COMP 5327 ADVANCED ALGORITHMS

FINAL PROJECT REPORT

Algorithmic runtime complexity improvement by the recurrent neural network.

Name of the team: THE MARVEL KNIGHT

Recurrent neural networks (RNNs) are a class of neural networks that are suited to processing time-series data and other sequential data. Since RNNs deal with sequential data, in this project we tried to improve runtime complexity by using recurrent neural networks(RNN). **The main purpose of this project is to convert O(N^2) to O(N) runtime complexity by using RNN train modeling.**

Here are the steps that we followed to train the model:

- Loading the necessary libraries.
- Collecting the data required to train and test the model.
- Build LSTM (**Long short-term memory**) model and Dense layers.
- Defining the variables.
- Defining the model
- Training the model

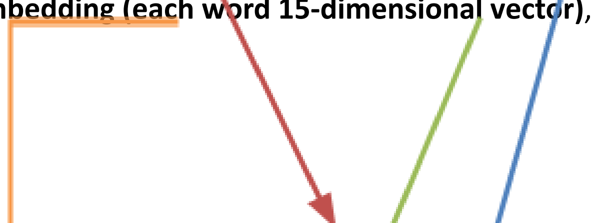
Loading the necessary libraries: For this project we used several Python Deep learning libraries such as **Keras Python Library** (to define and train neural network), **TensorFlow Library** (to train and run recurrent neural networks, sequence-to-sequence models for machine translation), Numpy, Pandas etc.

Data Collection: To train and test the model we collected our input and output data sets from the **www.leetcode.com** and **www.hackerrank.com**. We found 25 different problems and solved them with O(N^2) and O(N) time complexity. We created two different folders to store O(N^2) and O(N) data separately.

Building RNN, Features and Labels: There are some steps to be able to implement Recurrent Neural Network for text generation. We provided a sequence of words to train a Deep Learning model for next word prediction using Python. we need to have the knowledge about the previous data in order to predict the next output. For example, in order to predict the next word of a sentence, we should know its previous words. RNN comes into play in these kinds of situations. RNN is a kind of neuron cell, which has the ability to retain information about the sequence. We use the Tensor flow and Keras library in Python for the next word prediction model. We used Keras Sequential API which means we build the network up one layer at a time.

1. Implement the calculations needed for one time-step of the RNN.
2. Implement a loop over time-steps in order to process all the inputs, one at a time.

We used 500 words as features with the 501st as the label after that used the 2-501 as features and predicted the 52nd and so on. **We have 709,749 sequences each with 500 tokens.** Recurrent neural networks are able to train effectively when the labels are encoded. We encoded the labels using Numpy. After getting all of our features and labels properly formatted, we split them into a training and validation set. We have created some layers such as **Embedding (each word 15-dimensional vector)**, Masking Layer (this for masking any words with



no embedding. We assigned all as zero.), LSTM (for vanishing the gradient problems). We used the **Adam optimizer** to compile our model.

##Setting features for Embedding

```
In [147]:  n_features = 500 + 1
           n_steps_in = 15
           n_steps_out = 15
```

```
In [153]:  model, infenc, infdec = AutoEncoder(n_features, n_features, 128)
           model.summary()

           Model: "model_28"
           ______________________________________________________________________________________
           Layer (type)                   Output Shape          Param #     Connected to
           ======================================================================================
           input_37 (InputLayer)          (None, None, 501)     0
           ______________________________________________________________________________________
           input_38 (InputLayer)          (None, None, 501)     0
           ______________________________________________________________________________________
           lstm_19 (LSTM)                 [(None, 128), (None,  322560      input_37[0][0]
           ______________________________________________________________________________________
           lstm_20 (LSTM)                 [(None, None, 128),   322560      input_38[0][0]
                                                                            lstm_19[0][1]
                                                                            lstm_19[0][2]
           ______________________________________________________________________________________
           dense_10 (Dense)               (None, None, 501)     64629       lstm_20[0][0]
           ======================================================================================
           Total params: 709,749
           Trainable params: 709,749
           Non-trainable params: 0
           ______________________________________________________________________________________
```

##Importing Libraries

```
In [146]:  from keras.preprocessing.text import Tokenizer
           from keras.preprocessing.sequence import pad_sequences
           from keras.models import Sequential
           from keras import layers
           from sklearn.model_selection import train_test_split
           from sklearn.metrics import confusion_matrix
           import pandas as pd
           from random import randint
           from numpy import array
           from numpy import argmax
           from tensorflow.keras.callbacks import EarlyStopping
           from random import randint
           from numpy import array
           from numpy import argmax
           from numpy import array_equal
           from keras.utils import to_categorical
           from keras.models import Model
           from keras.layers import Input
           from keras.layers import LSTM
           from keras.layers import Dense
           from numpy import array_equal
           from keras.utils import to_categorical
           from keras.models import Model
           from keras.layers import Input
           from keras.layers import LSTM
           from keras.layers import Dense
           from keras.optimizers import Adam
           import nltk
           import os
           import numpy as np

           ##nltk.download('punkt')
           ##nltk.download('averaged_perceptron_tagger')
```

Training the Model & Testing: Model that can be trained given source, target, and shifted target. The model is trained on a given source and target sequence where the model takes both the source and a shifted version of the target sequence as input and predicts the whole target sequence. After all we tested and trained our model and were able to get 99 % accuracy at the end of the testing. There are times we were able to get 100% accuracy.

```
Epoch 48/50
158/158 [==============================] - 4s 25ms/step - loss: 0.0741 - accuracy: 0.997
3 - val_loss: 0.0950 - val_accuracy: 0.9973
Epoch 49/50
158/158 [==============================] - 4s 27ms/step - loss: 0.0727 - accuracy: 0.996
1 - val_loss: 0.0821 - val_accuracy: 0.9955
Epoch 50/50
158/158 [==============================] - 5s 32ms/step - loss: 0.0659 - accuracy: 0.996
0 - val_loss: 0.0847 - val_accuracy: 0.9919
```

My Contribution in this project:

As a Machine Learning, my tasks are basically as follows;

- Tokenizing the input data.
- Making prototype model
- Training of prototype

Tokenizing is the process of dividing text into a set of meaningful pieces. These pieces are called **tokens**. We need to implement this tokenizing to our data of O(n) and O(n^2) so that our computer gets understand any text, **we need** to break that word down in a way that our machine **can** understand.

For example

Input: “I love Python programming language”

Output: I, love, Python, programming, language

I need to deal with **RNN models** as Machine learning in order to model the sequence data. There are some steps of to make RNN model;

- Convert abstracts from list of strings into list of lists of integers (sequences)
- Create features and labels from sequences.
- Build LSTM models with Embedding, LSTM, and Dense layers.
- Load in pre-trained embedding.
- Train model to predict next work in sequence.

Training of prototype

One of the goals of training is to minimize the loss. To do this I used in this project A model.fit () training loop will check at the end of every epoch whether the loss is no longer decreasing. Once it's found no longer decreasing, model. stop training is marked True and the training terminates.

A problem with training neural networks is in the choice of the number of training epochs to use. Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. So we used 50 epochs in this project.

Our project from scratch

Importing Libraries

In [1]:
```python
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras import layers
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import pandas as pd
from random import randint
from numpy import array
from numpy import argmax
from tensorflow.keras.callbacks import EarlyStopping
from random import randint
from numpy import array
from numpy import argmax
from numpy import array_equal
from keras.utils import to_categorical
from keras.models import Model
from keras.layers import Input
from keras.layers import LSTM
from keras.layers import Dense
from numpy import array_equal
from keras.utils import to_categorical
from keras.models import Model
from keras.layers import Input
from keras.layers import LSTM
from keras.layers import Dense
from keras.optimizers import Adam
import nltk
import os
import numpy as np

##nltk.download('punkt')
##nltk.download('averaged_perceptron_tagger')
```

Setting features for Embedding

In [2]:
```
n_features = 500 + 1
n_steps_in = 15
n_steps_out = 15
```

Loading Dataset

In [3]:
```
def load_data(data_path):
  X = []
  Y = []

  for file_ in os.listdir(data_path+'O(n^2)'):
    with open(data_path+'O(n^2)/'+file_) as f:
      X.append(f.read())
    with open(data_path+'O(n)/'+file_) as f:
      Y.append(f.read())

  return X,Y
```

In [4]:
```
def get_dataset(X,Y, len_):

  X= tokenizer.texts_to_sequences(X)
  Y= tokenizer.texts_to_sequences(Y)

  y_=[]
  for i in range(len(Y)):
    for j in range(len(Y[i])):
      y_.append(to_categorical([Y[i][j]], num_classes=n_features))
  y_  = np.array(y_)

  y_ = y_[0:len_]

  x_1=[]
  for i in range(len(X)):
    for j in range(len(X[i])):
      x_1.append(to_categorical([X[i][j]], num_classes=n_features))
  x_1  = np.array(x_1)


  x_1 = x_1[0:len_]

  x_1 = x_1.reshape(int(len_/10),10,n_features)
  y_ = y_.reshape(int(len_/10),10,n_features)




  return x_1,y_
```

In [5]:
```
def train_test_split_(x_1, y_ ,test_size):

  l = int(len(x_1)*test_size)
  l_= int(len(x_1))
  x_1_t =  x_1[0:l_]
  x_1_v = x_1[l:len(x_1)]

  y_t =  y_[0:l_]
  y_v = y_[l:len(y_)]

  return x_1_t, x_1_v, y_t,y_v
```

In [6]:
```
def get_decoder_inp(inp_):
  x_2 = []
  for ind in range(len(inp_)):
    arr = np.zeros((10,501))
    arr[1:10] = inp_[ind][:-1]
    x_2.append(arr)
  x_2 = np.array(x_2)
  return x_2
```

Model AutoEncoder

In [7]:
```

def AutoEncoder(n_input, n_output, n_units):

  encoder_inputs = Input(shape=(None,n_input))
  encoder = LSTM(n_units, return_state=True)
  encoder_outputs, state_h, state_c = encoder(encoder_inputs)
  encoder_states = [state_h, state_c]
  # define training decoder
  decoder_inputs = Input(shape=(None, n_output))
  decoder_lstm = LSTM(n_units, return_sequences=True, return_state=True, dropout= 0)
  decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
  decoder_dense = Dense(n_output, activation='softmax')
  decoder_outputs = decoder_dense(decoder_outputs)
  model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
  # define inference encoder
  encoder_model = Model(encoder_inputs, encoder_states)
  # define inference decoder
  decoder_state_input_h = Input(shape=(n_units,))
  decoder_state_input_c = Input(shape=(n_units,))
  decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
  decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs, initial_state=decoder_states_inputs)
  decoder_states = [state_h, state_c]
  decoder_outputs = decoder_dense(decoder_outputs)
  decoder_model = Model([decoder_inputs] + decoder_states_inputs, [decoder_outputs] + decoder_states)
  # return all models
  return model, encoder_model, decoder_model
```

```
model, infenc, infdec = AutoEncoder(n_features, n_features, 128)
model.summary()
```

```
Model: "functional_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, None, 501)]  0
__________________________________________________________________________________________________
input_2 (InputLayer)            [(None, None, 501)]  0
__________________________________________________________________________________________________
lstm (LSTM)                     [(None, 128), (None, 322560      input_1[0][0]
__________________________________________________________________________________________________
lstm_1 (LSTM)                   [(None, None, 128),  322560      input_2[0][0]
                                                                 lstm[0][1]
                                                                 lstm[0][2]
__________________________________________________________________________________________________
dense (Dense)                   (None, None, 501)    64629       lstm_1[0][0]
==================================================================================================
Total params: 709,749
Trainable params: 709,749
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Load train data

```
X_t,Y_t = load_data('code/train/')
```

Load test data

In [12]:
```
X_v,Y_v = load_data(r'C:/TheMarvelKnight/code/test/')
```

In [13]:
```
X = X_t + X_v
Y = Y_t + Y_v
```

In [14]:
```
tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(X)
x_1,y_ = get_dataset(X, Y,1580)
x_1_t, x_1_v,y_t, y_v=train_test_split_(x_1, y_ ,test_size=0.3)
```

In [15]:
```
x_2_t = get_decoder_inp(y_t)
x_2_v = get_decoder_inp(y_v)
```

The earlystopping callback allowed us to specify the performance measure to monitor trigger, and once triggered , it stopped training process.

In [16]:
```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)
```

We validated our data by 50 epochs.

```
In [17]: history=model.fit([x_1_t, x_2_t], y_t  , epochs=50, batch_size=1,validation_data = ([x_

Epoch 1/50
158/158 [==============================] - 23s 77ms/step - loss: 5.5761 - accuracy: 0.05
35 - val_loss: 4.3741 - val_accuracy: 0.0802
Epoch 2/50
158/158 [==============================] - 7s 42ms/step - loss: 4.3741 - accuracy: 0.050
3 - val_loss: 4.3537 - val_accuracy: 0.0910
Epoch 3/50
158/158 [==============================] - 7s 46ms/step - loss: 4.2976 - accuracy: 0.071
2 - val_loss: 4.5734 - val_accuracy: 0.1468
Epoch 4/50
158/158 [==============================] - 7s 44ms/step - loss: 4.1779 - accuracy: 0.143
```

```
Epoch 48/50
158/158 [==============================] - 4s 25ms/step - loss: 0.0741 - accuracy: 0.997
3 - val_loss: 0.0950 - val_accuracy: 0.9973
Epoch 49/50
158/158 [==============================] - 4s 27ms/step - loss: 0.0727 - accuracy: 0.996
1 - val_loss: 0.0821 - val_accuracy: 0.9955
Epoch 50/50
158/158 [==============================] - 5s 32ms/step - loss: 0.0659 - accuracy: 0.996
0 - val_loss: 0.0847 - val_accuracy: 0.9919
```

REGARDS, ALI USTUNKOL