print("gradients[\"dbi\"][4]", gradients["dbi"][4])
print("gradients[\"dbi\"].shape", gradients["dbi"].shape)
print("gradients[\"dbc\"][4]", gradients["dbc"][4])
print("gradients[\"dbc\"].shape", gradients["dbc"].shape)
print("gradients[\"dbo\"][4]", gradients["dbo"][4])
print("gradients[\"dbo\"].shape", gradients["dbo"].shape)
```

Result:

```
gradients["dxt"][1][2] -0.0005948426375572136
gradients["dxt"].shape (3, 10)
gradients["da_prev"][2][3] -0.5867025381982609
gradients["da_prev"].shape (5, 10)
gradients["dc_prev"][2][3] 0.7975220387970015
gradients["dc_prev"].shape (5, 10)
gradients["dWf"][3][1] -0.1479548381644968
gradients["dWf"].shape (5, 8)
gradients["dWi"][1][2] 1.0574980552259903
gradients["dWi"].shape (5, 8)
gradients["dWc"][3][1] 2.304562163687667
gradients["dWc"].shape (5, 8)
gradients["dWo"][1][2] 0.3313115952892109
gradients["dWo"].shape (5, 8)
gradients["dbf"][4] [0.18864637]
gradients["dbf"].shape (5, 1)
gradients["dbi"][4] [-0.40142491]
gradients["dbi"].shape (5, 1)
gradients["dbc"][4] [0.25587763]
gradients["dbc"].shape (5, 1)
gradients["dbo"][4] [0.13893342]
gradients["dbo"].shape (5, 1)
```

4.2 Character Level Language Model

```
In [4]: import numpy as np
        from utils import *
        import random
```

- Problem Statement:

  Data Set and Preprocessing

```
In [7]: data = open('dinos.txt', 'r').read()
        data= data.lower()
        chars = list(set(data))
        data_size, vocab_size = len(data), len(chars)
        print('There are %d total characters and %d unique characters in your data.'
        % (data_size, vocab_size))
```

  Hasil

```
There are 19909 total characters and 27 unique characters in your data.
```

```
In [8]: char_to_ix = { ch:i for i,ch in enumerate(sorted(chars)) }
        ix_to_char = { i:ch for i,ch in enumerate(sorted(chars)) }
        print(ix_to_char)
```

  Result

```
{0: '\n', 1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 6: 'f', 7: 'g', 8: 'h', 9: 'i', 10: 'j', 11: 'k', 12: 'l', 13: 'm
', 14: 'n', 15: 'o', 16: 'p', 17: 'q', 18: 'r', 19: 's', 20: 't', 21: 'u', 22: 'v', 23: 'w', 24: 'x', 25: 'y', 26:
'z'}
```

  Overview of the model

- Building Blocks of the Model:

  Clipping The Gradients in the Optimization Loop

```
In [16]: def clip(gradients, maxValue):
             dWaa, dWax, dWya, db, dby = gradients['dWaa'], gradients['dWax'], gradients['dWya'], gradients['db'], gradients['
             for gradient in [dWax, dWaa, dWya, db, dby]:
                 np.clip(gradient, -maxValue, maxValue, gradient)
             gradients = {"dWaa": dWaa, "dWax": dWax, "dWya": dWya, "db": db, "dby": dby}
             return gradients
```

```
In [17]: dWax = np.random.randn(5,3)*10
         dWaa = np.random.randn(5,5)*10
         dWya = np.random.randn(2,5)*10
         db = np.random.randn(5,1)*10
         dby = np.random.randn(2,1)*10
         gradients = {"dWax": dWax, "dWaa": dWaa, "dWya": dWya, "db": db, "dby": dby}
         gradients = clip(gradients, 10)
         print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
         print("gradients[\"dWax\"][3][1] =", gradients["dWax"][3][1])
         print("gradients[\"dWya\"][1][2] =", gradients["dWya"][1][2])
         print("gradients[\"db\"][4] =", gradients["db"][4])
         print("gradients[\"dby\"][1] =", gradients["dby"][1])
```

  Result

```
gradients["dWaa"][1][2] = 2.429412595639908
gradients["dWax"][3][1] = 2.578222094355658
gradients["dWya"][1][2] = -7.789418565738093
gradients["db"][4] = [3.83272006]
gradients["dby"][1] = [-10.]
```

Sampling

In [27]:
```
def sample(parameters, char_to_ix, seed):
    Waa, Wax, Wya, by, b = parameters['Waa'], parameters['Wax'], parameters['Wya'], parameters['by'], parameters['b'
    vocab_size = by.shape[0]
    n_a = Waa.shape[1]
    n_x = Wax.shape[1]
    x = np.zeros((n_x, 1))
    a_prev = np.zeros((n_a, 1))
    indices = []
    idx = -1
    counter = 0
    newline_character = char_to_ix['\n']
    while (idx != newline_character and counter != 50):

        a = np.tanh(np.dot(Wax,x)+np.dot(Waa,a_prev)+b)
        z = np.dot(Wya,a)+by
        y = softmax(z)

        np.random.seed(counter+seed)
        idx = np.random.choice(range(0, vocab_size), p=y.ravel())
        indices.append(idx)

        x = np.zeros((vocab_size,1))
        x[idx] = 1
        a_prev = a

        seed += 1
        counter += 1
    if (counter == 50): indices.append(char_to_ix['\n'])
    return indices
```

In [29]:
```
np.random.seed(2)
_, n_a = 20, 100
Wax, Waa, Wya = np.random.randn(n_a, vocab_size), np.random.randn(n_a, n_a),np.random.randn(vocab_size, n_a)
b, by = np.random.randn(n_a, 1), np.random.randn(vocab_size, 1)
parameters = {"Wax": Wax, "Waa": Waa, "Wya": Wya, "b": b, "by": by}
indices = sample(parameters, char_to_ix, 0)
print("Sampling:")
print("list of sampled indices:", indices)
print("list of sampled characters:", [ix_to_char[i] for i in indices])
```

Result

```
Sampling:
list of sampled indices: [12, 17, 24, 14, 13, 9, 10, 22, 24, 6, 13, 11, 12, 6, 21, 15, 21, 14, 3, 2, 1, 21, 18, 24,
7, 25, 6, 25, 18, 10, 16, 2, 3, 8, 15, 12, 11, 7, 1, 12, 10, 2, 7, 7, 0]
list of sampled characters: ['l', 'q', 'x', 'n', 'm', 'i', 'j', 'v', 'x', 'f', 'm', 'k', 'l', 'f', 'u', 'o', 'u', '
n', 'c', 'b', 'a', 'u', 'r', 'x', 'g', 'y', 'f', 'y', 'r', 'j', 'p', 'b', 'c', 'h', 'o', 'l', 'k', 'g', 'a', 'l', '
j', 'b', 'g', 'g', '\n']
```

- Building the Language Model:
  Gradient Design,

In [8]:
```
def optimize(X, Y, a_prev, parameters, learning_rate = 0.01):
    loss, cache = rnn_forward(X, Y, a_prev, parameters)
    gradients, a = rnn_backward(X, Y, parameters, cache)
    gradients = clip(gradients, 5)
    parameters = update_parameters(parameters, gradients, learning_rate)
    return loss, gradients, a[len(X)-1]
```

In [9]:
```
np.random.seed(1)
vocab_size, n_a = 27, 100
a_prev = np.random.randn(n_a, 1)
Wax, Waa, Wya = np.random.randn(n_a, vocab_size), np.random.randn(n_a, n_a),np.random.randn(vocab_size, n_a)
b, by = np.random.randn(n_a, 1), np.random.randn(vocab_size, 1)
parameters = {"Wax": Wax, "Waa": Waa, "Wya": Wya, "b": b, "by": by}
X = [12,3,5,11,22,3]
Y = [4,14,11,22,25, 26]
loss, gradients, a_last = optimize(X, Y, a_prev, parameters, learning_rate = 0.01)
print("Loss =", loss)
print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
print("np.argmax(gradients[\"dWax\"]) =", np.argmax(gradients["dWax"]))
print("gradients[\"dWya\"][1][2] =", gradients["dWya"][1][2])
print("gradients[\"db\"][4] =", gradients["db"][4])
print("gradients[\"dby\"][1] =", gradients["dby"][1])
print("a_last[4] =", a_last[4])
```

Result:

```
Loss = 126.5039757216538
gradients["dWaa"][1][2] = 0.19470931534716257
np.argmax(gradients["dWax"]) = 93
gradients["dWya"][1][2] = -0.007773876032002955
gradients["db"][4] = [-0.06809825]
gradients["dby"][1] = [0.01538192]
a_last[4] = [-1.]
```

Training the Model

```
In [10]: def model(data, ix_to_char, char_to_ix, num_iterations = 35000, n_a = 50, dino_names = 7, vocab_size = 27):
    n_x, n_y = vocab_size, vocab_size
    parameters = initialize_parameters(n_a, n_x, n_y)
    loss = get_initial_loss(vocab_size, dino_names)
    with open("dinos.txt") as f: examples = f.readlines()
    examples = [x.lower().strip() for x in examples]

    np.random.seed(0)
    np.random.shuffle(examples)

    a_prev = np.zeros((n_a, 1))
    for j in range(num_iterations):

        index = j % len(examples)
        X = [None] + [char_to_ix[ch] for ch in examples[index]]
        Y = X[1:] + [char_to_ix["\n"]]

        curr_loss, gradients, a_prev = optimize(X, Y, a_prev, parameters,
        learning_rate = 0.01)
        loss = smooth(loss, curr_loss)

        if j % 2000 == 0:
            print('Iteration: %d, Loss: %f' % (j, loss) + '\n')

            seed = 0
            for name in range(dino_names):

                sampled_indices = sample(parameters, char_to_ix, seed)
                print_sample(sampled_indices, ix_to_char)
                seed += 1
            print('\n')
    return parameters
```

Result:

```
In [11]: parameters = model(data, ix_to_char, char_to_ix)
Iteration: 0, Loss: 23.087336

Nkzxwtdmfqoeyhsqwasjkjvu
Kneb
Kzxwtdmfqoeyhsqwasjkjvu
Neb
Zxwtdmfqoeyhsqwasjkjvu
Eb
Xwtdmfqoeyhsqwasjkjvu


Iteration: 2000, Loss: 27.884160

Liusskeomnolxeros
Hmdaairus
Hytroligoraurus
Lecalosapaus
Xusicikoraurus
Abalpsamantisaurus
```

4.3 Stock Market Prediction

- Introduction

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pandas as pd
```

- Loading Dataset

In [2]:
```
dataset_train = pd.read_csv('NSE-TATAGLOBAL.csv')
training_set = dataset_train.iloc[:, 1:2].values

dataset_train.head()
```

Result:

Out[2]:

	Date	Open	High	Low	Last	Close	Total Trade Quantity	Turnover (Lacs)
0	2018-09-28	234.05	235.95	230.20	233.50	233.75	3069914	7162.35
1	2018-09-27	234.55	236.80	231.10	233.80	233.25	5082859	11859.95
2	2018-09-26	240.00	240.00	232.50	235.00	234.25	2240909	5248.60
3	2018-09-25	233.30	236.75	232.00	236.25	236.10	2349368	5503.90
4	2018-09-24	233.55	239.20	230.75	234.00	233.30	3423509	7999.55

- Feature Scaling

In [3]:
```
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0,1))

training_set_scaled = sc.fit_transform(training_set)
```

- Creating Data with Timestep

In [4]:
```
X_train = []
y_train = []
for i in range(60, 2035):
    X_train.append(training_set_scaled[i-60:i, 0])
    y_train.append(training_set_scaled[i, 0])
X_train, y_train = np.array(X_train), np.array(y_train)

X_train = np.reshape(X_train, (y_train.shape[0], X_train.shape[1], 1))
```

- Building the LSTM

In [5]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
```

```
Using TensorFlow backend.
```

In [6]:
```
regressor = Sequential()

regressor.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1], 1)))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 50))
regressor.add(Dropout(0.2))

regressor.add(Dense(units = 1))

regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')

regressor.fit(X_train, y_train, epochs = 100, batch_size = 32)
```

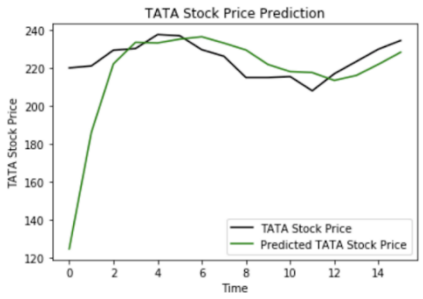
- Predicting Future Stock

In [7]:
```
dataset_test = pd.read_csv('tatatest.csv')
real_stock_price = dataset_test.iloc[:, 1:2].values
dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis = 0)
inputs = dataset_total[len(dataset_total) - len(dataset_test) - 60:].values
inputs = inputs.reshape(-1,1)
inputs = sc.transform(inputs)

X_test = []

for i in range(60, 76):
    X_test.append(inputs[i-60:i, 0])

X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

- Plotting the Result

```
In [8]: plt.plot(real_stock_price, color = 'black', label = 'TATA Stock Price')
        plt.plot(predicted_stock_price, color = 'green', label = 'Predicted TATA Stock Price')
        plt.title('TATA Stock Price Prediction')
        plt.xlabel('Time')
        plt.ylabel('TATA Stock Price')
        plt.legend()
        plt.show()
```

Result



4.4 Stock Market Prediction - APPLE Company

- Introduction

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pandas as pd
```

- Loading Dataset

```
In [2]: dataset_train = pd.read_csv('AAPL.csv')
        training_set = dataset_train.iloc[:, 1:2].values

        dataset_train.head()
```

Result

Out[2]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2019-10-25	243.160004	246.729996	242.880005	246.580002	245.841919	18369300
1	2019-10-24	244.509995	244.800003	241.809998	243.580002	242.850891	17318800
2	2019-10-23	242.100006	243.240005	241.220001	243.179993	242.452087	18957200
3	2019-10-22	241.160004	242.199997	239.619995	239.960007	239.241745	20573400
4	2019-10-21	237.520004	240.990005	237.320007	240.509995	239.790085	21811800

- Feature Scaling

```
In [3]: from sklearn.preprocessing import MinMaxScaler
        sc = MinMaxScaler(feature_range = (0,1))

        training_set_scaled = sc.fit_transform(training_set)
```

- Creating Data with Timestep

```
In [4]: X_train = []
        y_train = []
        for i in range(60, 2049):
            X_train.append(training_set_scaled[i-60:i, 0])
            y_train.append(training_set_scaled[i, 0])
        X_train, y_train = np.array(X_train), np.array(y_train)

        X_train = np.reshape(X_train, (y_train.shape[0], X_train.shape[1], 1))
```

- Building the LSTM

In [5]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
```

```
Using TensorFlow backend.
```

In [6]:
```
regressor = Sequential()

regressor.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1], 1)))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 50, return_sequences  = True))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 50))
regressor.add(Dropout(0.2))

regressor.add(Dense(units = 1))

regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')

regressor.fit(X_train, y_train, epochs = 100, batch_size = 32)
```

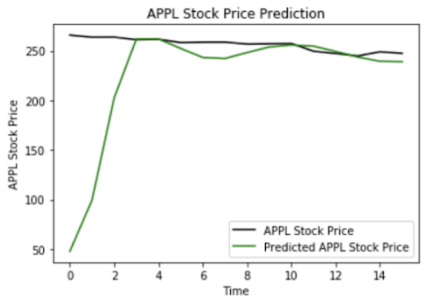
- Predicting Future Stock

In [7]:
```
dataset_test = pd.read_csv('AAPL-TEST.csv')
real_stock_price = dataset_test.iloc[:, 1:2].values
dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis = 0)
inputs = dataset_total[len(dataset_total) - len(dataset_test) - 60:].values
inputs = inputs.reshape(-1,1)
inputs = sc.transform(inputs)

X_test = []

for i in range(60, 76):
    X_test.append(inputs[i-60:i, 0])

X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

- Plotting the Result

In [8]:
```
plt.plot(real_stock_price, color = 'black', label = 'APPL Stock Price')
plt.plot(predicted_stock_price, color = 'green', label = 'Predicted APPL Stock Price')
plt.title('APPL Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('APPL Stock Price')
plt.legend()
plt.show()
```

Result



5 Conclusion

To conclude, this writing is showing that the RNN can be compatible to analyze the sequential data pattern. RNN continue to evolve and be implemented in areas such as speech recognition, music generation, machine translation, etc. The development of RNN is not only applied but also in method optimization. LSTM is one of the developments in RNN, where LSTM is fit to the longer arranged data pattern to be analyzed.

References

1. https://www.academia.edu/34438387/PENGENALAN_LSTM_LONG_SHORT_TERM_MEMORY_
2. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
3. Putra, Muhammad Wildan. "Analisis Dan Implementasi Long Short Term Memory Neural Network Untuk Prediksi Harga Bitcoin." e-Proceeding of Engineering, August 2, 2018.