In [1]:
```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Note: This notebook is taken from this official tutorial and refactored to run on the Tensorflow version used in class.

Text generation with an RNN

This tutorial demonstrates how to generate text using a character-based RNN. You will work with a dataset of Shakespeare's writing from Andrej Karpathy's The Unreasonable Effectiveness of Recurrent Neural Networks. 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.

Note: Enable GPU acceleration to execute this notebook faster. In Colab: *Runtime* > *Change runtime type* > *Hardware accelerator* > *GPU*.

This tutorial includes runnable code implemented using tf.keras and eager execution. The following is the sample output when the model in this tutorial trained for 30 epochs, and started with the prompt "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 no sin it should
be dead,
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.

Setup

Import TensorFlow and other libraries

In [2]:
```
import tensorflow as tf

import numpy as np
import os
import time
```

Read the data

First, look in the text:

In [3]:
```
path_to_file = f'./data/shakespeare.txt'
```

In [4]:
```
# Read, then decode for py2 compat.
text = open(path_to_file, 'rb').read().decode(encoding='utf-8')
# Length of text is the number of characters in it
print(f'Length of text: {len(text)} characters')
```

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

In [5]:
```
# Take a look at the first 250 characters in text
print(text[:250])
```

```
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 [6]:
```
# The unique characters in the file
vocab = sorted(set(text))
print(f'{len(vocab)} unique characters')
```

```
65 unique characters
```

Process the text

Vectorize the text

Before training, you need to convert the strings to a numerical representation.

The `tf.keras.layers.StringLookup` layer can convert each character into a numeric ID. It just needs the text to be split into tokens first.

In [7]:
```
example_texts = ['abcdefg', 'xyz']

chars = tf.strings.unicode_split(example_texts, input_encoding='UTF-8')
chars
```

Out[7]:
```
<tf.RaggedTensor [[b'a', b'b', b'c', b'd', b'e', b'f', b'g'], [b'x', b'y', b'z']]>
```

Now create the `tf.keras.layers.StringLookup` layer:

In [8]:
```
ids_from_chars = tf.keras.layers.StringLookup(
    vocabulary=list(vocab), mask_token=None)
```

It converts from tokens to character IDs:

In [9]:
```
ids = ids_from_chars(chars)
ids
```

Out[9]:
```
<tf.RaggedTensor [[40, 41, 42, 43, 44, 45, 46], [63, 64, 65]]>
```

Since the goal of this tutorial is to generate text, it will also be important to invert this representation and recover human-readable strings from it. For this you can use `tf.keras.layers.StringLookup(..., invert=True)` .

Note: Here instead of passing the original vocabulary generated with `sorted(set(text))` use the `get_vocabulary()` method of the `tf.keras.layers.StringLookup` layer so that the `[UNK]` tokens is set the same way.

In [10]:
```
chars_from_ids = tf.keras.layers.StringLookup(
    vocabulary=ids_from_chars.get_vocabulary(), invert=True, mask_token=None)
```

This layer recovers the characters from the vectors of IDs, and returns them as a `tf.RaggedTensor` of characters:

In [11]:
```
chars = chars_from_ids(ids)
chars
```

Out[11]:
```
<tf.RaggedTensor [[b'a', b'b', b'c', b'd', b'e', b'f', b'g'], [b'x', b'y', b'z']]>
```

You can `tf.strings.reduce_join` to join the characters back into strings.

In [12]:
```
tf.strings.reduce_join(chars, axis=-1).numpy()
```

Out[12]:
```
array([b'abcdefg', b'xyz'], dtype=object)
```

In [13]:
```
def text_from_ids(ids):
  return tf.strings.reduce_join(chars_from_ids(ids), axis=-1)
```

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.

In [14]:
```
all_ids = ids_from_chars(tf.strings.unicode_split(text, 'UTF-8'))
all_ids
```

Out[14]:
```
<tf.Tensor: shape=(1115394,), dtype=int64, numpy=array([19, 48, 57, ..., 46,  9,  1])>
```

In [15]:
```
ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
```

In [16]:
```
for ids in ids_dataset.take(10):
    print(chars_from_ids(ids).numpy().decode('utf-8'))
```

```
F
i
r
s
t

C
i
t
i
```

In [17]:
```
seq_length = 100
```

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

In [18]:
```
sequences = ids_dataset.batch(seq_length+1, drop_remainder=True)

for seq in sequences.take(1):
  print(chars_from_ids(seq))
```

```
tf.Tensor(
[b'F' b'i' b'r' b's' b't' b' ' b'C' b'i' b't' b'i' b'z' b'e' b'n' b':'
 b'\n' b'B' b'e' b'f' b'o' b'r' b'e' b' ' b'w' b'e' b' ' b'p' b'r' b'o'
 b'c' b'e' b'e' b'd' b' ' b'a' b'n' b'y' b' ' b'f' b'u' b'r' b't' b'h'
 b'e' b'r' b',' b' ' b'h' b'e' b'a' b'r' b' ' b'm' b'e' b' ' b's' b'p'
 b'e' b'a' b'k' b'.' b'\n' b'\n' b'A' b'l' b'l' b':' b'\n' b'S' b'p' b'e'
 b'a' b'k' b',' b' ' b's' b'p' b'e' b'a' b'k' b'.' b'\n' b'\n' b'F' b'i'
 b'r' b's' b't' b' ' b'C' b'i' b't' b'i' b'z' b'e' b'n' b':' b'\n' b'Y'
 b'o' b'u' b' '], shape=(101,), dtype=string)
```

It's easier to see what this is doing if you join the tokens back into strings:

In [19]:
```
for seq in sequences.take(5):
  print(text_from_ids(seq).numpy())
```

```
b'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '
b'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\n\nFirst Citizen:\nFirst, you k'
b"now Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we know't.\n\nFirst Citizen:\nLet us ki"
b"ll him, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo more talking on't; let it be d"
b'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citizen:\nWe are accounted poor citi'
```

For training you'll need a dataset of `(input, label)` pairs. Where `input` and `label` are sequences. At each time step the input is the current character and the label is the next character.

Here's a function that takes a sequence as input, duplicates, and shifts it to align the input and label for each timestep:

In [20]:
```
def split_input_target(sequence):
    input_text = sequence[:-1]
    target_text = sequence[1:]
    return input_text, target_text
```

In [21]:
```
split_input_target(list("Tensorflow"))
```

Out[21]:
```
(['T', 'e', 'n', 's', 'o', 'r', 'f', 'l', 'o'],
 ['e', 'n', 's', 'o', 'r', 'f', 'l', 'o', 'w'])
```

In [22]:
```
dataset_split = sequences.map(split_input_target)
```

In [23]:
```
for input_example, target_example in dataset_split.take(1):
    print("Input :", text_from_ids(input_example).numpy())
    print("Target:", text_from_ids(target_example).numpy())
```

```
Input : b'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou'
Target: b'irst Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '
```

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.

In [24]:
```
# 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
```

```
dataset = (
    dataset_split
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE, drop_remainder=True)
    .cache()
    .prefetch(tf.data.AUTOTUNE))

dataset
```

Out[24]:
```
<_PrefetchDataset element_spec=(TensorSpec(shape=(64, 100), dtype=tf.int64, name=None), TensorSpec(shape=(64, 100), dtype=tf.
int64, name=None))>
```

Build The Model

This section defines the model as a `keras.Model` subclass (For details see Making new Layers and Models via subclassing).

This model has three layers:

- `tf.keras.layers.Embedding` : The input layer. A trainable lookup table that will map each character-ID 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. It outputs one logit for each character in the vocabulary. These are the log-likelihood of each character according to the model.

In [25]:
```
# Length of the vocabulary in StringLookup Layer
vocab_size = len(ids_from_chars.get_vocabulary())

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

In [26]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.GRU(rnn_units, return_sequences=True),
    tf.keras.layers.Dense(vocab_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:

Note: For training you could use a `keras.Sequential` model here. To generate text later you'll need to manage the RNN's internal state. It's simpler to include the state input and output options upfront, than it is to rearrange the model architecture later. For more details see the Keras RNN guide.

Try the model

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

First check the shape of the output:

In [27]:
```
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, 66) # (batch_size, sequence_length, vocab_size)
```

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

In [28]:
```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(64, 100, 256)	16,896
gru (GRU)	(64, 100, 1024)	3,938,304
dense (Dense)	(64, 100, 66)	67,650

Total params: 4,022,850 (15.35 MB)
Trainable params: 4,022,850 (15.35 MB)
Non-trainable params: 0 (0.00 B)

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:

In [29]:
```
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:

In [30]:
```
sampled_indices
```

Out[30]:
```
array([32, 49, 57, 62, 39, 31, 56,  7, 22,  6,  5, 19,  8, 41, 36, 33, 45,
       26, 10, 14, 47, 28,  0, 35, 15, 34, 27, 34,  1, 40, 46, 42, 64, 13,
        2, 46,  4, 16, 34, 57, 58,  3, 20, 37, 29, 40, 15,  5, 44,  8, 65,
       63,  2, 43, 42, 31, 30, 38, 38, 51, 22, 29, 47, 10, 19, 27,  2, 63,
       33, 39, 53, 11, 23, 44, 42, 34, 30, 28, 10, 51, 53, 31, 15, 54, 63,
       33, 45, 56, 46, 54, 43, 57,  9, 33, 61, 53, 42, 57, 44, 26])
```

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

In [31]:
```
print("Input:\n", text_from_ids(input_example_batch[0]).numpy())
print()
print("Next Char Predictions:\n", text_from_ids(sampled_indices).numpy())
```

```
Input:
 b'\nISABELLA:\nShow me how, good father.\n\nDUKE VINCENTIO:\nThis forenamed maid hath yet in her the contin'

Next Char Predictions:
 b"SjrwZRq,I'&F-bWTfM3AhO[UNK]VBUNU\nagcy? g$CUrs!GXPaB&e-zx dcRQYYlIPh3FN xTZn:JecUQO3lnRBoxTfqgodr.TvncreM"
```

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.

In [32]:
```
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
```

In [33]:
```
example_batch_mean_loss = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_size, sequence_length, vocab_size)")
print("Mean loss:        ", example_batch_mean_loss)
```

```
Prediction shape:  (64, 100, 66)  # (batch_size, sequence_length, vocab_size)
Mean loss:         tf.Tensor(4.1902504, shape=(), dtype=float32)
```

A newly initialized model shouldn't be too sure of itself, the output logits should all have similar magnitudes. To confirm this you can check that the exponential of the mean loss is approximately equal to the vocabulary size. A much higher loss means the model is sure of its wrong answers, and is badly initialized:

In [34]:
```
tf.exp(example_batch_mean_loss).numpy()
```

Out[34]: 66.03932

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

In [35]:
```
model.compile(optimizer='adam', loss=loss, metrics=['sparse_categorical_accuracy'])
```

Execute the training

To keep training time reasonable, use 20 epochs to train the model. In Colab, set the runtime to GPU for faster training.

In [36]:
```
EPOCHS = 20
```

In [37]:
```
history = model.fit(dataset, epochs=EPOCHS)
```

```
Epoch 1/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - loss: 3.0742 - sparse_categorical_accuracy: 0.2367
Epoch 2/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.9251 - sparse_categorical_accuracy: 0.4339
Epoch 3/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.6381 - sparse_categorical_accuracy: 0.5121
Epoch 4/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.4914 - sparse_categorical_accuracy: 0.5501
Epoch 5/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.4039 - sparse_categorical_accuracy: 0.5722
Epoch 6/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.3424 - sparse_categorical_accuracy: 0.5870
Epoch 7/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.2938 - sparse_categorical_accuracy: 0.5994
Epoch 8/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.2510 - sparse_categorical_accuracy: 0.6105
Epoch 9/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.2103 - sparse_categorical_accuracy: 0.6215
Epoch 10/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.1716 - sparse_categorical_accuracy: 0.6324
Epoch 11/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.1324 - sparse_categorical_accuracy: 0.6438
Epoch 12/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.0913 - sparse_categorical_accuracy: 0.6565
Epoch 13/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.0525 - sparse_categorical_accuracy: 0.6683
Epoch 14/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 1.0157 - sparse_categorical_accuracy: 0.6793
Epoch 15/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 18ms/step - loss: 0.9840 - sparse_categorical_accuracy: 0.6887
Epoch 16/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 0.9548 - sparse_categorical_accuracy: 0.6970
Epoch 17/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 0.9292 - sparse_categorical_accuracy: 0.7048
Epoch 18/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 0.9066 - sparse_categorical_accuracy: 0.7108
Epoch 19/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 0.8871 - sparse_categorical_accuracy: 0.7164
Epoch 20/20
172/172 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - loss: 0.8694 - sparse_categorical_accuracy: 0.7216
```

Generate text

The simplest way to generate text with this model is to run it in a loop, and keep track of the model's internal state as you execute it.

Each time you call the model you pass in some text and an internal state. The model returns a prediction for the next character and its new state. Pass the prediction and state back in to continue generating text.

The following makes a single step prediction:

In [38]:
```
class OneStep(tf.keras.Model):
  def __init__(self, model, chars_from_ids, ids_from_chars, temperature=1.0):
    super().__init__()
    self.temperature = temperature
    self.model = model
    self.chars_from_ids = chars_from_ids
    self.ids_from_chars = ids_from_chars
```

```
    # Create a mask to prevent "[UNK]" from being generated.
    skip_ids = self.ids_from_chars(['[UNK]'])[:, None]
    sparse_mask = tf.SparseTensor(
        # Put a -inf at each bad index.
        values=[-float('inf')]*len(skip_ids),
        indices=skip_ids,
        # Match the shape to the vocabulary
        dense_shape=[len(ids_from_chars.get_vocabulary())])
    self.prediction_mask = tf.sparse.to_dense(sparse_mask)

  @tf.function
  def generate_one_step(self, inputs, states=None):
    # Convert strings to token IDs.
    input_chars = tf.strings.unicode_split(inputs, 'UTF-8')
    input_ids = self.ids_from_chars(input_chars).to_tensor()

    # Embedding layer
    x = self.model.layers[0](input_ids)
    # GRU layer
    x = self.model.layers[1](x, initial_state=states)
    # Get the hidden state of the last timestep
    states = x[:, -1, :]
    # Dense layer
    predicted_logits = self.model.layers[2](x)

    # Only use the last prediction.
    predicted_logits = predicted_logits[:, -1, :]
    predicted_logits = predicted_logits/self.temperature

    # Apply the prediction mask: prevent "[UNK]" from being generated.
    predicted_logits = predicted_logits + self.prediction_mask

    # # Sample the output logits to generate token IDs.
    predicted_ids = tf.random.categorical(predicted_logits, num_samples=1)
    predicted_ids = tf.squeeze(predicted_ids, axis=-1)

    # # Convert from token ids to characters
    predicted_chars = self.chars_from_ids(predicted_ids)

    # Return the characters and model state.
    return predicted_chars, states
```

In [39]:
```
one_step_model = OneStep(model, chars_from_ids, ids_from_chars)
```

Run it in a loop to generate some text. 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.

In [40]:
```
start = time.time()
states = None
next_char = tf.constant(['ROMEO:'])
result = [next_char]

for n in range(1000):
  next_char, states = one_step_model.generate_one_step(next_char, states=states)
  result.append(next_char)

result = tf.strings.join(result)
end = time.time()
print(result[0].numpy().decode('utf-8'), '\n\n' + '_'*80)
print('\nRun time:', end - start)
```

```
ROMEO:
O my myself are we have good men from the loss of
thee; perform'd with the day, sir,' O wond!
Go as the compose, my speech, he shall be here;
And yet, it, worthy sir; not a devil him; I come
All but joinned them: or that and all tyails;
It shall be so; he says he soother no for shame.

RUTLENCE:
Ih thy hearing sun, my Know thou shadows,
I'll tell thee how: I then be set:
Iterpt these my friends stood God hold thy light.
Oneights can give mine eyes to the ground!
One whom you bid me king, mine honour needs in request?

GLOUCESTER:
Who ever I call up and then anon.

ANGELO:
Therefore tell thee, my lord, whom you shall not go to this?

EXETER:
The more more hull things on the needs be present,
Even true approne to the crown it, huntages pass,
And in his body to the Volsces early upon her,
But or she Engling services, gamed,
And crafles the jost courles in thine eyes
Were sometime homely, ro, for his indeed.
The sun the mortimenty are in his
head; rare him by jest or too;
But letter, unly

_____________________________________________________________________

Run time: 2.195650100708008
```

The easiest thing you can do to improve the results is to train it for longer (try `EPOCHS = 30`).

You can also experiment with a different start string, try adding another RNN layer to improve the model's accuracy, or adjust the temperature parameter to generate more or less random predictions.

If you want the model to generate text *faster* the easiest thing you can do is batch the text generation. In the example below the model generates 5 outputs in about the same time it took to generate 1 above.

In [41]:
```python
start = time.time()
states = None
next_char = tf.constant(['ROMEO:', 'ROMEO:', 'ROMEO:', 'ROMEO:', 'ROMEO:'])
result = [next_char]

for n in range(1000):
  next_char, states = one_step_model.generate_one_step(next_char, states=states)
  result.append(next_char)

results = tf.strings.join(result)
end = time.time()

for text in results:
    print(text.numpy().decode('utf-8'), '\n\n' + '_'*80)

print('\nRun time:', end - start)
```

ROMEO:
Then is the dead, that did not this.
What, must I should leave this cure of this?

FLIAR LAURENCE:

GLOUCESTER:
How oft her queen is true, the mew to of grief?

PETRUCHIO:
Where is the barry of the mother and unbrues live
Doth live many yielding to the wall:
Bound am veins of tune, yet dignity'st unlovely
The hortess of the midnity.

CLARENCE:
Then all myself against him. Ifamilish claim
Attoning men in pristing.-
CARDINAND:
And, sir, your son, where, I should last my liege,
Hath been a child, his heapts in any her is,
Be paking's floge;' and that it were given too:
Ba king so apen done, methinks he was for the histom.

KING HENRY VI:
And good so, good, then, our streep's time; dishonour your honours,
And you, my Lord Hastings, Romeo slept,
Whom I weep the dog, I would be wrong: therefore
'tis car making and old tear her robt here.

GREMIO:
And he's onet of the benefice of the king.

Nurse:
Now, for my grave is and fairly senten?
Peace know'st thou ne'ers? Is there no more, may I that

ROMEO:
When you have made free: London, now thyself more: what hap
the neck maintain? Dare no further than honest Benvalous.

DUKE VINCENTIO:
O Romeo's wife go till then husband; though we
Then am Duke with him: yet say he
called my life may serve of his foul;
For Gremio, I hear my cousing stars!

KING RICHARD III:
Back, for the good gose-bart this punish quart Angel:
Then, since thyself upon it, strey, thy heart,
Divines litter sustings and ill window.

DUKE OF AUMERLET:
Be his firm. Then music of the tradesm;
But I, we shall be streat, nor I denier my news,
That, for my state, when monisher, hath coundression in hope,
Then were he no son to know his youth:
'Tis mine honesty since when thou wastives on the furthes dought
But to inshinted for a bury:
So yo walk, she's heat, and, whiles and moved mind to his glorious small
bearing brook, and bound at our own eyes too,
He are the death of Lancaster; let him be prepared.

DUKE VINCENTIO:
Your justice, know not, how can I go?

ADRIAN:
Twell me,

ROMEO:
Then again, I came
To mean me, lest, it is, when thou diest.

LEONTES:
One word.

LEONTES:
Why, what say! for these are you.

MENENIUS:
Or, if yield me not.

BENVOLIO:
Then what he hath are wonder? see, so stand up the air!
I am the needy that thou cowald, I'll rain:
Do you not here? father! give me thy head and men:
His ancient steps may moves him
To the forfer Bianca,
And you, my noble court-Well,
How if thou dead?

PAULINA:
He would not heave you what, you have wrong: but hear me.
Which are you or your grace? you mean--
So that alwory strengthens lent them; who now therefore diding Juliet?

PARIS:
My heart, quest, some noble isle, Lucentio.

CLIFFORD:
Believe me, sir; he hath not done, bit me for these fair Montague!
Is this for my prophet power,
Life and your suitor, even it with
Monach wretcy whife his purtinal counsel?
And yet master Doken here it to? 'Tis I protest
And blood the bloody deal himself to his country,
Her one half crack'd in this infamorous,
And was not in the axti

ROMEO:
These you unmost likes a present spetect and finest,
Goth kept thy underto use the people.

All:
Mark him hence.

PRINCE:
Come, come forth my heart, Grumio! for thy less
Than first. The duke has left conduct him with me.
And then, thy sheathness dangers, say. 'Tis very
greeness. Spirst fame, no more: then, ere I
See this drift.
Come, master, mark his malice till do repent;
Or, if you present her from her to him.

KING HENRY VI:
MARCINIUS:
Your awmit, sir, and your brother, I
And desire there dead the cale.

QUEEN MARGARET:
At the Doth not be devils, while his time to say!
Your tale, the law bear's grief.

ANTONIO:
When will he comes?

First Marshio, Hard! where be my brother, worthy sir,
Be here thou well think out of it,
Disinger me thy mother, though she speaks
By suffer'd labour Own it ear:
Marry, and you deply; and the one him of my head;
Or how are past.

MENENIUS:
This is no stoop and allWAMwitted blood or wretched went,
God keepings, and many a but fair.
Come, you are well for

ROMEO:
Then he that is not in thy heart be suffer'd too:
Unless he do exposingly unto him.

Whiles I have ta'en mere mine honour,
Monourable York and Lord Angelo,
And never till we hear 'to this enemy,

```
My arms presently he dill men
That modest with youth with her brother's incleant
Of blests, to my vow's will mar, presented monster,
To sped his trinking men to tell him not;
And all I say and, sir, we will respect
By any of the nurses of the bend of reason,
Or in proof about me, empty and
With Juliet that given me! only soon,
To reap the prince your household traitor
And be protector of the man: he did your lands
Lost wretched by my father's death?
How can we hear ' hear?

HENRY BOLINGBROKE:
My gracious lord, I hear the instrume spurs;
And III:
Heirs milleth nature, be not send Lord Green's mistress.

KING RICHARD IIII:
Ha!

AUFIDIUS:
Never, madam, sir; if you well devenge?

Nurse:
Op thy heavens like a little when you drown'd:
For my hearth! Give me my mistress.

HENRY PERCUT:
Now, blessed P

_____________________________________________________________________

Run time: 1.4787909984588623
```

Export the generator

This single-step model can easily be saved and restored, allowing you to use it anywhere a `tf.saved_model` is accepted.

In [42]:
```
tf.saved_model.save(one_step_model, 'one_step')
one_step_reloaded = tf.saved_model.load('one_step')
```

```
INFO:tensorflow:Assets written to: one_step/assets
INFO:tensorflow:Assets written to: one_step/assets
```

In [43]:
```
states = None
next_char = tf.constant(['ROMEO:'])
result = [next_char]

for n in range(100):
  next_char, states = one_step_reloaded.generate_one_step(next_char, states=states)
  result.append(next_char)

print(tf.strings.join(result)[0].numpy().decode("utf-8"))
```

```
ROMEO:
The nobles of his pleasure.
Anon, good man, the floody more of thy stinging.

MENENIUS:
For his fri
```