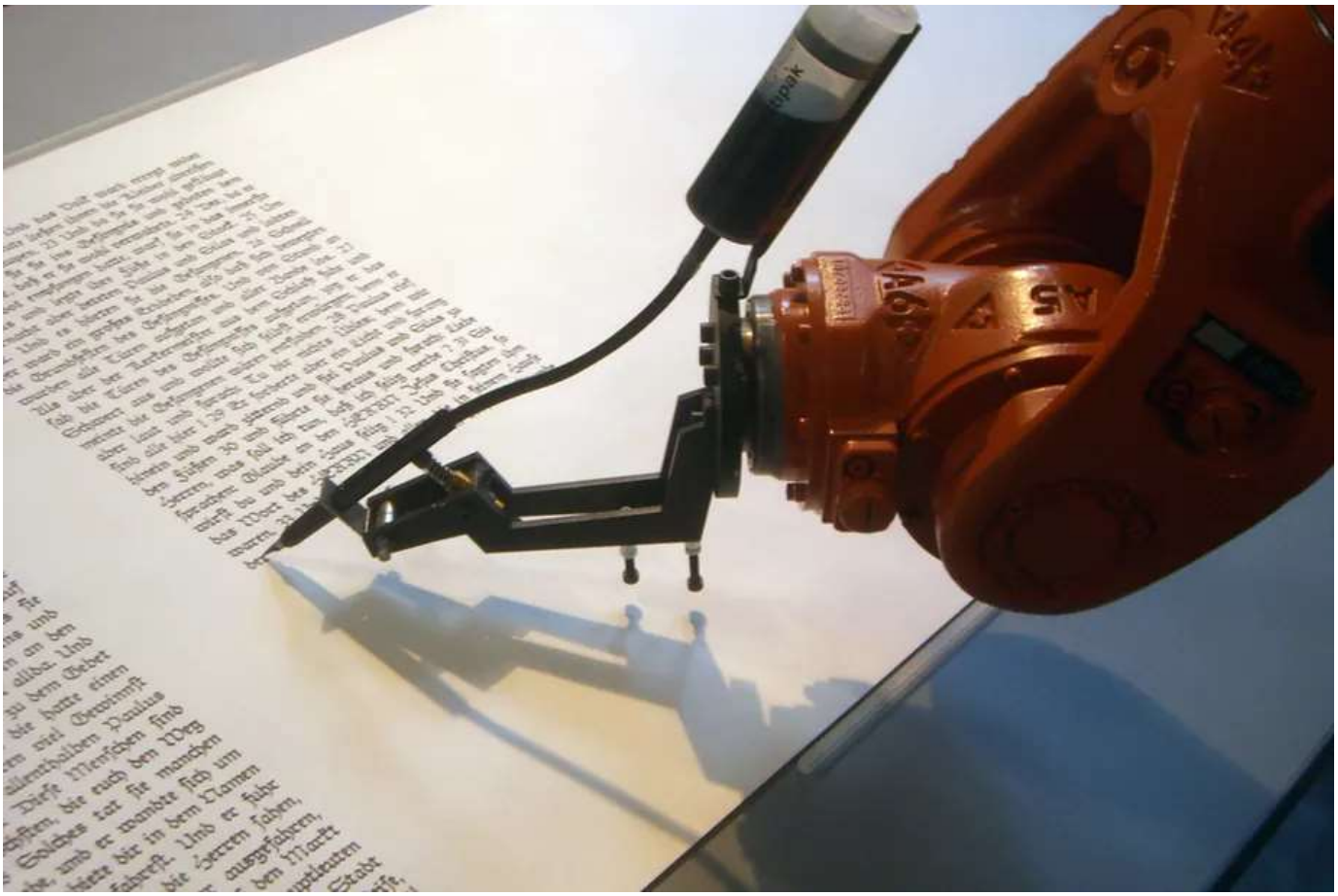
Text generation with an RNN

From short stories to writing 50,000 word novels, machines are churning out words like never before. There are tons of examples available on the web where developers have used machine learning to write pieces of text, and the results range from the absurd to delightfully funny.

Thanks to major advancements in the field of Natural Language Processing (NLP), machines are able to understand the context and spin up tales all by themselves.



To we will demonstrate how to generate text using a character-based RNN. You will work with a dataset of Shakespeare's writing from Andrej Karpathy's. Given a sequence of characters from this data ("Shakespear"), train a model to predict the next character in the sequence ("e"). Longer sequences of text can be generated by calling the model repeatedly.

This tutorial includes runnable code implemented using tf.keras. The following is sample output when the model in this tutorial trained for 10 epochs, and started with the string "Q":

```
QUEENE:
I had thought thou hadst a Roman; for the oracle,
Thus by All bids the man against the word,
Which are so weak of care, by old care done;
Your children were in your holy love,
```

```
And the precipitation through the bleeding throne.

BISHOP OF ELY:
Marry, and will, my lord, to weep in such a one were prettiest;
Yet now I was adopted heir
Of the world's lamentable day,
To watch the next way with his father with his face?

ESCALUS:
The cause why then we are all resolved more sons.

VOLUMNIA:
O, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, it is
And love and pale as any will to that word.

QUEEN ELIZABETH:
But how long have I heard the soul for this world,
And show his hands of life be proved to stand.

PETRUCHIO:
I say he look'd on, if I must be content
To stay him from the fatal of our country's bliss.
His lordship pluck'd from this sentence then for prey,
And then let us twain, being the moon,
were she such a case as fills m
```

While some of the sentences are grammatical, most do not make sense. The model has not learned the meaning of words, but consider:

- The model is character-based. When training started, the model did not know how to spell an English word, or that words were even a unit of text.
- The structure of the output resembles a play—blocks of text generally begin with a speaker name, in all capital letters similar to the dataset.
- As demonstrated below, the model is trained on small batches of text (100 characters each), and is still able to generate a longer sequence of text with coherent structure.

Import TensorFlow and other libraries

- TensorFlow is an artificial intelligence library that allows developers to create large-scale multi-layered neural networks.
- Importing numpy for working with arrays.
- The OS module in Python provides functions for interacting with the operating system.
- Time which allows us to handle various operations regarding time

```
#importing above mentioned libraries
```

Download the Shakespeare dataset

Change the following line to run this code on your own data.

Download the dataset:
https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt

using keras.utils.get_file function
https://www.tensorflow.org/api_docs/python/tf/keras/utils/get_file

```
#using keras.utils.get_file to download the dataset
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/sha
1122304/1115394 [==============================] - 0s 0us/step
1130496/1115394 [==============================] - 0s 0us/step
```

Read the data

First, look in the text:

```
# Read, then decode for py2 compat.

# length of text is the number of characters in it
```

```
Length of text: 1115394 characters
```

Now we will look the data

```
# Take a look at the first 250 characters in text
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.
```

```
First Citizen:
First, you know Caius Marcius is chief enemy to the people.
```

The unique characters in the file

```
# The unique characters in the file
#we will sort the text
#vocab = sorted(set(text))

#print unqiues characters.
```

```
65 unique characters
```

Process the text

Vectorize the text

- The code creates a mapping from unique characters to indices.
- The code then creates an array of integers, which is the index for each character in the vocabulary.

```
# Creating a mapping from unique characters to indices
#iterate over the vocab

#convert into array

#create array of integers [text_as_int = np.array([char2idx[c] for c in text])]
```

- The code starts by printing a string of characters.
- Then it iterates over the range from 0 to len(unique), printing each character and its index in the string.
- The code prints out a string of unique characters, each representing the index of the character in the string.

```
#iterate over range  into range of 50
#for char,_ in zip(char2idx, range(50)):

    #print thec char and char2idx
```

```
  '\n':   0,
```

```
  ' ' :   1,
  '!' :   2,
  '$' :   3,
  '&' :   4,
  "'" :   5,
  ',' :   6,
  '-' :   7,
  '.' :   8,
  '3' :   9,
  ':' :  10,
  ';' :  11,
  '?' :  12,
  'A' :  13,
  'B' :  14,
  'C' :  15,
  'D' :  16,
  'E' :  17,
  'F' :  18,
  'G' :  19,
  'H' :  20,
  'I' :  21,
  'J' :  22,
  'K' :  23,
  'L' :  24,
  'M' :  25,
  'N' :  26,
  'O' :  27,
  'P' :  28,
  'Q' :  29,
  'R' :  30,
  'S' :  31,
  'T' :  32,
  'U' :  33,
  'V' :  34,
  'W' :  35,
  'X' :  36,
  'Y' :  37,
  'Z' :  38,
  'a' :  39,
  'b' :  40,
  'c' :  41,
  'd' :  42,
  'e' :  43,
  'f' :  44,
  'g' :  45,
  'h' :  46,
  'i' :  47,
  'j' :  48,
  'k' :  49,
```

The prediction task

Given a character, or a sequence of characters, what is the most probable next character? This is the task you're training the model to perform. The input to the model will be a sequence of

characters, and you train the model to predict the output—the following character at each time step.

Since RNNs maintain an internal state that depends on the previously seen elements, given all the characters computed until this moment, what is the next character?

Create training examples and targets

Next divide the text into example sequences. Each input sequence will contain `seq_length` characters from the text.

For each input sequence, the corresponding targets contain the same length of text, except shifted one character to the right.

So break the text into chunks of `seq_length+1`. For example, say `seq_length` is 4 and our text is "Hello". The input sequence would be "Hell", and the target sequence "ello".

To do this first use the `tf.data.Dataset.from_tensor_slices` function to convert the text vector into a stream of character indices.

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

#examples per epoch=len(text)//(seq_length+1)


# Create training examples / targets
#using from data.Dataset.from_tensor_slices(text_as_int)


#iterate over dataset in range of 10.

    #print(idx2char[i.numpy()])


     F
     i
     r
     s
     t

     C
     i
     t
     i


#iterate over dataset in range of 10.

    #print(i.numpy)


     18
```

```
47
56
57
58
1
15
47
58
47
```

The batch method lets us easily convert these individual characters to sequences of the desired size. The repr() function returns the string representation of the value passed to eval function by default.

```
#creating batch of seq_length


#iterate over 5 range [for item in sequences.take(5):]

    #print and join idx2char from item.numpy()
    #print(repr(''.join(idx2char[item.numpy()])))
```

```
'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak
'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\n\nFirst
"now Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we know't.\n\nFi
"ll him, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo more talk
'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citizen:\nWe
```

- The code splits the input text into two parts: an input string and a target string.
- The split_input_target function takes in a chunk of text as its first parameter, which is then used to create the input string and target string.
- The code splits the input text into two parts, the first part is the input text and the second part is target text.

```
#define a function with chunk as parameter

    #input text=chunk[:-1]

    #target text=chunk[1:]

    #return input_text, target_text

#For each sequence, duplicate and shift it to form the input and target text by using the
```

Print the first example input and target values:

- The code is trying to take the first element of a list and then print out what that data looks like. - The code is doing this by taking the input_example, which is a string, and converting it into an array with one column (the index) and one row (char).
- The target_example is also a string, but instead of being converted into an array it's just printed out as-is.
- The code prints the input and target data for each example in the dataset.

```
#iterate over the dataset  of take 1

    #printing the input data

    #printing the target data


     Input data:  'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\r
     Target data: 'irst Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\n
```

Each index of these vectors is processed as a one time step. For the input at time step 0, the model receives the index for "F" and tries to predict the index for "i" as the next character. At the next timestep, it does the same thing but the `RNN` considers the previous step context in addition to the current input character.

```
#for i, (input_idx, target_idx) in enumerate(zip(input_example[:5], target_example[:5])):

    #print i

    #print input_idx [print("  input: {} ({:s})".format(input_idx, repr(idx2char[input_idx

    #print("  expected output: {} ({:s})".format(target_idx, repr(idx2char[target_idx])))


     Step    0
       input: 18 ('F')
       expected output: 47 ('i')
     Step    1
       input: 47 ('i')
       expected output: 56 ('r')
     Step    2
       input: 56 ('r')
       expected output: 57 ('s')
     Step    3
       input: 57 ('s')
       expected output: 58 ('t')
     Step    4
       input: 58 ('t')
       expected output: 1 (' ')
```

Create training batches

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

```
# 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).


#dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
```

Build The Model

Use tf.keras.Sequential to define the model. For this simple example three layers are used to define our model:

- tf.keras.layers.Embedding: The input layer. A trainable lookup table that will map the numbers of each character to a vector with embedding_dim dimensions;
- tf.keras.layers.GRU: A type of RNN with size units=rnn_units (You can also use an LSTM layer here.)
- tf.keras.layers.Dense: The output layer, with vocab_size outputs.

```
# Length of the vocabulary in chars


# The embedding dimension


# Number of RNN units
```

- The code builds a model that has an embedding layer, a GRU layer, and a dense layer.
- The code iterates over the batch size to create one sequence of input data for each iteration.
- The code starts by creating an EmbeddingLayer with the vocabulary size and embedding dimensionality.
- It then creates a GRU Layer which takes in sequences as inputs and returns sequences as outputs.
- Finally it creates a DenseLayer with the same vocabulary size as the previous layers so that they can be stacked on top of each other to form one large neural network.

- The code creates a model that has an embedding layer with the size of vocab_size, and GRU layers with rnn_units.
- The batch input shape is [batch_size, None].
- The last layer in the model is a Dense layer which has the size of vocab_size.

```
#define a function with vocab_size, embedding_dim, rnn_units, batch_size

    #create model sequential

        #embedding layer with vocab_size, embedding_dim,batch_input_shape=[batch_size, Non


        #GRU layer with rnn_units,return_sequences=True,stateful=True,recurrent_initialize



        #dense layers with vocab size


    #return the model
```

- The code starts by declaring the model and its parameters. - The first parameter is the size of the vocabulary, which in this case is set to be len(vocab) .
- Next, we have a dimension for our embedding space that will be set to be embedding_dim .
- Finally, we have rnn_units , which tells us how many units are in each layer of our RNN network. - The code then creates an instance of build_model() with those values. - This function takes three arguments: vocab_size , embedding_dim , and rnn_units .
- It returns a model object that can then be used as input into other functions like fit() or predict()

```
#intialize the model
#model= build model

    #giving vocab length

    #giving embedding dimensions

    #rnn_units=rnn_units

    #batch size=batch size
```

For each character the model looks up the embedding, runs the GRU one timestep with the embedding as input, and applies the dense layer to generate logits predicting the log-likelihood of the next character

Try the model

- The code starts by creating a list of lists called example_batch_predictions. - This is the list that will be used to store the predictions for each batch in the input dataset.
- The code then prints out this list, which contains two columns: # (batch_size, sequence_length, vocab_size).
- The first column (#) represents how many batches there are in the input dataset and their respective sizes.
- The second column (sequence length) represents how long each batch is and its size in characters.
- Finally, the third column (vocab size) represents how many unique words there are per batch and their respective sizes.
- The code creates a batch of examples and then trains the model on that batch. - The code prints out the input_example_batch, target_example_batch, and example_batch_predictions.

```
#iterate over dataset of take 1

    #example_batch_predictions = model(input_example_batch)

    #print the example_batch_predictions
```

```
[[-1.96485384e-03  1.78255350e-03  8.94756522e-05 ...  4.07895632e-03
   1.04601756e-02  1.31612401e-02]
 [-5.81071479e-04  2.33395770e-03  1.61856748e-02 ... -8.07785243e-03
   1.59236491e-02  7.71793118e-03]
 [-1.38272997e-04 -1.00779105e-02  6.35307934e-03 ... -1.60108414e-03
   1.38328718e-02  6.06756471e-03]
 ...
 [-8.21914524e-04  1.12408297e-02 -1.52672436e-02 ...  9.12214722e-03
   2.10789940e-03 -8.36632680e-03]
 [ 2.41937488e-03  2.85346853e-03 -1.39917685e-02 ...  1.16713336e-02
   9.35224816e-03 -4.21350403e-03]
 [ 1.50093166e-02  9.30098072e-03 -1.25947967e-02 ...  1.01170260e-02
   1.83977205e-02  9.40260477e-03]]

...

[[ 8.68798047e-03  1.13145793e-02 -2.96877720e-03 ...  4.31643799e-03
   1.24788377e-03 -5.09505626e-03]
 [ 4.94018756e-03  6.34226017e-03  1.38235968e-02 ... -9.89245158e-03
   1.24505805e-02 -3.74980830e-03]
 [ 4.35831584e-03  4.33360739e-03  2.03014538e-02 ... -1.65744331e-02
   1.65807828e-02 -3.53121338e-03]
 ...
 [-2.71280925e-03 -3.05695366e-03 -4.55850502e-03 ... -5.46512520e-03
   8.65035318e-03 -9.99659533e-04]
 [-3.64559330e-03  6.18741242e-03 -9.56539344e-03 ... -1.81274954e-04
   1.25026060e-02 -1.04528219e-02]
```

```
 [-6.16409443e-03  8.88865651e-03 -1.41350189e-02 ...  1.81990233e-03
   1.65529102e-02 -1.61389783e-02]]

 [[-2.64764391e-03  6.32730639e-03 -1.03329383e-02 ...  1.46063277e-03
   5.09178871e-03 -8.83869082e-03]
  [-1.22658592e-02 -2.58102128e-03  4.09572292e-03 ... -3.78617039e-03
   1.58928279e-02 -1.37493648e-02]
  [-2.49676667e-02  6.50973478e-03 -1.59513403e-03 ... -9.44718998e-03
   5.00438269e-04 -1.13879740e-02]
  ...
  [ 1.04702003e-02  1.00867078e-02  2.60599423e-04 ...  6.09889801e-04
   1.40370354e-02  8.14418960e-03]
  [ 6.62248814e-04 -1.00104194e-02  1.38778705e-04 ... -1.03664733e-02
   1.33410078e-02 -5.55688888e-03]
  [ 3.44776362e-03 -6.26655389e-03  2.45548226e-03 ... -1.21855959e-02
   2.00895173e-03 -4.56059398e-03]]

 [[-1.94945734e-03  4.75245528e-03 -4.61361976e-03 ...  1.18743174e-03
   4.49574227e-03  5.31799719e-03]
  [-6.10190956e-03  1.15359258e-02  4.03034221e-03 ...  6.28073653e-03
   5.56099880e-03 -1.97396753e-03]
  [ 4.49103629e-03  1.78115107e-02 -1.38927647e-03 ...  1.55257192e-02
  -2.30166130e-03  5.18175587e-03]
  ...
  [ 6.32132730e-03  8.29265825e-03 -1.01068998e-02 ...  5.13417413e-03
  -1.56294182e-02  4.32953518e-03]
  [-1.23841409e-03  1.19967246e-02  6.23297179e-04 ...  1.02194864e-02
  -4.62545501e-03 -2.15529464e-03]
  [ 7.36616598e-03  1.69860497e-02 -4.17990237e-03 ...  1.88896395e-02
```

In the above example the sequence length of the input is `100` but the model can be run on inputs of any length:

```
#print the model summary
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (64, None, 256)           16640

 gru (GRU)                   (64, None, 1024)          3938304

 dense (Dense)               (64, None, 65)            66625

=================================================================
Total params: 4,021,569
Trainable params: 4,021,569
Non-trainable params: 0
_________________________________________________________________
```

To get actual predictions from the model you 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:

- The code first creates a list of the predictions for each example.
- The code then randomly selects one prediction from this list and returns it as an array with 1 element.
- This is done by calling tf.random.categorical() .
- The code then takes the result of that call, which is an array with 1 element, and calls tf.squeeze() on it to remove any extra elements in the axis=1 dimension (the first dimension).

```
#sampled_indices = tf.random.categorical(example_batch_predictions[0], num_samples=1)

#now call the squeeze function  with sampled_indices,axis=-1).numpy()
#sampled_indices = tf.squeeze(sampled_indices,axis=-1).numpy()
```

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

```
#print the sampled_indices
```

```
array([45, 34,  1, 55, 12, 28, 21, 44, 13, 27,  1, 33, 33, 29, 36, 38, 47,
       18, 53, 37, 20, 52, 30, 45, 10,  9, 57, 62, 57, 41, 48, 22, 10, 46,
       15, 61, 21, 57, 17, 45, 59, 54, 54, 36, 52, 63,  0, 13, 63, 33, 47,
       25,  7, 40, 55, 30,  8, 45, 40, 51, 11, 35, 22, 13, 63,  4, 10, 41,
        8,  9, 43, 56,  4, 62, 23, 39, 50, 54, 30, 43, 33, 24, 44, 16, 43,
       24, 13, 20, 41,  2,  4,  0, 20, 45, 28, 54, 27,  9, 28, 36])
```

Decode these to see the text predicted by this untrained model:

- The code is meant to predict the next character in a string.
- The code starts by printing out the input and then prints out the next char predictions.

```
#pirnt the input
#print("Input: \n", repr("".join(idx2char[input_example_batch[0]])))
```

```
#print nexr cgar predictions
#print("Next Char Predictions: \n", repr("".join(idx2char[sampled_indices ])))
```

```
Input:
 "Now my soul's palace is become a prison:\nAh, would she break from hence, that this
```

```
Next Char Predictions:
 'gV q?PIfAO UUQXZiFoYHnRg:3sxscjJ:hCwIsEguppXny\nAyUiM-bqR.gbm;WJAy&:c.3er&xKalpReUl
```

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 your model returns logits, you need to set the `from_logits` flag.

```
#define a function with labels, logits

    #return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=Tr


#pass the loss with target_example_batch, example_batch_predictions

#print the prediction shape

#print the scalar loss
#print("scalar_loss:      ", example_batch_loss.numpy().mean())
```

```
Prediction shape:  (64, 100, 65)  # (batch_size, sequence_length, vocab_size)
scalar_loss:       4.174121
```

Configure the training procedure using the `tf.keras.Model.compile` method. Use `tf.keras.optimizers.Adam` with default arguments and the loss function.

```
#compling the model with adam optimizer and loss =loss
```

Configure checkpoints

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

```
# Directory where the checkpoints will be saved

# Name of the checkpoint files


#calling the check points
```

Execute the training

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

```
#epochs=10


#now we will fit our model
```

```
Epoch 1/10
172/172 [==============================] - 12s 51ms/step - loss: 2.6579
Epoch 2/10
172/172 [==============================] - 10s 51ms/step - loss: 1.9598
Epoch 3/10
172/172 [==============================] - 10s 51ms/step - loss: 1.6968
Epoch 4/10
172/172 [==============================] - 10s 52ms/step - loss: 1.5499
Epoch 5/10
172/172 [==============================] - 10s 52ms/step - loss: 1.4599
Epoch 6/10
172/172 [==============================] - 10s 52ms/step - loss: 1.3999
Epoch 7/10
172/172 [==============================] - 11s 53ms/step - loss: 1.3550
Epoch 8/10
172/172 [==============================] - 10s 52ms/step - loss: 1.3156
Epoch 9/10
172/172 [==============================] - 10s 52ms/step - loss: 1.2810
Epoch 10/10
172/172 [==============================] - 10s 53ms/step - loss: 1.2493
```

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`, you need to rebuild the model and restore the weights from the checkpoint.

```
#tf.train.latest_checkpoint(checkpoint_dir)
```

```
'./training_checkpoints/ckpt_10'
```

- The code will build a model with the specified parameters.
- The code above will load the weights from the latest checkpoint in the training folder.
- The code above will build a model with 1 input and None as an output shape.

```
#model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)


#load the weights
#model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))


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

The prediction loop

The following code block generates the text:

- Begin 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 of 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.

```
#define a function to genrate text with model, start_string

    # Evaluation step (generating text using the learned model)

    # Number of characters to generate
```

```
    #2000


    # Converting our start string to numbers (vectorizing)

    #input_eval = tf.expand_dims(input_eval, 0)


    # Empty string to store our results

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


    # Here batch size == 1
    #model.reset_states()

    #iterate over range of num_genrate
    #for i in range(num_generate):

        #predictions = model(input_eval)

        # remove the batch dimension


        # 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()


        # Pass the predicted character as the next input to the model
        # along with the previous hidden state


        #text_generated.append(idx2char[predicted_id])

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


#print the genrated text
```

```
QUKENTIO:
My brother Angell AUHERSO:
Midaimenly, go to? Camillo
Not only find and procure in: I
Am fire that heavy bouble; indevation.
Good man or ere thy king: my heart,
Look'd prick kine gouble consul, and let Richard the spurs.
How long!

DUKE OF AUMERLE:
Your queen.
```

```
QUEEN:
A blessed bodness beat, and treasure 'twas foreign he
ends, rest, that hearkily would I will how the issue here, must be
Extandath of the use Is, take on mine.

LUCIO:
I plund them and call thee, sir; the
news which sendsmands he man; angelo!
Longing and thing he made me with a thing ind sumpling nurse, the tear
That Dorest them speak against the villain and Ground:
Of our presence, unless the book of me than seem
My brother, that news?

Second God-delly nature.

ESCALUS:
The queen was crime these many feels of this allier;
And thou do stoler mewavy, that loos at their feather?

GLOUCESTER:
A blesser shall fell
Beaute and serve: he's murder, which, wis on rather have
Dott he him af enemy the friar?
This is a word may go most him that
Become the ble that will not gentlemen, only more than sevente, in on you are, loo
Which the proffed to God, though it makes palleath:
The noisure that to simple well about
The provost the grace of mine hands
In passagement that be here io.

LORD FIUS:
Thy brother likes a traitor; and:
Fortune, means, manamercome,
Whose seains--Nothing, is the lamb shall say to-day?

SLY:
Marry, and bite me be lulling. Would
your news we did low serve me for the sacred change;
Inget and rope this soul both good to hear me swear.

VIRGILIA:
O, no; then, with sluckled complates
If they drown'd, and then the more chieful game:
I'll undone thy thousand fire and go well!
Let his sue--that goes!
```

We have come to an end of this project but don't stop here, try as many projects of the similar type to get a better understanding of the use cases. Solve the practice sheet of this project to test yourself.!!

Great job!! You have come to the end of this assignment. Treat yourself for this :))

Do fill this feedback form

You may head on to the next assignment.