In [1]:

```
import tensorflow as tf
from tensorflow.keras.layers.experimental import preprocessing

import numpy as np
import os
import time
```

In [2]:

```
path_to_file = tf.keras.utils.get_file('shakespeare.txt', 'https://storage.googl
```

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

In [3]:

```
path_to_file
```

Out[3]:

```
'/root/.keras/datasets/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 [6]:

```
# print(text)
```

In [7]:

```
# 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 [8]:

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

```
65 unique characters
```

In [10]:

```
print(vocab)
```

```
['\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']
```

Process the text

In [16]:

```
#The preprocessing.StringLookup layer can convert each character into
#a numeric ID. It just needs the text to be split into tokens first.
example_texts = ['abcdefg', 'xyz']

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

Out[16]:

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

In [17]:

```
#Now create the preprocessing.StringLookup layer:
ids_from_chars = preprocessing.StringLookup(
    vocabulary=list(vocab))
```

In [18]:

```
#It converts form tokens to character IDs, padding with 0
ids = ids_from_chars(chars)
ids
```

Out[18]:

```
<tf.RaggedTensor [[41, 42, 43, 44, 45, 46, 47], [64, 65, 66]]>
```

In [19]:

```
#invert this representation and recover human-readable strings
chars_from_ids = tf.keras.layers.experimental.preprocessing.StringLookup(
    vocabulary=ids_from_chars.get_vocabulary(), invert=True)
```

In [20]:

```
#This layer recovers the characters from the vectors of IDs, and returns them as
chars = chars_from_ids(ids)
chars
```

Out[20]:

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

In [21]:

```
#You can tf.strings.reduce_join to join the characters back into strings.
def text_from_ids(ids):
  return tf.strings.reduce_join(chars_from_ids(ids), axis=-1)
```

In [22]:

```
text_from_ids(ids)
```

Out[22]:

```
<tf.Tensor: shape=(2,), dtype=string, numpy=array([b'abcdefg', b'xyz'], dtype=ob
ject)>
```

The prediction task

In [24]:
```
#Create training examples and targets
all_ids = ids_from_chars(tf.strings.unicode_split(text, 'UTF-8'))
all_ids
```

Out[24]:
```
<tf.Tensor: shape=(1115394,), dtype=int64, numpy=array([20, 49, 58, ..., 47, 10,
2])>
```

In [14]:
```
#use the tf.data.Dataset.from_tensor_slices function to convert the text vector
#a stream of character indices.
ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
```

In [23]:
```
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 [25]:
```
seq_length = 100
examples_per_epoch = len(text)//(seq_length+1)
```

In [26]:
```
#batch method lets you easily convert these individual characters
#to sequences of the desired size.
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)
```

In [28]:
```
#It's easier to see what this is doing if you join the tokens back into strings:
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 kno
w'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 mor
e talking on't; let it be d"
```

```
b'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citize
n:\nWe are accounted poor citi'
```

In [29]:

```python
#function that takes a sequence as input, duplicates, and shifts it to align the
#label for each timestep
def split_input_target(sequence):
    input_text = sequence[:-1]
    target_text = sequence[1:]
    return input_text, target_text
```

In [30]:

```python
dataset = sequences.map(split_input_target)
```

In [31]:

```python
for input_example, target_example in dataset.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\nAl
l:\nSpeak, speak.\n\nFirst Citizen:\nYou'
Target: b'irst Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\n
Speak, speak.\n\nFirst Citizen:\nYou '
```

Create training batches

In [32]:

```python
# 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
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE, drop_remainder=True)
    .prefetch(tf.data.experimental.AUTOTUNE))

dataset
```

Out[32]:

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

Build The Model

In [33]:

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

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

In [34]:

```
class MyModel(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, rnn_units):
    super().__init__(self)
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(rnn_units,
                                   return_sequences=True,
                                   return_state=True)
    self.dense = tf.keras.layers.Dense(vocab_size)

  def call(self, inputs, states=None, return_state=False, training=False):
    x = inputs
    x = self.embedding(x, training=training)
    if states is None:
      states = self.gru.get_initial_state(x)
    x, states = self.gru(x, initial_state=states, training=training)
    x = self.dense(x, training=training)

    if return_state:
      return x, states
    else:
      return x
```

In [35]:

```
model = MyModel(
    # Be sure the vocabulary size matches the `StringLookup` layers.
    vocab_size=len(ids_from_chars.get_vocabulary()),
    embedding_dim=embedding_dim,
    rnn_units=rnn_units)
```

In [36]:

```
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, voca
```

```
(64, 100, 67) # (batch_size, sequence_length, vocab_size)
```

In [37]:

```
model.summary()
```

```
Model: "my_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding (Embedding)        multiple                  17152     
_________________________________________________________________
gru (GRU)                    multiple                  3938304   
_________________________________________________________________
dense (Dense)                multiple                  68675     
=================================================================
Total params: 4,024,131
Trainable params: 4,024,131
Non-trainable params: 0
_________________________________________________________________
```

In [38]:

```
sampled_indices = tf.random.categorical(example_batch_predictions[0], num_sample
sampled_indices = tf.squeeze(sampled_indices, axis=-1).numpy()
```

In [39]:

```
sampled_indices
```

Out[39]:
```
array([27, 15,  9, 12, 24, 56, 55, 51,  5, 32, 51, 21, 33, 23, 28, 25, 64,
       16, 36, 10, 25, 41, 62,  3, 50, 24, 56, 53, 51,  3, 32, 26, 22, 55,
       14, 25, 40, 33, 21,  7, 19, 48, 39, 37,  3, 53, 52,  7, 61, 60, 16,
       50,  8, 36, 46, 55, 39, 60, 66,  2,  3, 23, 32, 64,  1, 36, 50, 11,
       26, 46,  4,  1, 15,  4, 10, 33,  6, 30, 63,  0,  0, 35, 51, 15, 63,
       33,  6, 59, 58, 48, 49, 51, 57, 57, 24, 66, 26,  1, 14, 41])
```

In [40]:
```
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"oubly portcullis'd with my teeth and lips;\nAnd dull unfeeling barren ignoran
ce\nIs made my gaoler to "

Next Char Predictions:
 b"MA-:Jpok$RkGSINKxBV.Kav jJpmk RLHo?KZSG'EhYW ml'utBj,VfoYtz\n IRx[UNK]Vj3Lf!
[UNK]A!.S&PwUkAwS&srhikqqJzL[UNK]?a"
```

Train the model

In [41]:
```
#Attach an optimizer, and a loss function
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
```

In [42]:
```
example_batch_loss = loss(target_example_batch, example_batch_predictions)
mean_loss = example_batch_loss.numpy().mean()
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_size, se
print("Mean loss:        ", mean_loss)
```

```
Prediction shape:  (64, 100, 67)  # (batch_size, sequence_length, vocab_size)
Mean loss:         4.2042265
```

In [43]:
```
tf.exp(mean_loss).numpy()
```

Out[43]:
```
66.96878
```

In [47]:
```
#Configure the training procedure using the tf.keras.Model.compile method. Use 
#arguments and the loss function
model.compile(optimizer='adam', loss=loss)
```

Configure checkpoints

In [48]:
```
#Use a tf.keras.callbacks.ModelCheckpoint to ensure that checkpoints are saved 
# Directory where the checkpoints will be saved
checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

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

Execute the training

In [51]:

```
EPOCHS = 40
```

In [52]:

```
history = model.fit(dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])
```

```
Epoch 1/40
172/172 [==============================] - 7s 36ms/step - loss: 0.6874
Epoch 2/40
172/172 [==============================] - 7s 36ms/step - loss: 0.6472
Epoch 3/40
172/172 [==============================] - 7s 36ms/step - loss: 0.6129
Epoch 4/40
172/172 [==============================] - 7s 36ms/step - loss: 0.5796
Epoch 5/40
172/172 [==============================] - 7s 37ms/step - loss: 0.5541
Epoch 6/40
172/172 [==============================] - 7s 37ms/step - loss: 0.5310
Epoch 7/40
172/172 [==============================] - 7s 36ms/step - loss: 0.5125
Epoch 8/40
172/172 [==============================] - 7s 36ms/step - loss: 0.4990
Epoch 9/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4866
Epoch 10/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4734
Epoch 11/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4648
Epoch 12/40
172/172 [==============================] - 7s 36ms/step - loss: 0.4558
Epoch 13/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4470
Epoch 14/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4458
Epoch 15/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4451
Epoch 16/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4357
Epoch 17/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4322
Epoch 18/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4300
Epoch 19/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4279
Epoch 20/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4258
Epoch 21/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4272
Epoch 22/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4222
Epoch 23/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4209
Epoch 24/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4239
Epoch 25/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4258
Epoch 26/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4284
Epoch 27/40
```

```
172/172 [==============================] - 7s 37ms/step - loss: 0.4214
Epoch 28/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4243
Epoch 29/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4234
Epoch 30/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4274
Epoch 31/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4323
Epoch 32/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4348
Epoch 33/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4364
Epoch 34/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4340
Epoch 35/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4322
Epoch 36/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4323
Epoch 37/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4354
Epoch 38/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4336
Epoch 39/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4408
Epoch 40/40
172/172 [==============================] - 7s 37ms/step - loss: 0.4487
```

Generate text

In [53]:

```python
#The following makes a single step prediction:
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 "" or "[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()

    # Run the model.
    # predicted_logits.shape is [batch, char, next_char_logits]
    predicted_logits, states = self.model(inputs=input_ids, states=states,
                                          return_state=True)
    # Only use the last prediction.
    predicted_logits = predicted_logits[:, -1, :]
    predicted_logits = predicted_logits/self.temperature
    # Apply the prediction mask: prevent "" or "[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 [54]:

```
one_step_model = OneStep(model, chars_from_ids, ids_from_chars)
```

In [55]:

```
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:
The last of Warwick call.

CAMILLO:
Nay, your office is dead.

LORTS:
So doth the death, they have a hearten mouth:
And how she'll sooner in his mother's.
The foul sir's news.

BIANCA:
Adgend and grief say he is colder,
For such a hand of her, of my heart
And hell in sucken me Above a cleagures
That you shall unto Lonnon o'er the land, whose hap that Clifford,
Was sentence of my life; for he dishollow'd attenting music and
personal, sir.

GLOUCESTER:
Sir Richard, Or little tongue
For beying foot to such absolate.

LEONTES:
What willoub children how?

ROMEO:
I pray thee, my lord.
```

```
DUKE VINCENTIO:
It is not meet him well: you are dishonour'd between out
Where'er the people of this iclan careless.
Say that she lives.

TRASALLE:
I do become him; for I think, let me hear
The wind sid little eye o' the people, who haviness
my wag's faithful fearful long, and nothing else.'
Hast thou behold it straight degree?

HORTENSIO:
For this affection given him hence,
And he will muck in promise beaute

________________________________________________________________________________

Run time: 1.754654884338379
```

In [ ]: