CISC 867: Project 3

Building a Recurrent Language Model

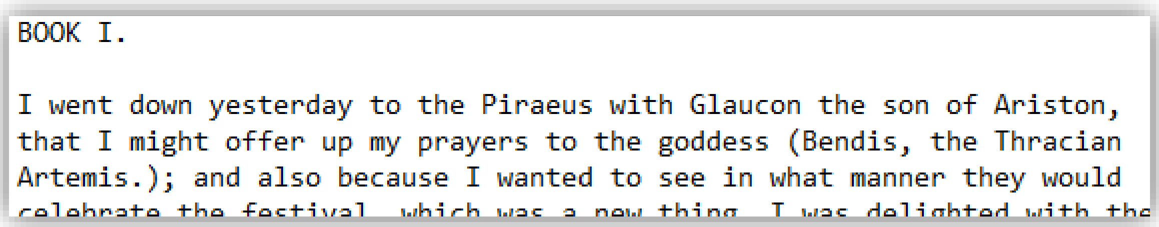
A language model can predict the probability of the next word in the sequence, based on the words already observed in the sequence.
Neural network models are a preferred method for developing statistical language models because they can use a distributed representation where different words with similar meanings have similar representation and because they can use a large context of recently observed words when making predictions.

Data source:

We will use The Republic by Plato as the source text that we can found it by using https://www.gutenberg.org/cache/epub/1497/pg1497.txt. But in this file, we will remove the front and back matter. This includes details about the book at the beginning, a long analysis, and license information at the end. So, the text will begin by:

BOOK I.

I went down yesterday to

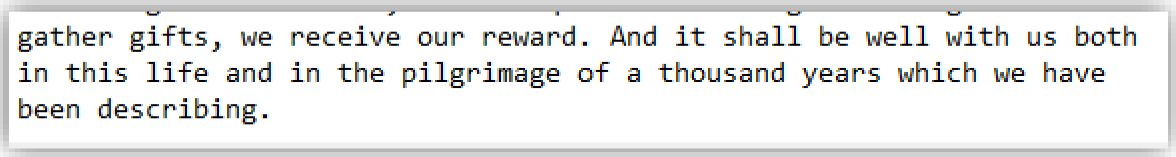


```
BOOK I.

I went down yesterday to the Piraeus with Glaucon the son of Ariston,
that I might offer up my prayers to the goddess (Bendis, the Thracian
Artemis.); and also because I wanted to see in what manner they would
celebrate the festival, which was a new thing. I was delighted with the
```

And end with:

in this life and in the pilgrimage

of a thousand years which we have been describing.



```
gather gifts, we receive our reward. And it shall be well with us both
in this life and in the pilgrimage of a thousand years which we have
been describing.
```

After we check the data file, we found text not cleaned as the project description observed so we will do some preprocessing on the text file to prepare it

CISC 867: Project 3

Data Preparation

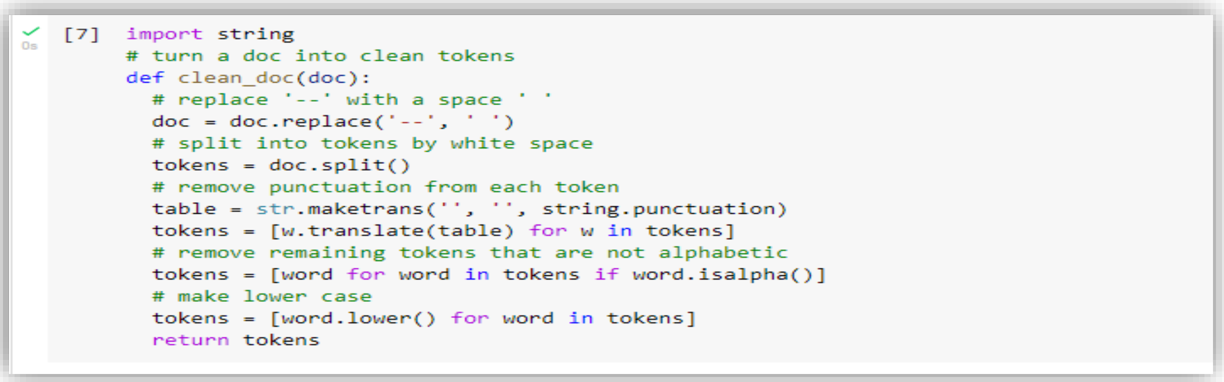
We will start by preparing the data for modeling.

Clean Text

We need to transform the raw text into a sequence of tokens or words that we can use as a source to train the model.

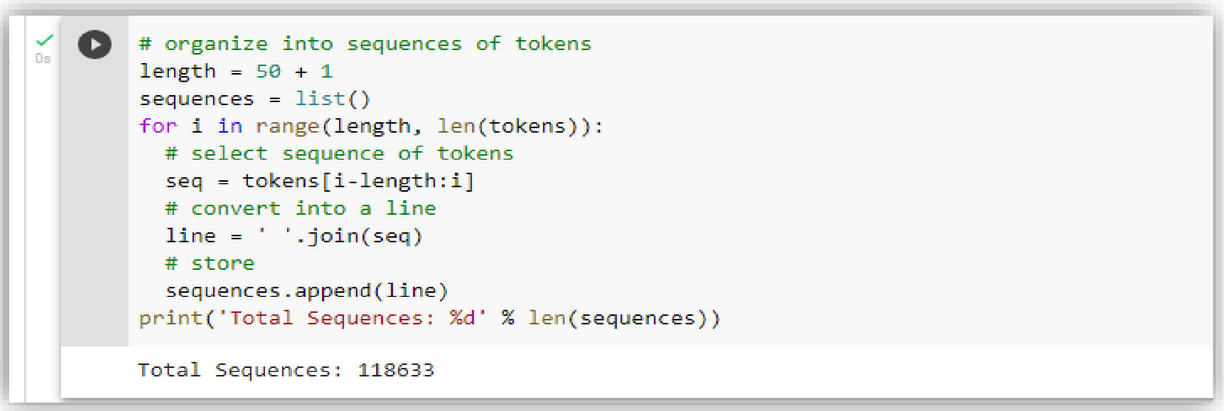
Below are some specific operations we will perform to clean the text. You may want to explore more cleaning operations yourself as an extension.

- Replace ‘–‘with a white space so we can split words better.
- Split words based on white space.
- Remove all punctuation from words to reduce the vocabulary size.
- Remove all words that are not alphabetic to remove standalone punctuation tokens.
- Normalize all words to lowercase to reduce the vocabulary size.



```
import string
# turn a doc into clean tokens
def clean_doc(doc):
  # replace '--' with a space ' '
  doc = doc.replace('--', ' ')
  # split into tokens by white space
  tokens = doc.split()
  # remove punctuation from each token
  table = str.maketrans('', '', string.punctuation)
  tokens = [w.translate(table) for w in tokens]
  # remove remaining tokens that are not alphabetic
  tokens = [word for word in tokens if word.isalpha()]
  # make lower case
  tokens = [word.lower() for word in tokens]
  return tokens
```

We can see that there are just “`Total Tokens: 118684`” words in the clean text and a vocabulary of just “`Unique Tokens: 7409`” words. We can organize the long list of tokens into sequences of 50 input words and 1 output word. That is, sequences of 51 words. So, the “`Total Sequences = 118633`”



```
# organize into sequences of tokens
length = 50 + 1
sequences = list()
for i in range(length, len(tokens)):
  # select sequence of tokens
  seq = tokens[i-length:i]
  # convert into a line
  line = ' '.join(seq)
  # store
  sequences.append(line)
print('Total Sequences: %d' % len(sequences))
```

```
Total Sequences: 118633
```

CISC 867: Project 3

Train Language Model

We can now train a statistical language model from the prepared data.

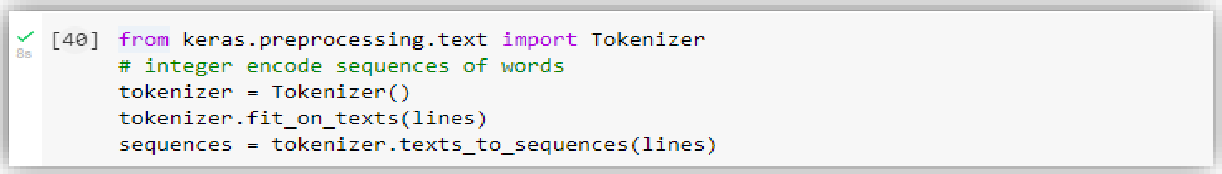
The model we will train is a neural language model. It has a few unique characteristics:

- It uses a distributed representation for words so that different words with similar meanings will have a similar representation.
- It learns the representation at the same time as learning the model.
- It learns to predict the probability for the next word using the context of the last 100 words.

Specifically, we will use an Embedding Layer to learn the representation of words, and a Long Short-Term Memory (LSTM) recurrent neural network to learn to predict words based on their context.

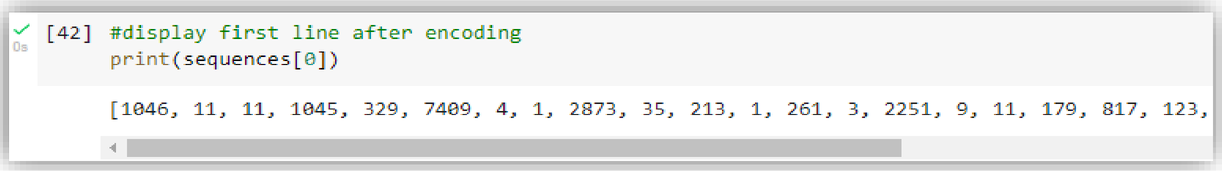
Encode Sequences

The word embedding layer expects input sequences to be comprised of integers. We can map each word in our vocabulary to a unique integer and encode our input sequences.



```
[40] from keras.preprocessing.text import Tokenizer
     # integer encode sequences of words
     tokenizer = Tokenizer()
     tokenizer.fit_on_texts(lines)
     sequences = tokenizer.texts_to_sequences(lines)
```

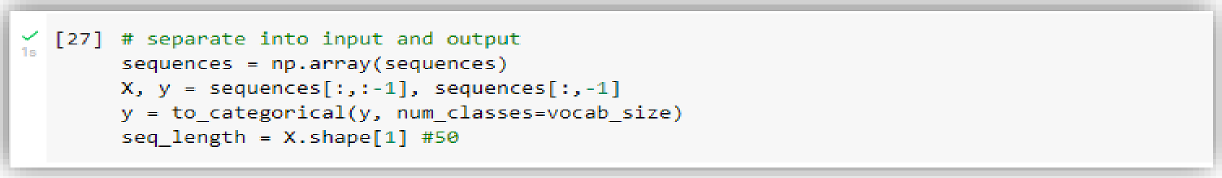
We can check the first line in the sequences after apply tokenizer.texts_to_sequences



```
[42] #display first line after encoding
     print(sequences[0])

     [1046, 11, 11, 1045, 329, 7409, 4, 1, 2873, 35, 213, 1, 261, 3, 2251, 9, 11, 179, 817, 123,
```

Sequence Inputs and Output

Now that we have encoded the input sequences, we need to separate them into input (*X*) and output (*y*) elements. We can do this with array slicing.



```
[27] # separate into input and output
     sequences = np.array(sequences)
     X, y = sequences[:,:-1], sequences[:,-1]
     y = to_categorical(y, num_classes=vocab_size)
     seq_length = X.shape[1] #50
```

CISC 867: Project 3

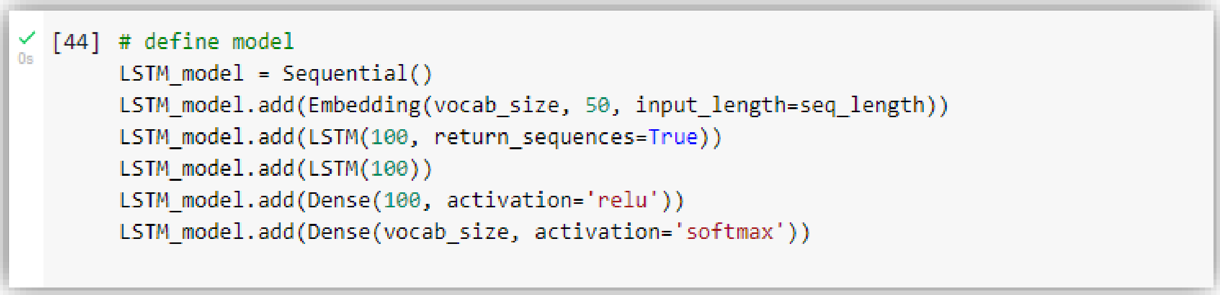
Fit LSTM Model

We can now define and fit our language model on the training data.

The learned embedding needs to know the size of the vocabulary and the length of input sequences as previously discussed. It also has a parameter to specify how many dimensions will be used to represent each word.

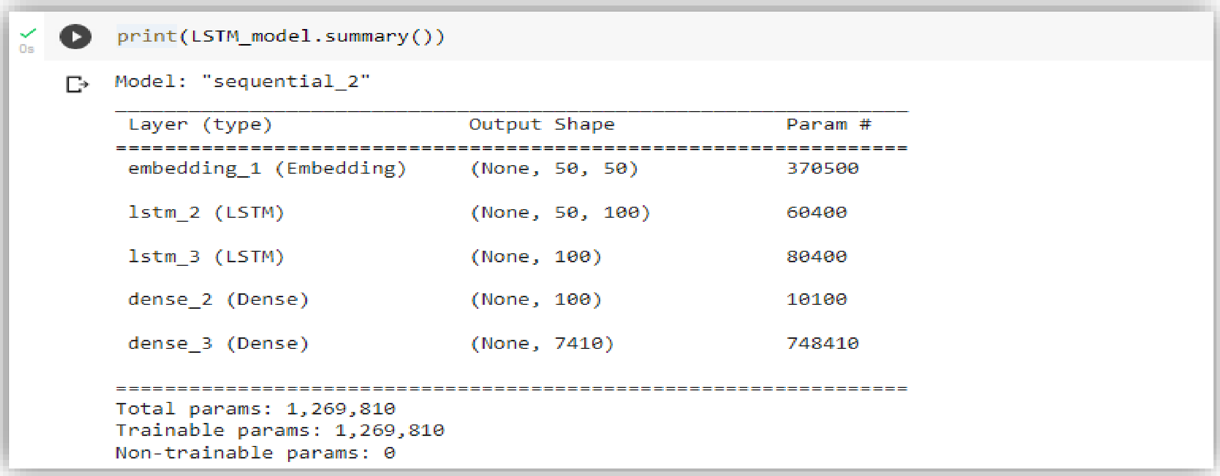
We will use a two LSTM hidden layers with 100 memory cells each.

A dense fully connected layer with 100 neurons connects to the LSTM hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A SoftMax activation function is used to ensure the outputs have the characteristics of normalized probabilities.



```
# define model
LSTM_model = Sequential()
LSTM_model.add(Embedding(vocab_size, 50, input_length=seq_length))
LSTM_model.add(LSTM(100, return_sequences=True))
LSTM_model.add(LSTM(100))
LSTM_model.add(Dense(100, activation='relu'))
LSTM_model.add(Dense(vocab_size, activation='softmax'))
```

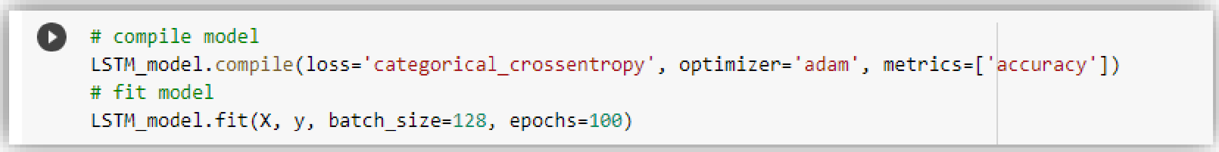
We can print the summary about our model as below:



```
print(LSTM_model.summary())
```

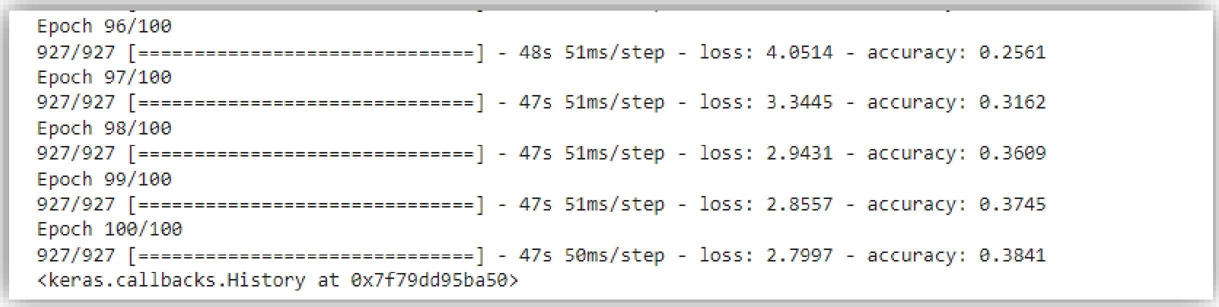
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 50, 50)            370500

 lstm_2 (LSTM)               (None, 50, 100)           60400

 lstm_3 (LSTM)               (None, 100)               80400

 dense_2 (Dense)             (None, 100)               10100

 dense_3 (Dense)             (None, 7410)              748410

=================================================================
Total params: 1,269,810
Trainable params: 1,269,810
Non-trainable params: 0
```

the model is fit on the data for 100 training epochs with a modest batch size of 128 to speed things up.



```
# compile model
LSTM_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
LSTM_model.fit(X, y, batch_size=128, epochs=100)
```

After we run our LSTM model, we observed that an accuracy increases for each epoch so we can increase the number of epochs to make good prediction for the next word in the sequence, which is not bad. We are not aiming for 100% accuracy

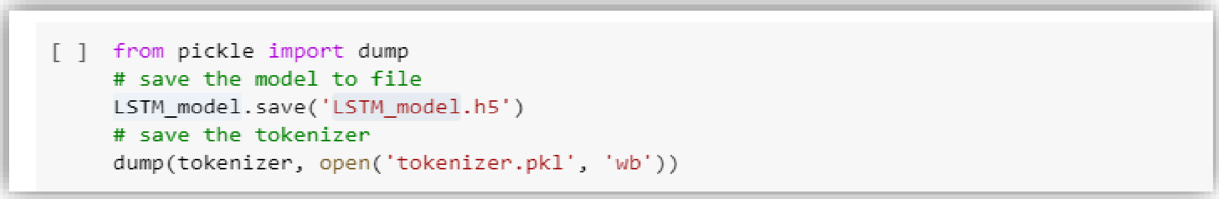


```
Epoch 96/100
927/927 [==============================] - 48s 51ms/step - loss: 4.0514 - accuracy: 0.2561
Epoch 97/100
927/927 [==============================] - 47s 51ms/step - loss: 3.3445 - accuracy: 0.3162
Epoch 98/100
927/927 [==============================] - 47s 51ms/step - loss: 2.9431 - accuracy: 0.3609
Epoch 99/100
927/927 [==============================] - 47s 51ms/step - loss: 2.8557 - accuracy: 0.3745
Epoch 100/100
927/927 [==============================] - 47s 50ms/step - loss: 2.7997 - accuracy: 0.3841
<keras.callbacks.History at 0x7f79dd95ba50>
```

Save Model

At the end of the run, the trained model is saved to file.
Here, we use the Keras model API to save the model to the file 'LSTM_model.*h5*' in the current working directory.

Later, when we load the model to make predictions



```
[ ]  from pickle import dump
     # save the model to file
     LSTM_model.save('LSTM_model.h5')
     # save the tokenizer
     dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Based on our project description we will check another model such as GRU model so we will describe it below

CISC 867: Project 3

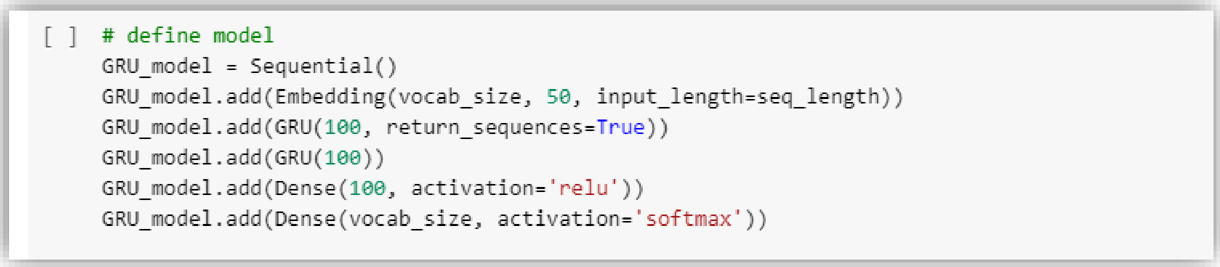
Fit GRU Model

We can now define and fit our language model on the training data.

The learned embedding needs to know the size of the vocabulary and the length of input sequences as previously discussed. It also has a parameter to specify how many dimensions will be used to represent each word.

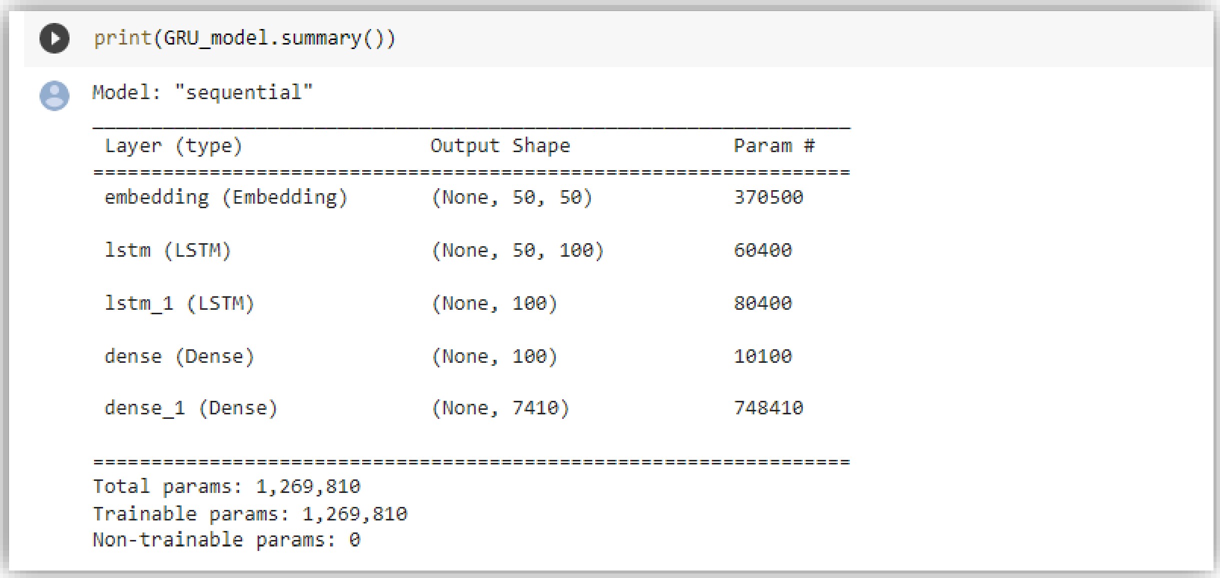
We will use a two GRU hidden layers with 100 memory cells each.

A dense fully connected layer with 100 neurons connects to the GRU hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A SoftMax activation function is used to ensure the outputs have the characteristics of normalized probabilities.



```
# define model
GRU_model = Sequential()
GRU_model.add(Embedding(vocab_size, 50, input_length=seq_length))
GRU_model.add(GRU(100, return_sequences=True))
GRU_model.add(GRU(100))
GRU_model.add(Dense(100, activation='relu'))
GRU_model.add(Dense(vocab_size, activation='softmax'))
```

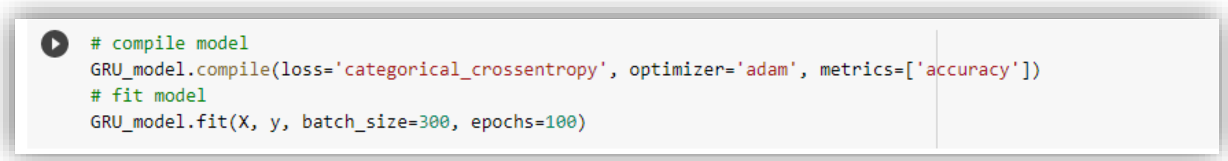
We can print the summary about our model as below:



```
print(GRU_model.summary())
```

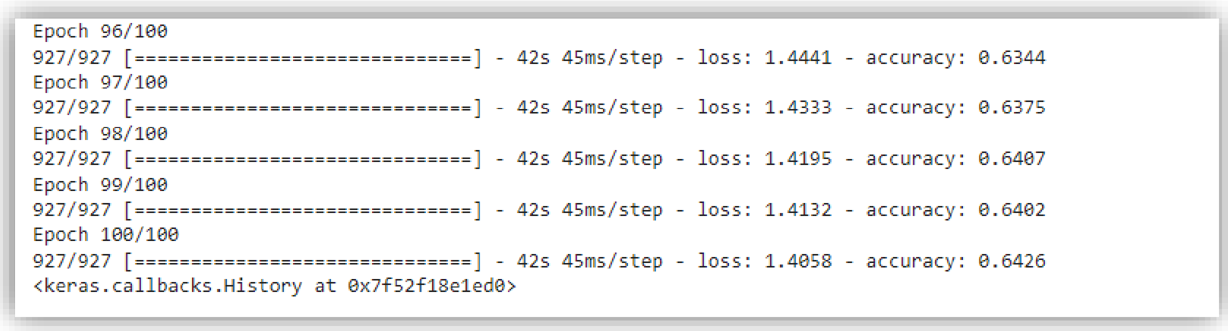
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 50, 50)            370500

 lstm (LSTM)                 (None, 50, 100)           60400

 lstm_1 (LSTM)               (None, 100)               80400

 dense (Dense)               (None, 100)               10100

 dense_1 (Dense)             (None, 7410)              748410

=================================================================
Total params: 1,269,810
Trainable params: 1,269,810
Non-trainable params: 0
```

the model is fit on the data for 100 training epochs with a modest batch size of 300 to speed things up.



```
# compile model
GRU_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
GRU_model.fit(X, y, batch_size=300, epochs=100)
```

After we run our GRU model, we observed that an accuracy of just around 65% of predicting the next word in the sequence, but if we increase the number of epochs the acc will increase which is not bad. We are not aiming for 100% accuracy

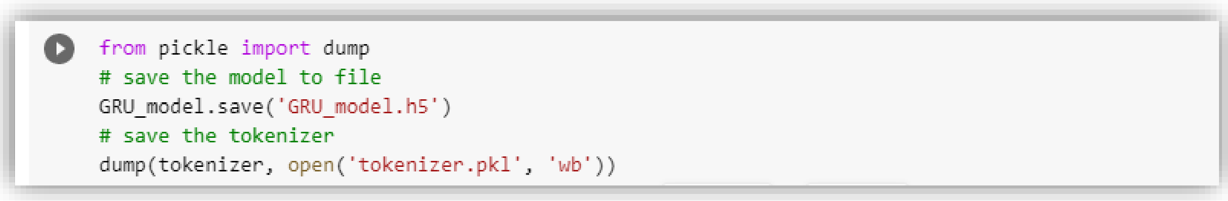


```
Epoch 96/100
927/927 [==============================] - 42s 45ms/step - loss: 1.4441 - accuracy: 0.6344
Epoch 97/100
927/927 [==============================] - 42s 45ms/step - loss: 1.4333 - accuracy: 0.6375
Epoch 98/100
927/927 [==============================] - 42s 45ms/step - loss: 1.4195 - accuracy: 0.6407
Epoch 99/100
927/927 [==============================] - 42s 45ms/step - loss: 1.4132 - accuracy: 0.6402
Epoch 100/100
927/927 [==============================] - 42s 45ms/step - loss: 1.4058 - accuracy: 0.6426
<keras.callbacks.History at 0x7f52f18e1ed0>
```

Save Model

At the end of the run, the trained model is saved to file.
Here, we use the Keras model API to save the model to the file 'GRU_model.*h5*' in the current working directory.

Later, when we load the model to make predictions



```
from pickle import dump
# save the model to file
GRU_model.save('GRU_model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

We can add more GRU layers and check the accuracy and we can check by add more units so we will make more model by different structures

CISC 867: Project 3

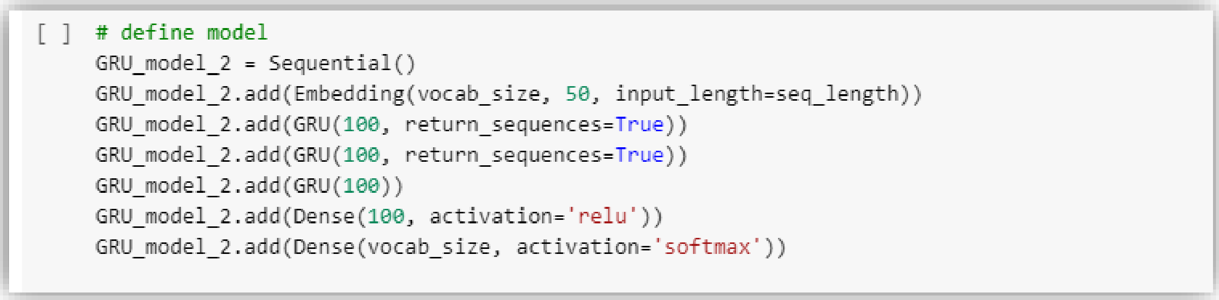
Fit GRU Model 2

We can now define and fit our language model on the training data.

The learned embedding needs to know the size of the vocabulary and the length of input sequences as previously discussed. It also has a parameter to specify how many dimensions will be used to represent each word.

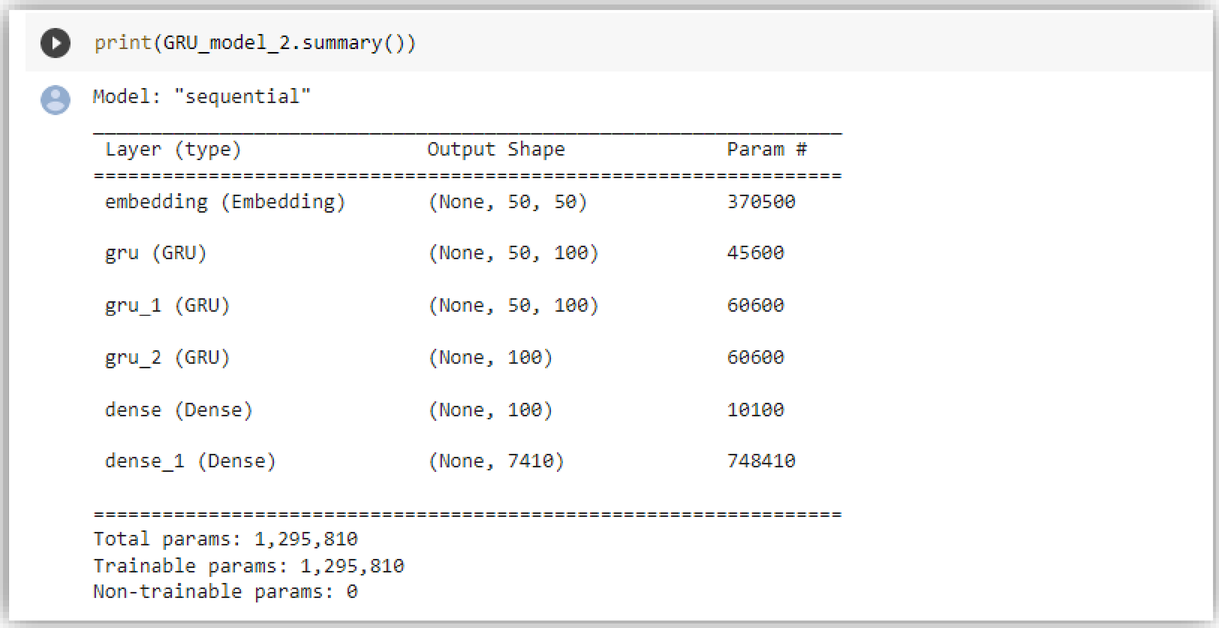
We will use a three GRU hidden layers with 100 memory cells each.

A dense fully connected layer with 100 neurons connects to the GRU hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A SoftMax activation function is used to ensure the outputs have the characteristics of normalized probabilities.



```
[ ]  # define model
     GRU_model_2 = Sequential()
     GRU_model_2.add(Embedding(vocab_size, 50, input_length=seq_length))
     GRU_model_2.add(GRU(100, return_sequences=True))
     GRU_model_2.add(GRU(100, return_sequences=True))
     GRU_model_2.add(GRU(100))
     GRU_model_2.add(Dense(100, activation='relu'))
     GRU_model_2.add(Dense(vocab_size, activation='softmax'))
```

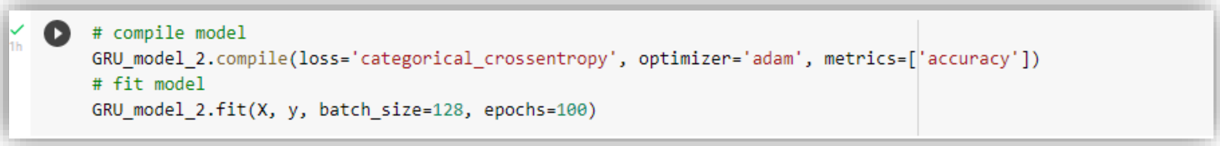
We can print the summary about our model as below:



```
print(GRU_model_2.summary())
```

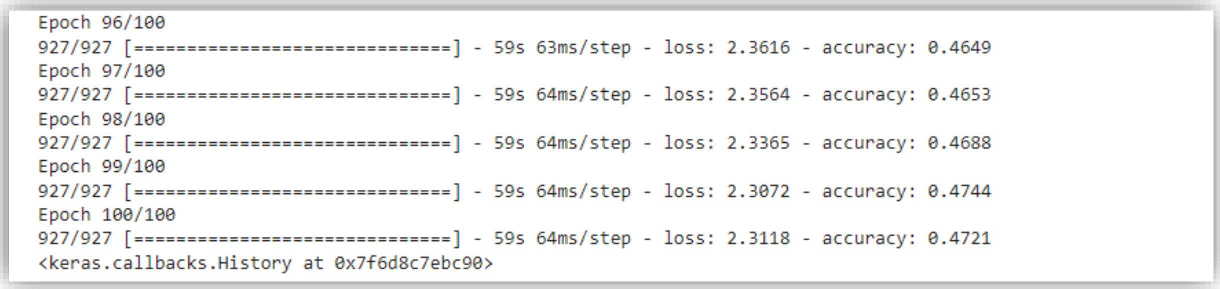
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 50, 50)            370500

 gru (GRU)                   (None, 50, 100)           45600

 gru_1 (GRU)                 (None, 50, 100)           60600

 gru_2 (GRU)                 (None, 100)               60600

 dense (Dense)               (None, 100)               10100

 dense_1 (Dense)             (None, 7410)              748410

=================================================================
Total params: 1,295,810
Trainable params: 1,295,810
Non-trainable params: 0
```

the model is fit on the data for 100 training epochs with a modest batch size of 128 to speed things up.



```
# compile model
GRU_model_2.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
GRU_model_2.fit(X, y, batch_size=128, epochs=100)
```

After we run our GRU model 2, we observed that an accuracy of just around 50% of predicting the next word in the sequence, but if we increase the number of epochs the acc will increase which is not bad. We are not aiming for 100% accuracy

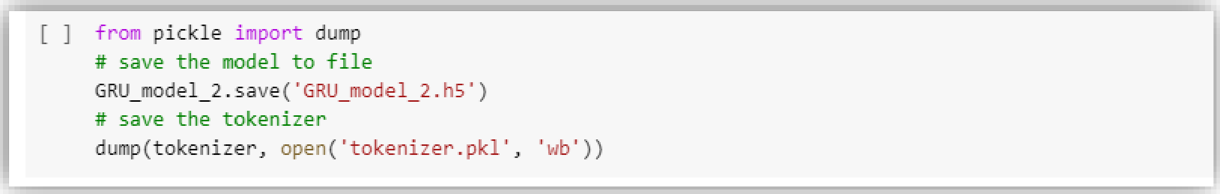


```
Epoch 96/100
927/927 [==============================] - 59s 63ms/step - loss: 2.3616 - accuracy: 0.4649
Epoch 97/100
927/927 [==============================] - 59s 64ms/step - loss: 2.3564 - accuracy: 0.4653
Epoch 98/100
927/927 [==============================] - 59s 64ms/step - loss: 2.3365 - accuracy: 0.4688
Epoch 99/100
927/927 [==============================] - 59s 64ms/step - loss: 2.3072 - accuracy: 0.4744
Epoch 100/100
927/927 [==============================] - 59s 64ms/step - loss: 2.3118 - accuracy: 0.4721
<keras.callbacks.History at 0x7f6d8c7ebc90>
```

Save Model

At the end of the run, the trained model is saved to file.
Here, we use the Keras model API to save the model to the file 'GRU_model2.*h5*' in the current working directory.

Later, when we load the model to make predictions



```
[ ]  from pickle import dump
     # save the model to file
     GRU_model_2.save('GRU_model_2.h5')
     # save the tokenizer
     dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

We can change at the next model in number of units and check the accuracy

CISC 867: Project 3

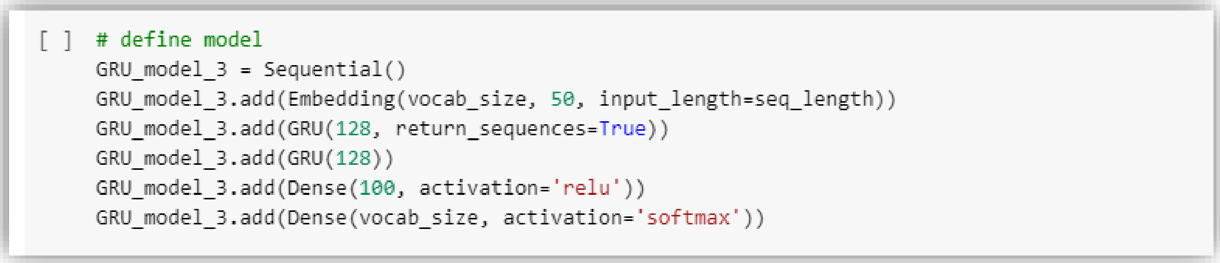
Fit GRU Model 3

We can now define and fit our language model on the training data.

The learned embedding needs to know the size of the vocabulary and the length of input sequences as previously discussed. It also has a parameter to specify how many dimensions will be used to represent each word.

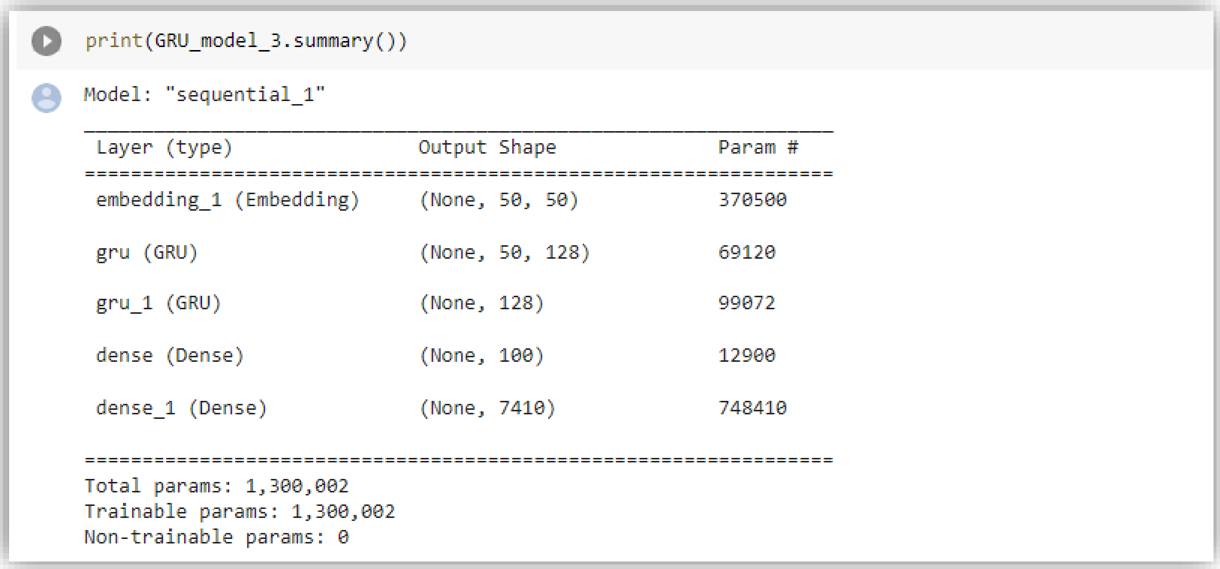
We will use a three GRU hidden layers with 128 memory cells each.

A dense fully connected layer with 100 neurons connects to the GRU hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A SoftMax activation function is used to ensure the outputs have the characteristics of normalized probabilities.



```
[ ] # define model
    GRU_model_3 = Sequential()
    GRU_model_3.add(Embedding(vocab_size, 50, input_length=seq_length))
    GRU_model_3.add(GRU(128, return_sequences=True))
    GRU_model_3.add(GRU(128))
    GRU_model_3.add(Dense(100, activation='relu'))
    GRU_model_3.add(Dense(vocab_size, activation='softmax'))
```

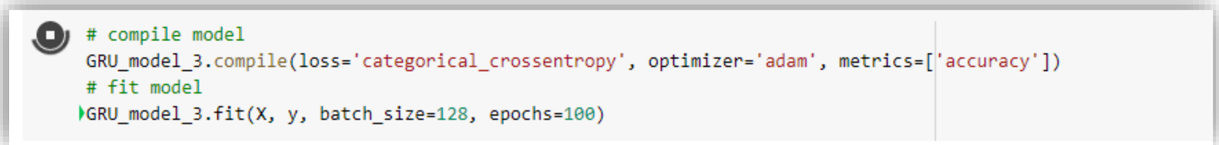
We can print the summary about our model as below:



```
print(GRU_model_3.summary())
```

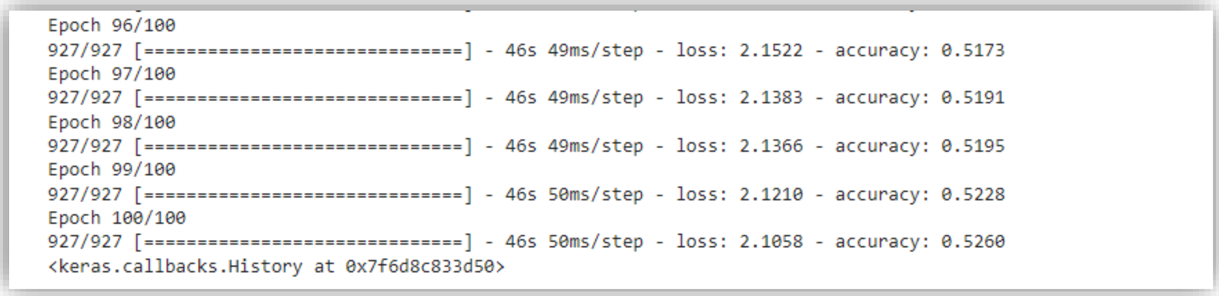
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding_1 (Embedding)     (None, 50, 50)            370500    
                                                                 
 gru (GRU)                   (None, 50, 128)           69120     
                                                                 
 gru_1 (GRU)                 (None, 128)               99072     
                                                                 
 dense (Dense)               (None, 100)               12900     
                                                                 
 dense_1 (Dense)             (None, 7410)              748410    
                                                                 
=================================================================
Total params: 1,300,002
Trainable params: 1,300,002
Non-trainable params: 0
```

the model is fit on the data for 100 training epochs with a modest batch size of 128 to speed things up.



```
# compile model
GRU_model_3.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
GRU_model_3.fit(X, y, batch_size=128, epochs=100)
```

After we run our GRU model 3, we observed that an accuracy of just around 55% of predicting the next word in the sequence, but if we increase the number of epochs the acc will increase which is not bad. We are not aiming for 100% accuracy

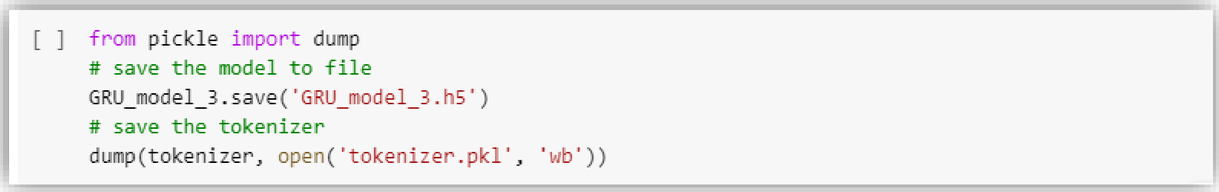


```
Epoch 96/100
927/927 [==============================] - 46s 49ms/step - loss: 2.1522 - accuracy: 0.5173
Epoch 97/100
927/927 [==============================] - 46s 49ms/step - loss: 2.1383 - accuracy: 0.5191
Epoch 98/100
927/927 [==============================] - 46s 49ms/step - loss: 2.1366 - accuracy: 0.5195
Epoch 99/100
927/927 [==============================] - 46s 50ms/step - loss: 2.1210 - accuracy: 0.5228
Epoch 100/100
927/927 [==============================] - 46s 50ms/step - loss: 2.1058 - accuracy: 0.5260
<keras.callbacks.History at 0x7f6d8c833d50>
```

Save Model

At the end of the run, the trained model is saved to file.
Here, we use the Keras model API to save the model to the file 'GRU_model3.*h5*' in the current working directory.

Later, when we load the model to make predictions



```
[ ] from pickle import dump
    # save the model to file
    GRU_model_3.save('GRU_model_3.h5')
    # save the tokenizer
    dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

We can change at the next model in number of units by reducing it and check the accuracy

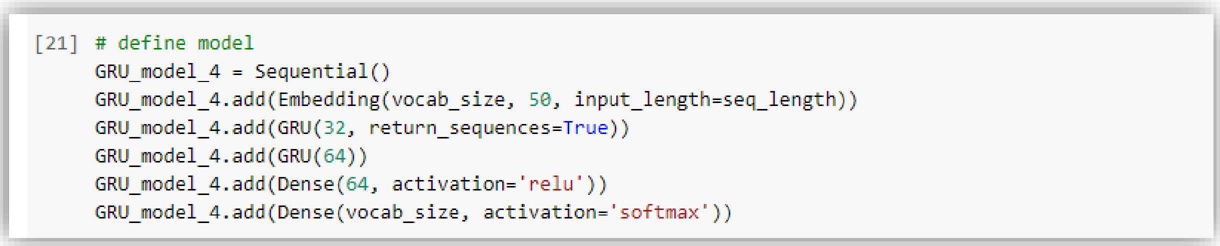
Fit GRU Model 4

We can now define and fit our language model on the training data.

The learned embedding needs to know the size of the vocabulary and the length of input sequences as previously discussed. It also has a parameter to specify how many dimensions will be used to represent each word.

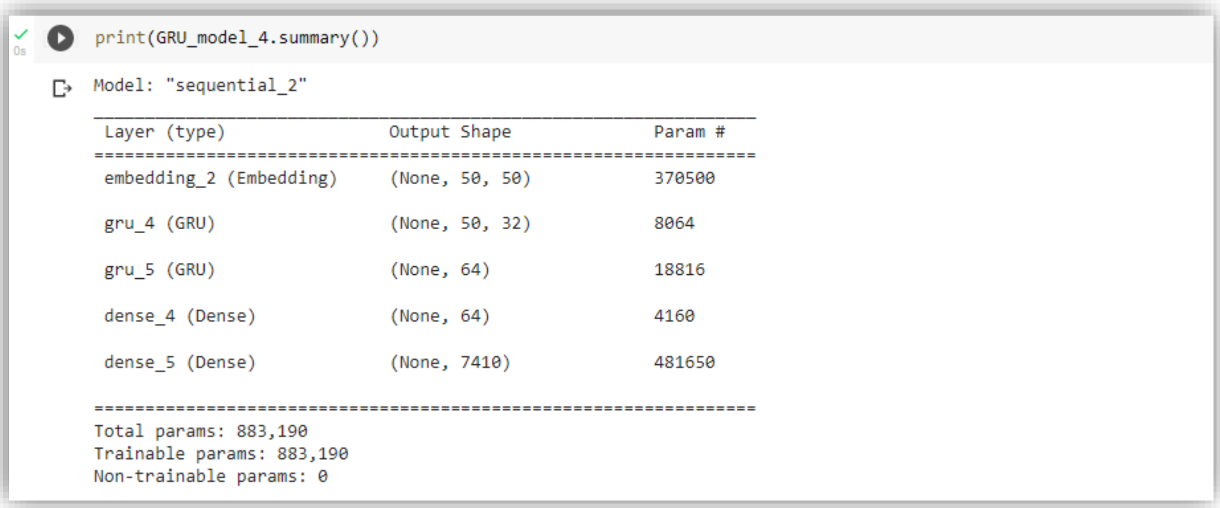
We will use a three GRU hidden layers one with 32 unit and another with 64 memory cells each.

A dense fully connected layer with 64 neurons connects to the GRU hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A SoftMax activation function is used to ensure the outputs have the characteristics of normalized probabilities.



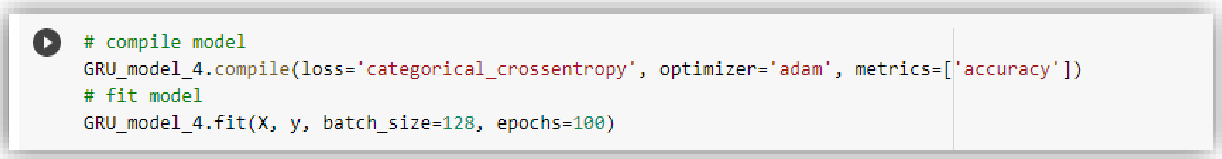
```
[21] # define model
     GRU_model_4 = Sequential()
     GRU_model_4.add(Embedding(vocab_size, 50, input_length=seq_length))
     GRU_model_4.add(GRU(32, return_sequences=True))
     GRU_model_4.add(GRU(64))
     GRU_model_4.add(Dense(64, activation='relu'))
     GRU_model_4.add(Dense(vocab_size, activation='softmax'))
```

We can print the summary about our model as below:



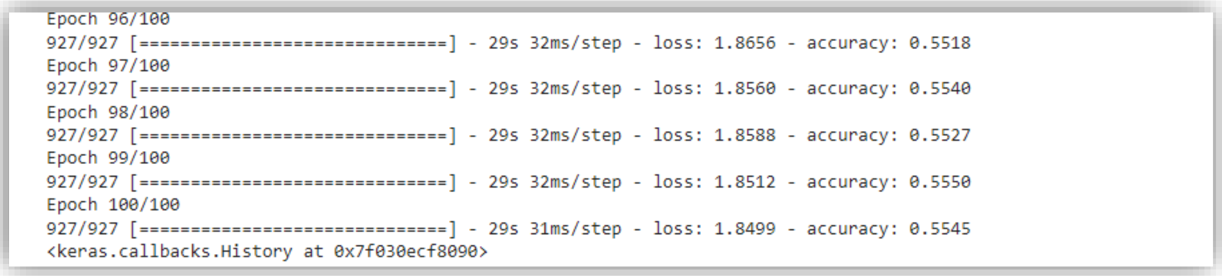
```
print(GRU_model_4.summary())

Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 50, 50)            370500

 gru_4 (GRU)                 (None, 50, 32)            8064

 gru_5 (GRU)                 (None, 64)                18816

 dense_4 (Dense)             (None, 64)                4160

 dense_5 (Dense)             (None, 7410)              481650

=================================================================
Total params: 883,190
Trainable params: 883,190
Non-trainable params: 0
```

the model is fit on the data for 100 training epochs with a modest batch size of 128 to speed things up.



```
# compile model
GRU_model_4.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
GRU_model_4.fit(X, y, batch_size=128, epochs=100)
```

After we run our GRU model 4, we observed that an accuracy of just around 55% of predicting the next word in the sequence, but if we increase the number of epochs the acc will increase which is not bad. We are not aiming for 100% accuracy

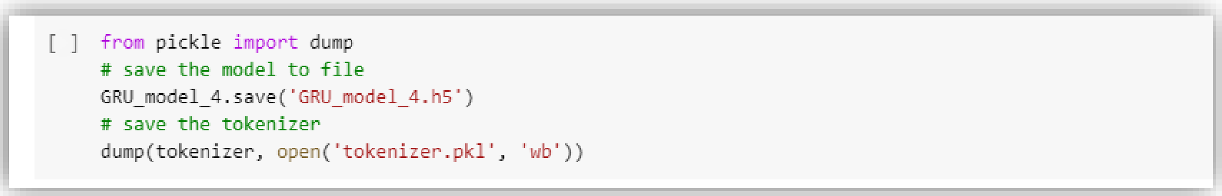


```
Epoch 96/100
927/927 [==============================] - 29s 32ms/step - loss: 1.8656 - accuracy: 0.5518
Epoch 97/100
927/927 [==============================] - 29s 32ms/step - loss: 1.8560 - accuracy: 0.5540
Epoch 98/100
927/927 [==============================] - 29s 32ms/step - loss: 1.8588 - accuracy: 0.5527
Epoch 99/100
927/927 [==============================] - 29s 32ms/step - loss: 1.8512 - accuracy: 0.5550
Epoch 100/100
927/927 [==============================] - 29s 31ms/step - loss: 1.8499 - accuracy: 0.5545
<keras.callbacks.History at 0x7f030ecf8090>
```

Save Model

At the end of the run, the trained model is saved to file.
Here, we use the Keras model API to save the model to the file 'GRU_model4.*h5*' in the current working directory.

Later, when we load the model to make predictions



```
[ ] from pickle import dump
    # save the model to file
    GRU_model_4.save('GRU_model_4.h5')
    # save the tokenizer
    dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Note: After we finish in NLP, I would recommend using BLEU instead of accuracy. Accuracy does not have any useful meaning. The Python Natural Language Toolkit library, or NLTK, provides an implementation of the BLEU score that you can use to evaluate your generated text against a reference.

CISC 867: Project 3

Use Language Model

Now that we have a trained language model, we can use it.

In this case, we can use it to generate new sequences of text that have the same statistical properties as the source text.

This is not practical, at least not for this example, but it gives a concrete example of what the language model has learned.

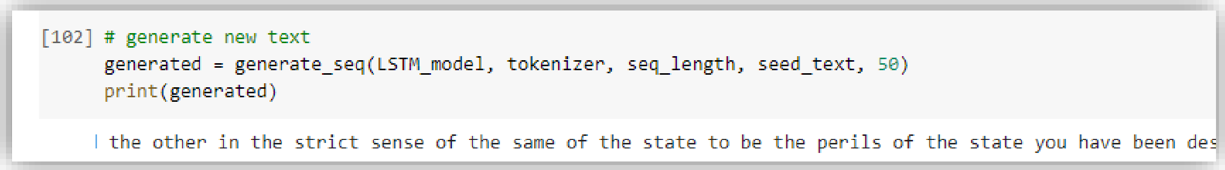
Generate Text

The first step in generating text is preparing a seed input.

We will select a random line of text from the input text for this purpose.

We can create a function called generate_seq() that takes as input the model, the tokenizer, input sequence length, the seed text, and the number of words to generate. It then returns a sequence of words generated by the model.

We are now ready to generate a sequence of new words given some seed text.



```
[102] # generate new text
      generated = generate_seq(LSTM_model, tokenizer, seq_length, seed_text, 50)
      print(generated)

        the other in the strict sense of the same of the state to be the perils of the state you have been des
```

Done by: -

Ahmed Ibrahim Salem Abdelhamid