University of Missouri-Kansas City

School of Computing and Engineering

ICP: ICP-05

Course Name: Big Data Analytics and Applications
Course ID: COMP-SCI 5542

Semester /Session : Spring 2022

Student ID : **Student Name**

16334245 Mustavi Islam

16321217 Keenan Flynn

Description of the Problem :
Recurrent neural networks and LSTMs help us solve problems with streamed data. This data can also be thought of as time series data. Each piece of information is fed into the RNN step by step. The goal of the RNN is to predict the next time step or the next series of time steps. RNNs and LSTMs do this by taking the previous outputs or hidden states as an input. In this way our network can begin to build up a type of memory so that it can predict certain types of sequences.

ICP 5 focuses on the layers and hyperparameter tuning of a recurrent neural network. We decided to make our model an LSTM as this type of model has higher memory capacity than a traditional LSTM. We also tuned some hyperparameters which can change the learning rate and computational cost of the model.

Description of Solution :

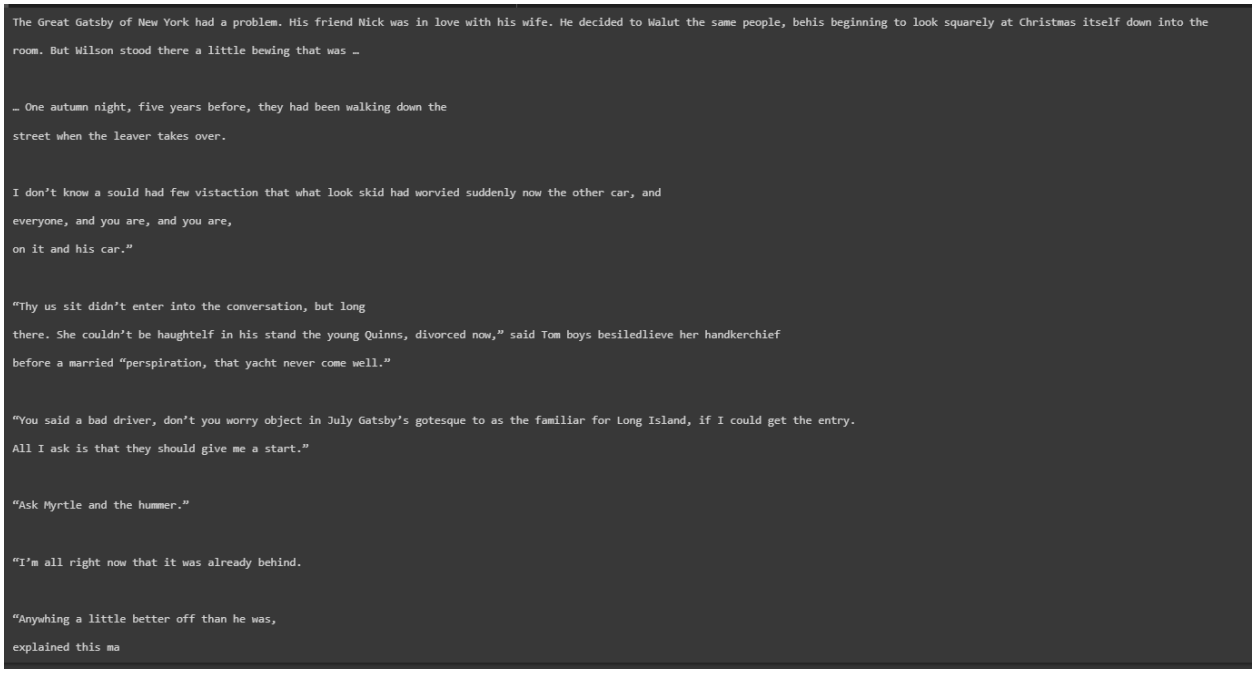
1. The model provided in the source code for the ICP was adapted for a Project Gutenberg book
    a. We used the book 'The Great Gatsby'. We sliced this initial book to get rid of some legal disclosures.
    b. We added some preprocessing to the model. We used a Counter() object to aggregate the data into buckets by character and get counts for each character.
    c. With these counts, we made some decisions to remove some characters and replace them with others. Essentially we removed outliers of the data. For instance, there was one instance of the character 'ç' which was replaced with a 'c'. 'We hope that this will have a small effect on the output.
    d. We also assigned indexes based on these counts. There were 81 unique characters. The lowest frequency character 'Z' got assigned the index of 0 while the highest frequency character ' ' (space) got assigned the index of 80. By doing this we hope to introduce logic into the numeric indexes. The higher the number, the greater likelihood that the character has a high frequency.
2. Processing the text and changing sequence length.
    a. We kept the text processing mostly the same as in the source code.
    b. One change we made was to increase the sequence length. The sequence length is the amount of characters in a set which the model then has to predict from. We have a large dataset (270,000) characters, so increasing the sequence length will allow the model to have more information in which to make its decision.
    c. We changed the sequence length to 200.

    d. Doing this gives the model 200 time steps in which to learn the characters.
3. Changing Batch size and creating an infinite dataset.
    a. We learned in the last ICP that a low batch size and low epochs lead to a fast learning rate, but doing this fails to capture some of the variance in the data and thus increases the loss.
    b. We decided to increase the batch size to 128. With a larger batch size, the model ‘sees’ more data before it starts learning. This allows the model to get closer to the global optima, meaning that the gradient is better.
    c. With a larger batch size, we will need to increase the number of epochs as a higher batch size decreases the learning rate.
    d. We also made our dataset ‘infinite’ through use of the repeat() helper function. This function makes it so we will never run out of data in the training process.
4. Increasing epoch size
    a. In this ICP we have increased the epoch size dynamically. Ideally we want the model to stop when it hits the most optimal epoch. By hard coding the epoch number we are unable to always stop at the best epoch.
    b. One solution is to add callbacks. We have 2 callbacks, 1 tracks the weights of the previous epochs. The other stops the model when the loss ceases to decrease.
    c. This way we can increase the epoch number to a size of our choosing, but the model will hopefully stop itself before then.
    d. We used an epoch size of 50.
5. Adding an LSTM layer
    a. The LSTM layer replaced the GRU. The GRU has 2 gates: the reset and update gates, whereas the LSTM has 3 gates: input, output, and forget gates. The GRU is easy to stand up for simple problems, but the LSTM generally outperforms the GRU in language modeling problems because the LSTM has a longer ‘memory’.
    b. The Keras model of sequential neural networks allowed us to easily replace the GRU layer with an LSTM layer.
    c. To use the LSTM layer, we need an additional hyperparameter named the steps_per_epoch. Because we added the repeat() helper to the shuffle function, our dataset is theoretically endless. The steps_per_epoch parameter tells the model how many batches to learn on during each epoch. We found good performance by changing this number to 200.
6. Adding dropout
    a. During one of our test runs, the generate_text() function generate an expert straight out of the training data. We want this function to generate

pseudo-real text and not repeat text that it has already seen. This is a case of overfitting.

b. We can slow down overfitting by adding in a Dropout() layer so that some of the data becomes hidden from the LSTM.
c. We used a dropout ratio of 0.1.

7. Changing Temperature
    a. During our testing, we found that sometimes the model started to copy text from the training set and output it as 'new' text. This is a case of overfitting. We found that changing the temperature in the generate_text() function could help to prevent this from happening.
    b. Our final model has a temperature of 1.3.
8. Results



Transcript of our output:

The Great Gatsby of New York had a problem. His friend Nick was in love with his wife. He decided to Walut the same people, behis beginning to look squarely at Christmas itself down into the
room. But Wilson stood there a little bewing that was …

… One autumn night, five years before, they had been walking down the
street when the leaver takes over.

I don't know a sould had few vistaction that what look skid had worvied suddenly now the other car, and
everyone, and you are, and you are,
on it and his car."

"Thy us sit didn't enter into the conversation, but long
there. She couldn't be haughtelf in his stand the young Quinns, divorced now," said Tom boys besiledlieve her handkerchief
before a married "perspiration, that yacht never come well."

"You said a bad driver, don't you worry object in July Gatsby's gotesque to as the familiar for Long Island, if I could get the entry.

All I ask is that they should give me a start."

"Ask Myrtle and the hummer."

"I'm all right now that it was already behind.

"Anywhing a little better off than he was,
explained this ma

Challenges:
The biggest challenge this ICP faced was making informed decisions regarding the tuning of hyperparameters. The LSTM neural network takes a significant amount of time to train. We found that tuning these hyperparameters could drastically increase the running time of the model. Deadlines exist in this class and in industry settings, so doing research is extremely important before changing things about the model. This is where scientific method can greatly help expedite things. We need to first make observations about the model, form a hypothesis and ask a question, do research about the question, and finally test the hypothesis. In this way we are not wasting our time with poorly made hyperparameters.

Learning Outcomes :

1. We learned how LSTMs work.
2. We learned how to process text data to help solve a time series question.
3. We learned about callback functions.

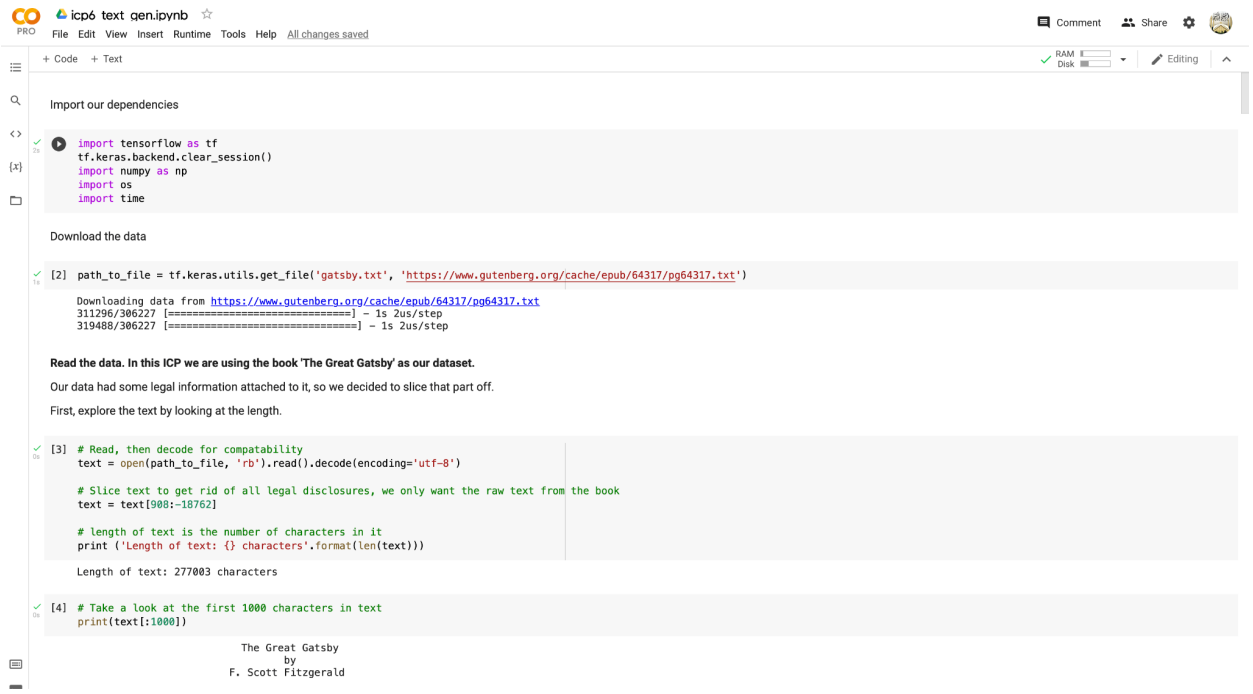
4. We learned how to test a neural network before training it so as to get an idea of the output.
5. We learned how loss was affected by batch and epoch size.

Resources:
https://towardsdatascience.com/generating-cooking-recipes-using-tensorflow-and-lstm-recurrent-neural-network-a7bf242acad3

Used this link for general information and some code snippets

Video Link:
https://youtu.be/QMQ9mBdOfYE

Screenshots :



```
[4] that I've been turning over in my mind ever since.

    "Whenever you feel like criticizing anyone," he told me, "just
    remember that all the people in this world haven't had the advantages
    that you've had."

    He didn't say any more, but we've always been unusually communicative
    in a reserved way, and I understood that he meant a great deal more
    than that. In consequence, I'm inclined to reserve all
```

Data Exploration

Lets group the text by character to see the frequency distribution of each character

This is the frequency distribution for the raw data. We use the Counter module to aggregate the characters into buckets. We can then make decisions about preprocessing the data.

```
[5] from collections import Counter

    # using collections.Counter() to get
    # count of each element in string
    res = Counter(text)

    # printing result
    print ("Count of all characters in Gatsby is :\n " +  str(res))
```

```
Count of all characters in Gatsby is :
 Counter({' ': 44191, 'e': 25006, 't': 18096, 'a': 16841, 'o': 15738, 'n': 14063, 'i': 12528, 's': 12367, 'h': 12237, 'r': 11339, 'd': 9609, 'l': 8173, '\r': 6399, '\n': 6398, 'u': 5833, 'm':
```

icp6 text gen.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
RAM Disk Editing

We have decided to remove some characters and replace others.

This reduction of data will hopefully help the model to learn faster.

```
[6] # Use replace() to remove characters and replace with another
    filtered_text = text.replace('\r', '\n').replace('\u200a', ' ').replace('é', 'e').replace('[', '(').replace(']', ')').replace('$', '').replace('ô','o').replace('ê','e').replace('ç','c')

    # using collections.Counter() to get
    # count of each element in string
    char_freq = Counter(filtered_text)

    # printing result
    # we now have removed some of the outliers
    print ("Count of filtered characters in Gatsby is :\n " +  str(char_freq))
```

```
Count of filtered characters in Gatsby is :
 Counter({' ': 44203, 'e': 25024, 't': 18096, 'a': 16841, 'o': 15739, 'n': 14063, '\n': 12797, 'i': 12528, 's': 12367, 'h': 12237, 'r': 11339, 'd': 9609, 'l': 8173, 'u': 5833, 'm': 5205, 'w':
```

By replacing characters in the original text, we have reduced the number of unique characters from 90 to 81.

```
# The unique characters in the file

#Instead of having the vocab letters in an arbitrary format, we can sort the letters based on frequency
#For example, 'Z' has the lowest frequency of all letters and will get assigned the index of 0

#Hopefully this will have an impact on the learning
vocab = sorted(char_freq, key=char_freq.get, reverse=False)
print ('{} unique characters'.format(len(vocab)))
#Heres the list of all characters that appear in the text
print(vocab)
```

```
81 unique characters
['Z', 'X', '7', '4', '8', 'Q', '2', '6', '*', '3', '5', '(', ')', 'U', '\t', '9', '1', '0', "'", 'V', 'K', 'R', '…', 'P', ':', ':', 'L', 'F', 'E', '!', 'O', 'J', 'z', 'q', 'N', 'C', 'Y', 'j',
```

Process the text

Vectorize the text

Before training, we need to map strings to a numerical representation. Create two lookup tables: one mapping characters to numbers, and another for numbers to characters.

```
# Creating a mapping from unique characters to indices
#Create a dictionary, i tracks the index of the char, u tracks the char
char2idx = dict((u,i) for i, u in enumerate(vocab))

#can access a char based on index
idx2char = np.array(vocab)

text_as_int = np.array([char2idx[c] for c in filtered_text])
#the corpus has been turned into an index for each char
text_as_int[500:600]

#print(repr(''.join(idx2char[text_as_int[500:600]])))                    #edit me
```

```
array([80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80,
       80, 80, 80, 53, 74, 74, 74, 74, 53, 75, 80, 66, 62, 80, 62, 76, 67,
       75, 64, 79, 70, 80, 77, 75, 69, 80, 66, 76, 70, 79, 80, 54, 67, 68,
       75, 79, 70, 77, 58, 68, 79, 80, 62, 79, 77, 70, 72, 80, 66, 62, 80,
       61, 77, 78, 71, 79, 70, 80, 64, 77, 54, 79, 80, 66, 79, 80, 72, 76,
       66, 79, 80, 77, 69, 54, 73, 63, 79, 74, 74, 78, 71, 77, 78])
```

We have changed the sequence length

We hope that by changing the sequence length to be longer, the LSTM will have more information to learn from.

Doing this increases the number of time steps in which the model has to make a prediction.

```
# The maximum length sentence we want for a single input in characters
seq_length = 100

examples_per_epoch = len(text)//(seq_length+1)
 TensorSliceDataset: char_dataset
 TensorSliceDataset with 277001 items  is good for streaming data
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

print(type(char_dataset))

#take() displays first x members of the tensor DataSet. For each element, we can get the data by calling .numpy() method
# we can then enter i into our lookup array to get the resultant character
for i in char_dataset.take(10):
  print(idx2char[i.numpy()])
  print(i)
```

```
<class 'tensorflow.python.data.ops.dataset_ops.TensorSliceDataset'>

tf.Tensor(14, shape=(), dtype=int64)

tf.Tensor(14, shape=(), dtype=int64)

tf.Tensor(14, shape=(), dtype=int64)

tf.Tensor(80, shape=(), dtype=int64)

tf.Tensor(80, shape=(), dtype=int64)

tf.Tensor(80, shape=(), dtype=int64)
T
tf.Tensor(49, shape=(), dtype=int64)
h
tf.Tensor(71, shape=(), dtype=int64)
e
tf.Tensor(79, shape=(), dtype=int64)

tf.Tensor(80, shape=(), dtype=int64)
```



```
#group the digits into strings of length 101
sequences = char_dataset.batch(seq_length+1, drop_remainder=True)

#Heres an example of changing a sequence of numbers back into alphanumeric chars
for item in sequences.take(5):
  print(repr(''.join(idx2char[item.numpy()])))
```

```
'\t\t\t   The Great Gatsby\n\n\t\t\t\t  by\n\n\t\t\t F. Scott Fitzgerald\n\n\n\n\n\n                          Table of Co'
'ntents\n\n\n\nI\n\nII\n\nIII\n\nIV\n\nV\n\nVI\n\nVII\n\nVIII\n\nIX\n\n\n\n\n\n                          Once again\n\n       '
'                          to\n\n                                  Zelda\n\n\n\n  Then wear the gold hat, if'
' that will move her;\n\n  If you can bounce high, bounce for her too,\n\n  Till she cry “Lover, gold-hatt'
'ed, high-bouncing lover,\n\n  I must have you!”\n\n\n\n  Thomas Parke d’Invilliers\n\n\n\n\n\n                  '
```

For each sequence, duplicate and shift it to form the input and target text by using the map method to apply a simple function to each batch:

```
#Create a function to split the data
def split_input_target(chunk):
  input_text = chunk[:-1]
  target_text = chunk[1:]
  return input_text, target_text

#Create a mappedDataset object. For each item in this dataset, there is the input or training array and the output or testing array
dataset = sequences.map(split_input_target)

#We can see that the resultant is 2 numpy arrays for each batch
# The first array is the input and the second is the target
for i in dataset.take(1):
  print(i)
```

```
(<tf.Tensor: shape=(100,), dtype=int64, numpy=
array([14, 14, 14, 80, 80, 80, 49, 71, 79, 80, 45, 70, 79, 77, 78, 80, 45,
       77, 78, 72, 58, 62, 74, 74, 14, 14, 14, 80, 80, 58, 62, 74, 74,
       14, 14, 14, 80, 27, 60, 80, 43, 63, 76, 78, 78, 80, 27, 73, 78, 32,
       64, 79, 70, 77, 68, 69, 74, 74, 74, 74, 74, 80, 80, 80, 80, 80,
       80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80,
       80, 80, 80, 80, 80, 49, 77, 58, 68, 79, 80, 76, 61, 80, 35])>, <tf.Tensor: shape=(100,), dtype=int64, numpy=
array([14, 14, 80, 80, 80, 49, 71, 79, 80, 45, 70, 79, 77, 78, 80, 45, 77,
       78, 72, 58, 62, 74, 74, 14, 14, 14, 80, 80, 58, 62, 74, 74, 14,
       14, 14, 80, 27, 60, 80, 43, 63, 76, 78, 78, 80, 27, 73, 78, 32, 64,
       79, 70, 77, 68, 69, 74, 74, 74, 74, 74, 80, 80, 80, 80, 80, 80,
       80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80,
       80, 80, 80, 80, 49, 77, 58, 68, 79, 80, 76, 61, 80, 35, 76])>)
```

Print the first examples input and target values:

Create training batches

We used tf.data to split the text into manageable sequences. But before feeding this data into the model, we need to shuffle the data and pack it into batches.

We added the .repeat() helper to the end of the shuffle method

This will allow the model to repeatable and so it will never end. This gives us a huge advantage as Deep Learning needs to have lots of data. This allows us to increase our Steps per epoch in the model.

```
[12] # Batch size
     BATCH_SIZE = 64

     # Buffer size to shuffle the dataset
     # (TF data is designed to work with possibly infinite sequences,
     # so it doesn't attempt to shuffle the entire sequence in memory. Instead,
     # it maintains a buffer in which it shuffles elements).
     BUFFER_SIZE = 10000

     #Add the repeat() helper so that the dataset is endless
     dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True).repeat()

     dataset

     <RepeatDataset element_spec=(TensorSpec(shape=(64, 100), dtype=tf.int64, name=None), TensorSpec(shape=(64, 100), dtype=tf.int64, name=None))>
```

icp6_text_gen.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
RAM Disk Editing

embedding_dim dimensions;

tf.keras.layers.GRU: A type of RNN with size units=rnn_units (You can also use a LSTM layer here.)

tf.keras.layers.Dense: The output layer, with vocab_size outputs.

```
     # Length of the vocabulary in chars
     vocab_size = len(vocab)

     # The embedding dimension
     embedding_dim = 256

     # Number of RNN units
     rnn_units = 1024                                                    #KF: we could increase this
```

Adding a LSTM layer LSTM has a memory cell which can hold information in memory for a longer period of time. A set of gates is used to control when information enters the memory, when it's output, and when it's forgotten. We could have used the GRU but the problem with GRU is that it doesnt have the seperate memory cell and they have fewer gate cells.

Adding a Dropuout layer We have added a Dropout Layer of Dropout(0.1) which means that 10% of data will be dropped out which aims to decrease the prossibility of overfitting ,thus we can decrease the loss and make a better prediction.

```
[14] def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
       model = tf.keras.Sequential([
         #The embedding layer helps to add semantic meaning to the input
         #With embedding we can do word math such as   King - man + woman = Queen
         tf.keras.layers.Embedding(vocab_size, embedding_dim,
                                   batch_input_shape=[batch_size, None]),
         # The LSTM layer brings in the concept of cell state. These cell states are
         # how the network remembers what has previously been entered
         tf.keras.layers.LSTM(rnn_units,
                              return_sequences=True,
                              stateful=True,
                              recurrent_initializer='glorot_uniform'),
         # The dropout layer prevents overfitting
         tf.keras.layers.Dropout(0.1) ,
         # The dense layer is where the model decides what to output
         tf.keras.layers.Dense(vocab_size)

       ])
       return model
```

Here we build the model with the chosen params

```
[15] model = build_model(
       vocab_size = len(vocab),
       embedding_dim=embedding_dim,
       rnn_units=rnn_units,
       batch_size=BATCH_SIZE)
```

Try the model

Now run the model to see that it behaves as expected.

First check the shape of the output:

```
[16] for input_example_batch, target_example_batch in dataset.take(1):
       example_batch_predictions = model(input_example_batch)
       print(example_batch_predictions.shape, "# (batch_size, sequence_length, vocab_size)")
```

```
(64, 100, 81) # (batch_size, sequence_length, vocab_size)
```

In the above example, the sequence length of the input is **XXXXXX** but the model can be run on inputs of any length:

```
# We can summarize the model to get information about the layers
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (64, None, 256)           20736

 lstm (LSTM)                 (64, None, 1024)          5246976

 dropout (Dropout)           (64, None, 1024)          0

 dense (Dense)               (64, None, 81)            83025

=================================================================
Total params: 5,350,737
Trainable params: 5,350,737
Non-trainable params: 0
_________________________________________________________________
```

To get actual predictions from the model we need to sample from the output distribution, to get actual character indices. This distribution is defined by the logits over the character vocabulary.

Note: It is important to sample from this distribution as taking the argmax of the distribution can easily get the model stuck in a loop.

Try it for the first example in the batch:

```
[18] # We have not actually run the model, so this information will be gibberish
     # However it helps us to understand what is going on in the model
     # We use random so that the model does not output the same thing for similar inputs
     sampled_indices = tf.random.categorical(example_batch_predictions[0], num_samples=1)
     sampled_indices = tf.squeeze(sampled_indices,axis=-1).numpy()

     #This gives us, at each timestep, a prediction of the next character index:
     sampled_indices
```

```
array([27, 37, 68, 76,  9, 63, 54, 28, 71, 23, 32, 50, 64, 66, 19, 40, 48,
       57, 71, 67, 38, 60, 22, 29, 55, 80, 60,  7, 74, 29, 49, 56,  0, 58,
       41, 37, 22, 70, 54, 57, 43, 74, 35, 32, 31, 41, 71, 49, 12, 44,  1,
       12, 42, 79, 46, 58, 77, 44, 69, 70, 60, 80, 39, 50,  2, 77, 55, 16,
       46, 80, 50, 31, 14, 75, 22, 35, 23, 44, 80, 70, 72, 59,  8, 64, 19,
       32, 73, 21,  4, 19, 17,  6,  6, 78, 11, 67, 76, 28, 62, 41])
```


Train the model

At this point the problem can be treated as a standard classification problem. Given the previous RNN state, and the input this time step, predict the class of the next character.

Attach an optimizer, and a loss function The standard tf.keras.losses.sparse_categorical_crossentropy loss function works in this case because it is applied across the last dimension of the predictions.

Because our model returns logits, we need to set the from_logits flag.

```
# Define the loss function, we use sparse because our data is not one hot encoded
def loss(labels, logits):
  return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

# Before we run the model, lets see what the loss output will look like
example_batch_loss  = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_size, sequence_length, vocab_size)")
print("scalar_loss:      ", example_batch_loss.numpy().mean())
```

```
Prediction shape:  (64, 100, 81)  # (batch_size, sequence_length, vocab_size)
scalar_loss:       4.39246
```

Configure the training procedure using the tf.keras.Model.compile method. We'll use tf.keras.optimizers.Adam with default arguments and the loss function.

```
[20] # Compile the model with the adam optimizer
     model.compile(optimizer='adam', loss=loss)
```

Configure checkpoints

Use a tf.keras.callbacks.ModelCheckpoint to ensure that checkpoints are saved during training:

Configure checkpoints

Use a tf.keras.callbacks.ModelCheckpoint to ensure that checkpoints are saved during training:

```
[21] # Directory where the checkpoints will be saved
     checkpoint_dir = './training_checkpoints'
     # Name of the checkpoint files
     checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

     checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(
         filepath=checkpoint_prefix,
         save_weights_only=True)
```

We also define an early stopping callback. This callback will allow the model to stop if the loss does not keep improving.

With this callback, we can use large number of epochs but the model will stop after the optimal loss is found.

```
[22] early_stopping_callback = tf.keras.callbacks.EarlyStopping(
         patience=5,
         monitor='loss',
         restore_best_weights=True,
         verbose=1
     )
```

Execute the training

To keep training time reasonable, use 10 epochs to train the model.

+ Code + Text

Changing Epoch size from 10 to 30.

Increasing the epoch size aims to decrease the loss. The increased epoch size is to provide more iterations with our dataset.

Increased epochs allow for more learning to be done.

```
[23] EPOCHS = 30
     INITIAL_EPOCH = 1
     STEPS_PER_EPOCH = 200
```

+ Code + Text RAM Disk Editing

```
[24] Epoch 28/30
     200/200 [==============================] - 12s 61ms/step - loss: 0.1812
     Epoch 29/30
     200/200 [==============================] - 12s 61ms/step - loss: 0.1788
     Epoch 30/30
     200/200 [==============================] - 12s 60ms/step - loss: 0.1761
```

Generate text

Restore the latest checkpoint To keep this prediction step simple, use a batch size of 1.

Because of the way the RNN state is passed from timestep to timestep, the model only accepts a fixed batch size once built.

To run the model with a different batch_size, we need to rebuild the model and restore the weights from the checkpoint.

```
tf.train.latest_checkpoint(checkpoint_dir)
model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)

model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

model.build(tf.TensorShape([1, None]))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding_1 (Embedding)      (1, None, 256)            20736

 lstm_1 (LSTM)                (1, None, 1024)           5246976

 dropout_1 (Dropout)          (1, None, 1024)           0

 dense_1 (Dense)              (1, None, 81)             83025

=================================================================
Total params: 5,350,737
Trainable params: 5,350,737
Non-trainable params: 0
_________________________________________________________________
```

```
Non-trainable params: 0
_________________________________________________________________
```

The prediction loop

The following code block generates the text:

It Starts by choosing a start string, initializing the RNN state and setting the number of characters to generate.

Get the prediction distribution of the next character using the start string and the RNN state.

Then, use a categorical distribution to calculate the index of the predicted character. Use this predicted character as our next input to the model.

The RNN state returned by the model is fed back into the model so that it now has more context, instead than only one character. After predicting the next character, the modified RNN states are again fed back into the model, which is how it learns as it gets more context from the previously predicted characters.

Looking at the generated text, you'll see the model knows when to capitalize, make paragraphs and imitates a Shakespeare-like writing vocabulary. With the small number of training epochs, it has not yet learned to form coherent sentences.

```
def generate_text(model, start_string):
  # Evaluation step (generating text using the learned model)

  # Number of characters to generate
  num_generate = 1000

  # Converting our start string to numbers (vectorizing)
  input_eval = [char2idx[s] for s in start_string]
  input_eval = tf.expand_dims(input_eval, 0)

  # Empty string to store our results
  text_generated = []

  # Low temperatures results in more predictable text.
  # Higher temperatures results in more surprising text.
  # Experiment to find the best setting.
  temperature = 1.0
```

✓ 8s completed at 6:31 PM

+ Code + Text RAM Disk Editing

```
[26]
  # Here batch size == 1
  model.reset_states()
  for i in range(num_generate):
    predictions = model(input_eval)
    # remove the batch dimension
    predictions = tf.squeeze(predictions, 0)

    # using a categorical distribution to predict the character returned by the model
    predictions = predictions / temperature
    predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()

    # We pass the predicted character as the next input to the model
    # along with the previous hidden state
    input_eval = tf.expand_dims([predicted_id], 0)

    text_generated.append(idx2char[predicted_id])

  return (start_string + ''.join(text_generated))
```

```
print(generate_text(model, start_string="The Great Gatsby of New York had a problem. His friend Nick was in love with his wife. He decided to "))
```

```
The Great Gatsby of New York had a problem. His friend Nick was in love with his wife. He decided to call to him. Miss Baker had been cold, but I meet around and more.

“I’m not like him,” said Mr. Wolfshiem, shaking my hand

earnestly, “and what do you think I did?”


“Go what?” demanded Tom.


“Two studies. One of them I call Michaelis had to ask that it was already behind him, somewhere back in that

vast obscurity beyond the city, where there warn in are
man can store up in his ghostly telaph, in a wasteback to play from his Proke

to an contraltion was distasteful to Gatsby.


“I don’t think this afternoon.”


“Very well. I’m too him that the thing that we remained to the book, reading each item aloud and then
```

✓ 8s completed at 6:31 PM