In [0]:

```
import tensorflow as tf
import numpy as np
import os
import time
```

In [2]:

```
path_to_file = tf.keras.utils.get_file('shakespeare.txt', 'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')
```

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

In [3]:

```
text = open(file= path_to_file).read()
print('Length of no of characters:', len(text),'\n')
print(text[:250])
```

```
Length of no of characters: 1115394

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.
```

In [4]:

```
# unique characters

vocab = sorted(set(text))
print('Length of vocab:', len(vocab),'\n')
vocab
```

```
Length of vocab: 65
```

Out[4]:

```
['\n',
 ' ',
 '!',
 '$',
 '&',
 "'",
 ',',
 '-',
 '.',
 '3',
 ':',
 ';',
 '?',
 'A',
 'B',
 'C',
 'D',
 'E',
 'F',
 'G',
 'H',
 'I',
 'J',
 'K',
```

```
 'K',
 'L',
 'M',
 'N',
 'O',
 'P',
 'Q',
 'R',
 'S',
 'T',
 'U',
 'V',
 'W',
 'X',
 'Y',
 'Z',
 'a',
 'b',
 'c',
 'd',
 'e',
 'f',
 'g',
 'h',
 'i',
 'j',
 'k',
 'l',
 'm',
 'n',
 'o',
 'p',
 'q',
 'r',
 's',
 't',
 'u',
 'v',
 'w',
 'x',
 'y',
 'z']
```

In [5]:

```
char_to_idx = {j:i for i , j in enumerate(vocab)}

char_to_idx
```

Out[5]:

```
{'\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,
'l': 50,
'm': 51,
'n': 52,
'o': 53,
'p': 54,
'q': 55,
'r': 56,
's': 57,
't': 58,
'u': 59,
'v': 60,
'w': 61,
'x': 62,
'y': 63,
'z': 64}
```

In [6]:

```
# change all characters to integers by the mapping of numbers of char_to_idx
print(text[:100], '\n')

text_to_int = np.array([char_to_idx[i] for i in text])
print(text_to_int[:100])
```

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

All:
Speak, speak.

First Citizen:
You

[18 47 56 57 58  1 15 47 58 47 64 43 52 10  0 14 43 44 53 56 43  1 61 43
  1 54 56 53 41 43 43 42  1 39 52 63  1 44 59 56 58 46 43 56  6  1 46 43
 39 56  1 51 43  1 57 54 43 39 49  8  0  0 13 50 50 10  0 31 54 43 39 49
  6  1 57 54 43 39 49  8  0  0 18 47 56 57 58  1 15 47 58 47 64 43 52 10
  0 37 53 59]
```

The prediction task

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

In [7]:

```
# This length is used to slice the text
seq_length= 100

examples_per_epoch = len(text) // (seq_length+1)
print('examples_per_epoch:', examples_per_epoch, '\n')

# Create examples slicing all of text individually to pass each character.
char_dataset = tf.data.Dataset.from_tensor_slices(tensors =  text_to_int)

# idx_to_char means index to char in np.array
idx_to_char = np.array(vocab)
print('idx_to_char:\n', idx_to_char, '\n')
print('First five elements:')
for i in char_dataset.take(count= 5):
  print(idx_to_char[i])
```

```
examples_per_epoch: 11043

idx_to_char:
 ['\n' ' ' '!' '$' '&' "'" ',' '-' '.' '3' ':' ';' '?' 'A' 'B' 'C' 'D' 'E'
 'F' 'G' 'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O' 'P' 'Q' 'R' 'S' 'T' 'U' 'V' 'W'
 'X' 'Y' 'Z' 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
 'p' 'q' 'r' 's' 't' 'u' 'v' 'w' 'x' 'y' 'z']

First five elements:
F
i
r
s
t
```

The batch method lets us easily convert these individual characters to sequences of the desired size.

drop_remainder= False

i/p --> dataset = tf.data.Dataset.range(8) dataset = dataset.batch(3)

o/p --> list(dataset.as_numpy_iterator()) [array([0, 1, 2]), array([3, 4, 5]), array([6, 7])]

drop_remainder= True (means truncate the last if the length of tensor doesnt match with existing tensors)

i/p --> dataset = tf.data.Dataset.range(8) dataset = dataset.batch(3, drop_remainder=True)

o/p --> list(dataset.as_numpy_iterator()) [array([0, 1, 2]), array([3, 4, 5])]

In [8]:

```
# seq_length + 1 is shown in below 'dataset'
# if seq_length= 4 and seq_length + 1 --> 'hello' = 'hell' and 'ello'
sequences = char_dataset.batch(batch_size= seq_length+1, drop_remainder= True)

for i in sequences.take(count= 2):
  print('Length of each sequence:',len(i))
  print("".join(idx_to_char[i]),'\n')
```

```
Length of each sequence: 101
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You

Length of each sequence: 101
are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you k
```

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:

In [9]:

```
def split_chunks(chunk):
  input_text = chunk[:-1]
  target_text= chunk[1:]
  return input_text, target_text

dataset = sequences.map(map_func= split_chunks)

# We print and check
for i, j in dataset.take(count= 1):
  print('Length of input sequence:', len(i), '\n')
  print('Input_text:', '\n', "".join(idx_to_char[i]), '\n')
  print('Length of target sequence:', len(j), '\n')
  print('Target_text:', '\n', "".join(idx_to_char[j]), '\n')
```

```
Length of input sequence: 100

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

All:
Speak, speak.

First Citizen:
You

Length of target sequence: 100

Target_text:
 irst Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You
```

Each index of these vectors are processed as one time step. For the input at time step 0, the model receives the index for "F" and trys 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.

In [10]:

```
for m, (n, o) in enumerate(zip(i[:5], j[:5])):
  print('Step:', m)
  print('Input:  {}, {} '.format(idx_to_char[n], n))
  print('Output: {}, {} '.format(idx_to_char[o], o),'\n')
```

```
Step: 0
Input:  F, 18
Output: i, 47

Step: 1
Input:  i, 47
Output: r, 56

Step: 2
Input:  r, 56
Output: s, 57

Step: 3
Input:  s, 57
Output: t, 58

Step: 4
Input:  t, 58
Output:  , 1
```

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

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.

Randomly shuffles the elements of this dataset.

This dataset fills a buffer with buffer_size elements, then randomly samples elements from this buffer, replacing the selected elements with new elements. For perfect shuffling, a buffer size greater than or equal to the full size of the dataset is required.

For instance, if your dataset contains 10,000 elements but buffer_size is set to 1,000, then shuffle will initially select a random element from only the first 1,000 elements in the buffer. Once an element is selected, its space in the buffer is replaced by the next (i.e. 1,001-st) element, maintaining the 1,000 element buffer.

In [11]:

```
# Batch size
batch_size= 64

steps_per_epoch = examples_per_epoch//batch_size
# 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

dataset = dataset.shuffle(buffer_size= buffer_size)
dataset = dataset.batch(batch_size= batch_size, drop_remainder= True)

dataset
```

Out[11]:

<BatchDataset shapes: ((64, 100), (64, 100)), types: (tf.int64, tf.int64)>

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 a LSTM layer here.)
> tf.keras.layers.Dense: The output layer, with vocab_size outputs.

In [0]:

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

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

In [13]:

```
# This is identical to the following:
# model = tf.keras.Sequential()
# model.add(tf.keras.layers.Dense(8, input_dim=16)) and so on...

def model_build(vocab_size, embedding_dim, rnn_units, batch_size):
  model = tf.keras.Sequential(layers=[tf.keras.layers.Embedding(input_dim= vocab_size, output_dim= embedding_dim,
                                      batch_input_shape= [batch_size, None]),
                       tf.keras.layers.LSTM(units= rnn_units, return_sequences= True, stateful= True,
                                   recurrent_initializer= 'glorot_uniform'),
                       tf.keras.layers.Dense(units= vocab_size,) #  activation= 'softmax' not used and logits are used
                       ] )
  return model

model = model_build( vocab_size = vocab_size, embedding_dim=embedding_dim, rnn_units=rnn_units, batch_size= batch_size)

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
```

```
embedding (Embedding)        (64, None, 256)           16640
_________________________________________________________________
lstm (LSTM)              (64, None, 1024)        5246976
_________________________________________________________________
dense (Dense)              (64, None, 65)           66625
=================================================================
Total params: 5,330,241
Trainable params: 5,330,241
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.

In [14]:

```
for ip, op in dataset.take(count= 1):
  # pass input into model and we get 1st batch shape as
  example_batch = model(ip)
  # https://stackoverflow.com/a/15181942/10219869
  print('batch size: {0}, sequence length: {1}, vocab_size: {2} '.format(*example_batch.shape))
```

```
batch size: 64, sequence length: 100, vocab_size: 65
```

In [15]:

```
# we try a sample prediction
# example_batch[0] means first row of 100 out of 64 rows.
# we obtain indices

indices = tf.random.categorical(example_batch[0], num_samples= 1)
# https://stackoverflow.com/a/58843005/10219869 to understand squeeze
indices = tf.squeeze(indices, axis= -1)
indices
```

Out[15]:

```
<tf.Tensor: shape=(100,), dtype=int64, numpy=
array([36, 27, 50,  8, 22, 11, 52, 41, 21, 53, 56, 44, 50, 29, 31, 52, 12,
       29,  8, 45, 16, 50, 31,  0, 29, 54,  7, 51,  8, 40,  7, 64,  4, 53,
       31, 29, 49, 64, 54, 29, 27, 38, 16, 52, 61, 13,  4,  9,  5,  2, 32,
        7,  8, 44,  2,  3, 30, 38, 20, 38, 20, 11, 40, 24, 57, 39, 58, 51,
       13, 36, 38, 37, 60, 62, 63, 12, 43, 23,  6, 56, 56, 19, 48, 28, 32,
       48, 45, 23, 57, 16, 14, 42, 26, 64,  0, 63, 41, 52, 63, 20])>
```

In [16]:

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

print('Input:')
# without join, the output gets separated
print(''.join(idx_to_char[ip[0]]), '\n')
print('Next char predictions:')
print(''.join(idx_to_char[indices]))
```

```
Input:
ess than I was born to:
A man at least, for less I should not be;
And men may talk of kings, and why

Next char predictions:
XOl.J;nclorflQSn?Q.gDlS
Qp-m.b-z&oSQkzpQOZDnwA&3'!T-.f!$RZHZH;bLsatmAXZYvxy?eK,rrGjPTjgKsDBdNz
ycnyH
```

In [17]:

```
example_loss= tf.keras.losses.sparse_categorical_crossentropy(y_true= op, y_pred= example_batch, from_logits= True )

example_loss.numpy().mean()
```

Out[17]:

```
4.173674
```

In [0]:

```
checkpoint=tf.keras.callbacks.ModelCheckpoint( filepath= '/content/rnn.h5', save_weights_only=True, )
```

In [0]:

```
def loss(labels, logits):
  return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

model.compile(optimizer= 'rmsprop', loss = loss)
```

In [20]:

```
epoch= 5

history = model.fit(dataset.repeat(), epochs= epoch, steps_per_epoch= steps_per_epoch, callbacks= checkpoint)
```

```
Epoch 1/5
172/172 [==============================] - 9s 55ms/step - loss: 2.6894
Epoch 2/5
172/172 [==============================] - 9s 54ms/step - loss: 1.9237
Epoch 3/5
172/172 [==============================] - 9s 55ms/step - loss: 1.6474
Epoch 4/5
172/172 [==============================] - 9s 55ms/step - loss: 1.5084
Epoch 5/5
172/172 [==============================] - 9s 55ms/step - loss: 1.4262
```

In [21]:

```
model = model_build(vocab_size, embedding_dim, rnn_units, batch_size=1)

model.load_weights('/content/rnn.h5')

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

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (1, None, 256)            16640
_________________________________________________________________
lstm_1 (LSTM)                (1, None, 1024)           5246976
_________________________________________________________________
dense_1 (Dense)              (1, None, 65)             66625
=================================================================
Total params: 5,330,241
Trainable params: 5,330,241
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.

In [22]:

```
# Low temperatures results in more predictable text.
# Higher temperatures results in more surprising text.
```

```python
def generate_text(model, start_string, temp):
  # Number of characters to generate
  num_generate = 1000

  # converting start string to numbers
  input_eval = [char_to_idx[s] for s in start_string]
  print(input_eval)

  # This operation is useful if you want to add a batch dimension to a single element.
  # For example, if you have a single image of shape [height, width, channels],
  # you can make it a batch of one image with expand_dims(image, 0), which will make the shape [1, height, width, channels].
  input_eval = tf.expand_dims(input_eval, 0)
  print(input_eval)

  # empty string to store results
  text_generated = []

  # Experiment to find the best setting. (0-1)
  temperature = temp

  # here batch size = 1
  model.reset_states()
  for i in range(num_generate):
    predictions = model(input_eval)

    # remove batch dimentions
    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(idx_to_char[predicted_id])

  return (start_string + ''.join(text_generated))
print(generate_text(model, start_string=u"ROMEO: ", temp= 0.5))
```

```
[30, 27, 25, 17, 27, 10, 1]
tf.Tensor([[30 27 25 17 27 10  1]], shape=(1, 7), dtype=int32)
ROMEO: I must not speak,
And do not scorn to me the could shall not say
That shall be many than the manner to dischanged,
And say I would so Marcius, now in that wind
Than what will be made a vonced for me.

JULIET:
Ay, in the condemn'd with him on most surmand.

KING RICHARD III:
What is the enemies.

DUKE VINCENTIO:
I will not be so dismand, and when I shall be so;
The land and so revenge him a thousand country's hand,
I will not think and pack'd him and look'd to his soul.

LUCIO:
What is the signior days shall be so.

CAPULET:
How now, my soul boy, What thou art not fould not so so.

BRUTUS:
Shall I be so.

Second Keeper:
What am money.

SICINIUS:
What had the waste of love, and in die to be some consul.

DUKE VINCENTIO:
Here comes to come and wife.

DUKE VINCENTIO:
Here's no servant, and how the light and more
Shall be so done. I will make a desport?

KING RICHARD III:
So find many more hands, and depart mine.
```

So find many more hands, and depart mine.

KING RICHARD III:
Then shall be married to him a borthous more than the day.

In [23]:

```
# Low temperatures results in more predictable text.
# Higher temperatures results in more surprising text.

def generate_text(model, start_string, temp):
  # Number of characters to generate
  num_generate = 1000

  # converting start string to numbers
  input_eval = [char_to_idx[s] for s in start_string]
  print(input_eval)

  # This operation is useful if you want to add a batch dimension to a single element.
  # For example, if you have a single image of shape [height, width, channels],
  # you can make it a batch of one image with expand_dims(image, 0), which will make the shape [1, height, width, channels].
  input_eval = tf.expand_dims(input_eval, 0)
  print(input_eval)

  # empty string to store results
  text_generated = []

  # Experiment to find the best setting. (0-1)
  temperature = temp

  # here batch size = 1
  model.reset_states()
  for i in range(num_generate):
      predictions = model(input_eval)

      # remove batch dimentions
      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(idx_to_char[predicted_id])

  return (start_string + ''.join(text_generated))
print(generate_text(model, start_string=u"ROMEO: ", temp= 0.1))
```

[30, 27, 25, 17, 27, 10, 1]
tf.Tensor([[30 27 25 17 27 10 1]], shape=(1, 7), dtype=int32)
ROMEO: I will be so so.

PETRUCHIO:
What is the cause?

CAPULET:
What may come to the contract of his hands.

KING RICHARD III:
Why, then a fair of the company.

KING RICHARD III:
And what a son of some of your honour.

KING RICHARD III:
What is the condemn'd with a grave and the prince,
And so did not so dispatch and soldier,
And therefore the consul to the fair and disposed to him.

KING RICHARD III:
What is the manner of the warst of his father's life.

KING RICHARD III:
What is the consul, and the manner of him.

KING RICHARD III:
What is the state of his friends and the worst.

KING RICHARD III:
Why, then I say, the consul's days and more than the state

Why, then I say, the consul's days and more than the state,
Which will be so strike a single and his son,
And with him to him and the manner of the state,
And so shall be so for a fair and all the people,
And will I speak a courtery.

KING RICHARD III:
The consul of the contracted with him.

DUKE VINCENTIO:
The worst senselves and the state of his son,
Which have stand for his honour'd hands and

In [24]:

```
# Low temperatures results in more predictable text.
# Higher temperatures results in more surprising text.

def generate_text(model, start_string, temp):
  # Number of characters to generate
  num_generate = 1000

  # converting start string to numbers
  input_eval = [char_to_idx[s] for s in start_string]
  print(input_eval)

  # This operation is useful if you want to add a batch dimension to a single element.
  # For example, if you have a single image of shape [height, width, channels],
  # you can make it a batch of one image with expand_dims(image, 0), which will make the shape [1, height, width, channels].
  input_eval = tf.expand_dims(input_eval, 0)
  print(input_eval)

  # empty string to store results
  text_generated = []

  # Experiment to find the best setting. (0-1)
  temperature = temp

  # here batch size = 1
  model.reset_states()
  for i in range(num_generate):
    predictions = model(input_eval)

    # remove batch dimentions
    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(idx_to_char[predicted_id])

  return (start_string + ''.join(text_generated))
print(generate_text(model, start_string=u"ROMEO: ", temp= 0.9))
```

[30, 27, 25, 17, 27, 10, 1]
tf.Tensor([[30 27 25 17 27 10 1]], shape=(1, 7), dtype=int32)
ROMEO: My willing speak?

DUCHESS OF YORK:
This was, ark not all and spirit,
Bointy to complaive to unfortune fears, and highness
Laster and many in God's upon me; or thy doth had hooned
Hand the chimumed stands that saud within thy horring dither,
Romeowh Farsio, my gentle Isage,
With assurance things and divenjed madver himsal is any perfoced asperies?

ISABELLA:
Ko was you maidful well.

KATHARINA:
My triban, thou must have weld dead?
What's ever loveds me to this spirit,
No beghant Bulian:
Which thou, damn'st none of what back.

JULIET:
Ay, jointed, my gifter Henry, the recharge man's noble father'd hours, and edemy too langurest young banished.
And, in more soldier may sleep.

ISABELLA:
Poor man?

DUCHESS OF YORK:
Romeo!
I am aboved, or your success! and must never beautions,
In Eagon!

WARWICK:
Grieve you, as my laughter.

DUKE VINCENTIO:
What a haven baid!

ANGELO:
His hope, I must not fall wis truels too
Sometimen's such a single or the end.

WARWICK:
And now now, beargon of him.

ANG

Operations are recorded if they are executed within this context manager and at least one of their inputs is being "watched".

Trainable variables (created by tf.Variable or tf.compat.v1.get_variable, where trainable=True is default in both cases) are automatically watched. Tensors can be manually watched by invoking the watch method on this context manager.

```
tf.GradientTape(persistent=False, watch_accessed_variables=True)
```

For example, consider the function y = x * x. The gradient at x = 3.0 can be computed as:

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
  g.watch(x)
  y = x * x
dy_dx = g.gradient(y, x) # Will compute to 6.0
```

GradientTapes can be nested to compute higher-order derivatives. For example,

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
  g.watch(x)
  with tf.GradientTape() as gg:
    gg.watch(x)
    y = x * x
  dy_dx = gg.gradient(y, x)     # Will compute to 6.0
d2y_dx2 = g.gradient(dy_dx, x)  # Will compute to 2.0
```

By default, the resources held by a GradientTape are released as soon as GradientTape.gradient() method is called. To compute multiple gradients over the same computation, create a persistent gradient tape. This allows multiple calls to the gradient() method as resources are released when the tape object is garbage collected. For example:

```
x = tf.constant(3.0)
with tf.GradientTape(persistent=True) as g:
  g.watch(x)
  y = x * x
  z = y * y
dz_dx = g.gradient(z, x)  # 108.0 (4*x^3 at x = 3)
dy_dx = g.gradient(y, x)  # 6.0
del g  # Drop the reference to the tape
```

By default GradientTape will automatically watch any trainable variables that are accessed inside the context. If you want fine grained control over which variables are watched you can disable automatic tracking by passing watch_accessed_variables=False to the tape constructor:

```
with tf.GradientTape(watch_accessed_variables=False) as tape:
  tape.watch(variable_a)
  y = variable_a ** 2  # Gradients will be available for`variable_a`.
  z = variable_b ** 3  # No gradients will be available since `variable_b` is not being watched.
```

Note that when using models you should ensure that your variables exist when using watch_accessed_variables=False. Otherwise it's quite easy to make your first iteration not have any gradients:

```
a = tf.keras.layers.Dense(32)
b = tf.keras.layers.Dense(32)

with tf.GradientTape(watch_accessed_variables=False) as tape:
  tape.watch(a.variables)  # Since `a.build` has not been called at this point `a.variables` will return an empty list and the tape will not be wat
ching anything.
  result = b(a(inputs))
  tape.gradient(result, a.variables)  # The result of this computation will be a list of `None`s since a's variables are not being watched.
```

Note that only tensors with real or complex dtypes are differentiable.

In [0]:

```
optimizer = tf.keras.optimizers.Adam()
```

Contrast:

```
def dense(x, W, b):
  return tf.nn.sigmoid(tf.matmul(x, W) + b)

@tf.function
def multilayer_perceptron(x, w0, b0, w1, b1, w2, b2 ...):
  x = dense(x, w0, b0)
  x = dense(x, w1, b1)
  x = dense(x, w2, b2)
  ...
  # You still have to manage w_i and b_i, and their shapes are
    defined far away from the code.
```

with the Keras version:

```
# Each layer can be called, with a signature equivalent to linear(x)
layers = [tf.keras.layers.Dense(hidden_size, activation=tf.nn.sigmoid) for _ in range(n)]
perceptron = tf.keras.Sequential(layers)

# layers[3].trainable_variables => returns [w3, b3]
# perceptron.trainable_variables => returns [w0, b0, ...]
```

In [0]:

```
@tf.function
def train_step(inp, target):
  with tf.GradientTape( ) as tape:
    predictions = model(inp)
    loss = tf.reduce_mean(input_tensor=
                tf.keras.losses.sparse_categorical_crossentropy(y_true= target,
                                                                 y_pred= predictions,
                                                                  from_logits= True))

  # Keras models and layers offer the convenient 'variables' and 'trainable_variables' properties,
  # which recursively gather up all dependent variables. This makes it easy to manage variables
  # locally to where they are being used. (refer above)
  grads = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  return loss
```

In [0]:

```
checkpoint_prefix = os.path.join('/content', "ckpt_{epoch}")

model = model_build( vocab_size = vocab_size, embedding_dim=embedding_dim, rnn_units=rnn_units, batch_size= batch_size)
```

In [28]:

```
for epoch in range(30):
  start= time.time()
```

```
  # initializing the hidden state at the start of every epoch
  # initally hidden is None
  hidden = model.reset_states()

  for(i, (ip, op)) in enumerate(dataset):
    loss = train_step(inp = ip, target = op)

    if i % 100 == 0:
      print('Epoch {} Batch {} Loss {}'.format(epoch+1, i, loss))

  # saving (checkpoint) the model every 5 epochs
  if (epoch + 1) % 5 == 0:
    model.save_weights(checkpoint_prefix.format(epoch=epoch))

  print ('Epoch {} Loss {:.4f}'.format(epoch+1, loss))
  print ('Time taken for 1 epoch {} sec\n'.format(time.time() - start))

model.save_weights(checkpoint_prefix.format(epoch=epoch))
```

```
Epoch 1 Batch 0 Loss 4.1754255294799805
Epoch 1 Batch 100 Loss 2.288322687149048
Epoch 1 Loss 2.0443
Time taken for 1 epoch 11.471296787261963 sec

Epoch 2 Batch 0 Loss 2.9659547805786133
Epoch 2 Batch 100 Loss 1.820554494857788
Epoch 2 Loss 1.7222
Time taken for 1 epoch 10.026393413543701 sec

Epoch 3 Batch 0 Loss 1.669248104095459
Epoch 3 Batch 100 Loss 1.625135898590088
Epoch 3 Loss 1.5390
Time taken for 1 epoch 10.032483577728271 sec

Epoch 4 Batch 0 Loss 1.5384776592254639
Epoch 4 Batch 100 Loss 1.4924185276031494
Epoch 4 Loss 1.4415
Time taken for 1 epoch 9.897800922393799 sec

Epoch 5 Batch 0 Loss 1.421339988708496
Epoch 5 Batch 100 Loss 1.4078408479690552
Epoch 5 Loss 1.3751
Time taken for 1 epoch 10.052716255187988 sec

Epoch 6 Batch 0 Loss 1.3726691007614136
Epoch 6 Batch 100 Loss 1.3876155614852905
Epoch 6 Loss 1.3272
Time taken for 1 epoch 10.050699472427368 sec

Epoch 7 Batch 0 Loss 1.2765899896621704
Epoch 7 Batch 100 Loss 1.338740587234497
Epoch 7 Loss 1.3208
Time taken for 1 epoch 10.013462781906128 sec

Epoch 8 Batch 0 Loss 1.2312674522399902
Epoch 8 Batch 100 Loss 1.3026504516601562
Epoch 8 Loss 1.2785
Time taken for 1 epoch 10.017131328582764 sec

Epoch 9 Batch 0 Loss 1.1907991170883179
Epoch 9 Batch 100 Loss 1.2574515342712402
Epoch 9 Loss 1.2876
Time taken for 1 epoch 10.022722721099854 sec

Epoch 10 Batch 0 Loss 1.1921272277832031
Epoch 10 Batch 100 Loss 1.239667296409607
Epoch 10 Loss 1.2474
Time taken for 1 epoch 10.094823837280273 sec

Epoch 11 Batch 0 Loss 1.1304038763046265
Epoch 11 Batch 100 Loss 1.1984503269195557
Epoch 11 Loss 1.1991
Time taken for 1 epoch 9.984236240386963 sec

Epoch 12 Batch 0 Loss 1.0817675590515137
Epoch 12 Batch 100 Loss 1.1611994504928589
Epoch 12 Loss 1.1701
Time taken for 1 epoch 10.086061000823975 sec

Epoch 13 Batch 0 Loss 1.0644326210021973
Epoch 13 Batch 100 Loss 1.1253539323806763
Epoch 13 Loss 1.1568
```

```
Epoch 13 Loss 1.1368
Time taken for 1 epoch 10.098074436187744 sec

Epoch 14 Batch 0 Loss 1.0325512886047363
Epoch 14 Batch 100 Loss 1.0482486486434937
Epoch 14 Loss 1.0914
Time taken for 1 epoch 10.088799476623535 sec

Epoch 15 Batch 0 Loss 0.9930705428123474
Epoch 15 Batch 100 Loss 1.0514111518859863
Epoch 15 Loss 1.0678
Time taken for 1 epoch 10.078505992889404 sec

Epoch 16 Batch 0 Loss 0.9693559408187866
Epoch 16 Batch 100 Loss 0.9628669619560242
Epoch 16 Loss 1.0311
Time taken for 1 epoch 9.989897727966309 sec

Epoch 17 Batch 0 Loss 0.897329568862915
Epoch 17 Batch 100 Loss 0.9938720464706421
Epoch 17 Loss 0.9999
Time taken for 1 epoch 9.990666389465332 sec

Epoch 18 Batch 0 Loss 0.8676051497459412
Epoch 18 Batch 100 Loss 0.9251348376274109
Epoch 18 Loss 0.9483
Time taken for 1 epoch 10.02344274520874 sec

Epoch 19 Batch 0 Loss 0.8076867461204529
Epoch 19 Batch 100 Loss 0.8923636078834534
Epoch 19 Loss 0.9300
Time taken for 1 epoch 10.014105558395386 sec

Epoch 20 Batch 0 Loss 0.7986055612564087
Epoch 20 Batch 100 Loss 0.866606593132019
Epoch 20 Loss 0.8599
Time taken for 1 epoch 9.996323823928833 sec

Epoch 21 Batch 0 Loss 0.736359179019928
Epoch 21 Batch 100 Loss 0.7974749803543091
Epoch 21 Loss 0.8622
Time taken for 1 epoch 9.956365585327148 sec

Epoch 22 Batch 0 Loss 0.7135876417160034
Epoch 22 Batch 100 Loss 0.7906237840652466
Epoch 22 Loss 0.8438
Time taken for 1 epoch 9.98558759689331 sec

Epoch 23 Batch 0 Loss 0.6805621385574341
Epoch 23 Batch 100 Loss 0.779574990272522
Epoch 23 Loss 0.7950
Time taken for 1 epoch 10.117630004882812 sec

Epoch 24 Batch 0 Loss 0.6632631421089172
Epoch 24 Batch 100 Loss 0.7230767607688904
Epoch 24 Loss 0.7571
Time taken for 1 epoch 10.00714898109436 sec

Epoch 25 Batch 0 Loss 0.5955040454864502
Epoch 25 Batch 100 Loss 0.71584153175354
Epoch 25 Loss 0.7389
Time taken for 1 epoch 10.07948637008667 sec

Epoch 26 Batch 0 Loss 0.5960854887962341
Epoch 26 Batch 100 Loss 0.6746395826339722
Epoch 26 Loss 0.6922
Time taken for 1 epoch 10.128275871276855 sec

Epoch 27 Batch 0 Loss 0.5712471008300781
Epoch 27 Batch 100 Loss 0.6652942895889282
Epoch 27 Loss 0.6718
Time taken for 1 epoch 10.076830625534058 sec

Epoch 28 Batch 0 Loss 0.5400032997131348
Epoch 28 Batch 100 Loss 0.6365725994110107
Epoch 28 Loss 0.6869
Time taken for 1 epoch 10.057198762893677 sec

Epoch 29 Batch 0 Loss 0.5070273876190186
Epoch 29 Batch 100 Loss 0.6035774946212769
Epoch 29 Loss 0.6360
Time taken for 1 epoch 10.005269527435303 sec
```

```
Epoch 30 Batch 0 Loss 0.465404748916626
Epoch 30 Batch 100 Loss 0.5846977829933167
Epoch 30 Loss 0.6359
Time taken for 1 epoch 10.06916069984436 sec
```

In [32]:

```
model = model_build(vocab_size, embedding_dim, rnn_units, batch_size=1)

model.load_weights('/content/ckpt_29')

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

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_6 (Embedding)      (1, None, 256)            16640
_________________________________________________________________
lstm_6 (LSTM)                (1, None, 1024)           5246976
_________________________________________________________________
dense_6 (Dense)              (1, None, 65)             66625
=================================================================
Total params: 5,330,241
Trainable params: 5,330,241
Non-trainable params: 0
_________________________________________________________________
```

In [33]:

```
# Low temperatures results in more predictable text.
# Higher temperatures results in more surprising text.

def generate_text(model, start_string, temp):
  # Number of characters to generate
  num_generate = 1000

  # converting start string to numbers
  input_eval = [char_to_idx[s] for s in start_string]
  print(input_eval)

  # This operation is useful if you want to add a batch dimension to a single element.
  # For example, if you have a single image of shape [height, width, channels],
  # you can make it a batch of one image with expand_dims(image, 0), which will make the shape [1, height, width, channels].
  input_eval = tf.expand_dims(input_eval, 0)
  print(input_eval)

  # empty string to store results
  text_generated = []

  # Experiment to find the best setting. (0-1)
  temperature = temp

  # here batch size = 1
  model.reset_states()
  for i in range(num_generate):
      predictions = model(input_eval)

      # remove batch dimentions
      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(idx_to_char[predicted_id])

  return (start_string + ''.join(text_generated))
print(generate_text(model, start_string=u"ROMEO: ", temp= 0.5))
```

```
[30, 27, 25, 17, 27, 10, 1]
tf.Tensor([[30 27 25 17 27 10  1]], shape=(1, 7), dtype=int32)
ROMEO: here is that heaven
from the foul musician, and the Lord Hastings,
Her four grace with his life to do him doing of the king.
```

DUKE OF AUMERLE:
Northumberland comes back to die:
I will take order for her beauty makes
As doth a sail, fill'd with a stamp, of which he hath wronged on
my heart.

ANTONIO:
Nay, good my lord,
I dare deliver me the matter: then,
if the in the other's tale against our soldiers?

WARWICK:
Henry made you now?
The way or wife?

ELBOW:
My Lord of Surrey, why for a name and the match'd that she say,
I was too hot a day as her power in his hands, the aughter of Lancaster.
You are a louthed splay'd for a new-made grave
And hope the love to her her; thou shalt not have accuseved
To some remote and covert the like a clout
Attended to by favour it hath set adied before the watch, and that it may put before the watch, and cruel with the fire
Of his bosom of the maid you are,
That it may bear me speak, I'll be your honour,
Doing to behind the heavens that hath done me so,
T