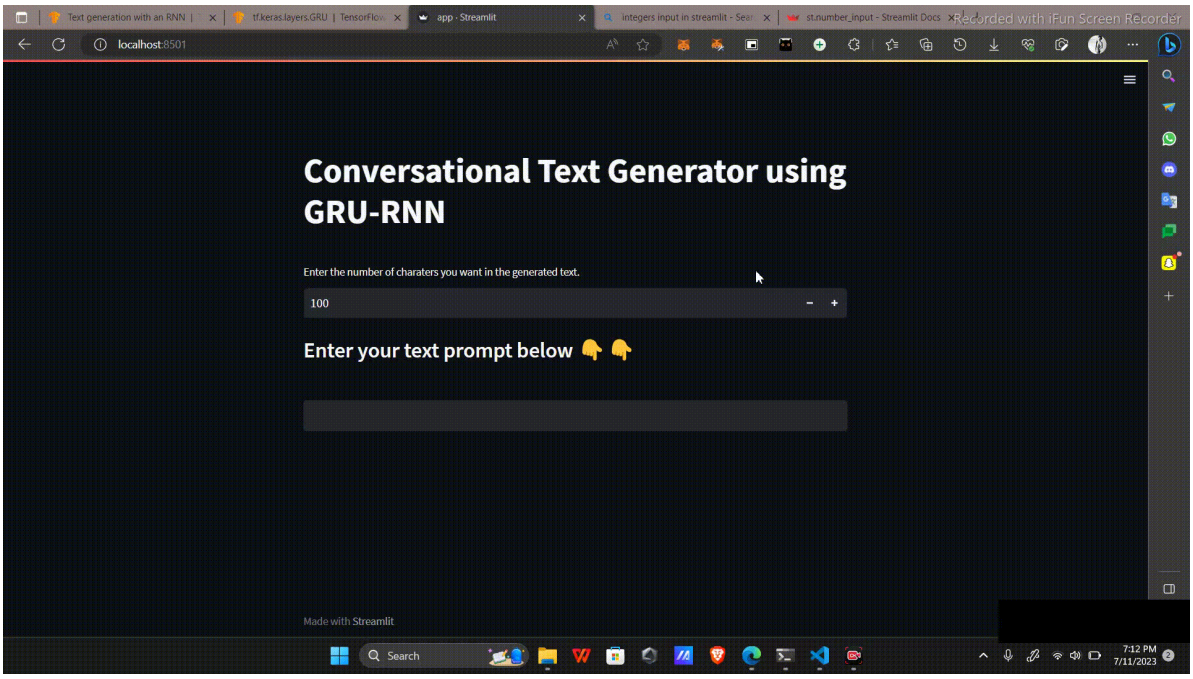
Conversational Text Generation Using GRU-RNN

Submitted By: `- Deepankar Sharma`



In [1]:
```
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.
```

In [3]:
```
# 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 [4]:
```
with open('data.txt', 'w')as f:
    f.write(text)
```

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

```
65 unique characters
```

In [7]:
```python
vocab
```

Out[7]:
```
['\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 [8]:
```
'''vectorization'''
ids_from_chars = tf.keras.layers.StringLookup(
    vocabulary=list(vocab), mask_token=None) # function to get numeric ID for each cha

chars_from_ids = tf.keras.layers.StringLookup(
    vocabulary=ids_from_chars.get_vocabulary(), invert=True, mask_token=None) # functi
```

In [9]:
```
def text_from_ids(ids):
  return tf.strings.reduce_join(chars_from_ids(ids), axis=-1)
```

In [16]:
```
print(ids_from_chars('L'))
print(ids_from_chars('u'))
print(ids_from_chars('c'))
print(ids_from_chars('k'))
print(ids_from_chars('y'))
```

```
tf.Tensor(25, shape=(), dtype=int64)
tf.Tensor(60, shape=(), dtype=int64)
tf.Tensor(42, shape=(), dtype=int64)
tf.Tensor(50, shape=(), dtype=int64)
tf.Tensor(64, shape=(), dtype=int64)
```

In [17]:
```
ids= [25, 60, 42, 50, 64]
text_from_ids(ids)
```

Out[17]:
```
<tf.Tensor: shape=(), dtype=string, numpy=b'Lucky'>
```

In [23]:
```
s= tf.strings.unicode_split("Deepankar Sharma", 'UTF-8')
ids= ids_from_chars(s)
print(ids)
```

```
tf.Tensor([17 44 44 55 40 53 50 40 57  2 32 47 40 57 52 40], shape=(16,), dtype=int6
4)
```

In [24]:
```
text_from_ids(ids)
```

Out[24]:
```
<tf.Tensor: shape=(), dtype=string, numpy=b'Deepankar Sharma'>
```

Training Example

In [25]:
```
data_ids = ids_from_chars(tf.strings.unicode_split(text, 'UTF-8'))
data_ids
```

Out[25]:
```
<tf.Tensor: shape=(1115394,), dtype=int64, numpy=array([19, 48, 57, ..., 46,  9,  1],
dtype=int64)>
```

In [27]:
```
ids_dataset = tf.data.Dataset.from_tensor_slices(data_ids)
```

In [37]:
```
print(ids_dataset.take(1))
```

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

In [30]:
```
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 [31]:
```
seq_length = 100
```

In [34]:
```
sequences = ids_dataset.batch(seq_length+1, drop_remainder=True) # convert indivisual

for seq in sequences.take(1):
  print(chars_from_ids(seq)) # batch of first seq_length(100) characters
```

```
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 [35]:
```
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, spea
k.\n\nFirst Citizen:\nYou '
b'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\n\nFirs
t Citizen:\nFirst, you k'
b"now Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we know't.\n\nF
irst Citizen:\nLet us ki"
b"ll him, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo more tal
king 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'
```

In [38]:
```
def split_input_target(sequence):
    ''' This function will return the tuple of (X, y) where X is the input text and y
    input_text = sequence[:-1] # first character to second last character
    target_text = sequence[1:] # second character to last character
    return input_text, target_text
```

In [42]:
```
print(split_input_target("Deepankar"))
print(split_input_target("Sharma"))
```

```
('Deepanka', 'eepankar')
('Sharm', 'harma')
```

In [43]:
```
dataset = sequences.map(split_input_target) # applying on our batches
```

In [46]:
```
for input_example, target_example in dataset.take(5):
    print("Input :", text_from_ids(input_example).numpy())
    print("Target:", text_from_ids(target_example).numpy())
    print()
```

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

Input : b'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolve
d.\n\nFirst Citizen:\nFirst, you '
Target: b're all resolved rather to die than to famish?\n\nAll:\nResolved. resolve
d.\n\nFirst Citizen:\nFirst, you k'

Input : b"now Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we kno
w't.\n\nFirst Citizen:\nLet us k"
Target: b"ow Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we kno
w't.\n\nFirst Citizen:\nLet us ki"

Input : 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 "
Target: b"l him, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo m
ore talking on't; let it be d"

Input : b'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citi
zen:\nWe are accounted poor cit'
Target: b'ne: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citiz
en:\nWe are accounted poor citi'
```

Model Building

In [47]:
```
# Batch size
BATCH_SIZE = 64
# Buffer Size for shuffling the data set before batching it into batches of BATCH
BUFFER_SIZE = 10000

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

dataset
```

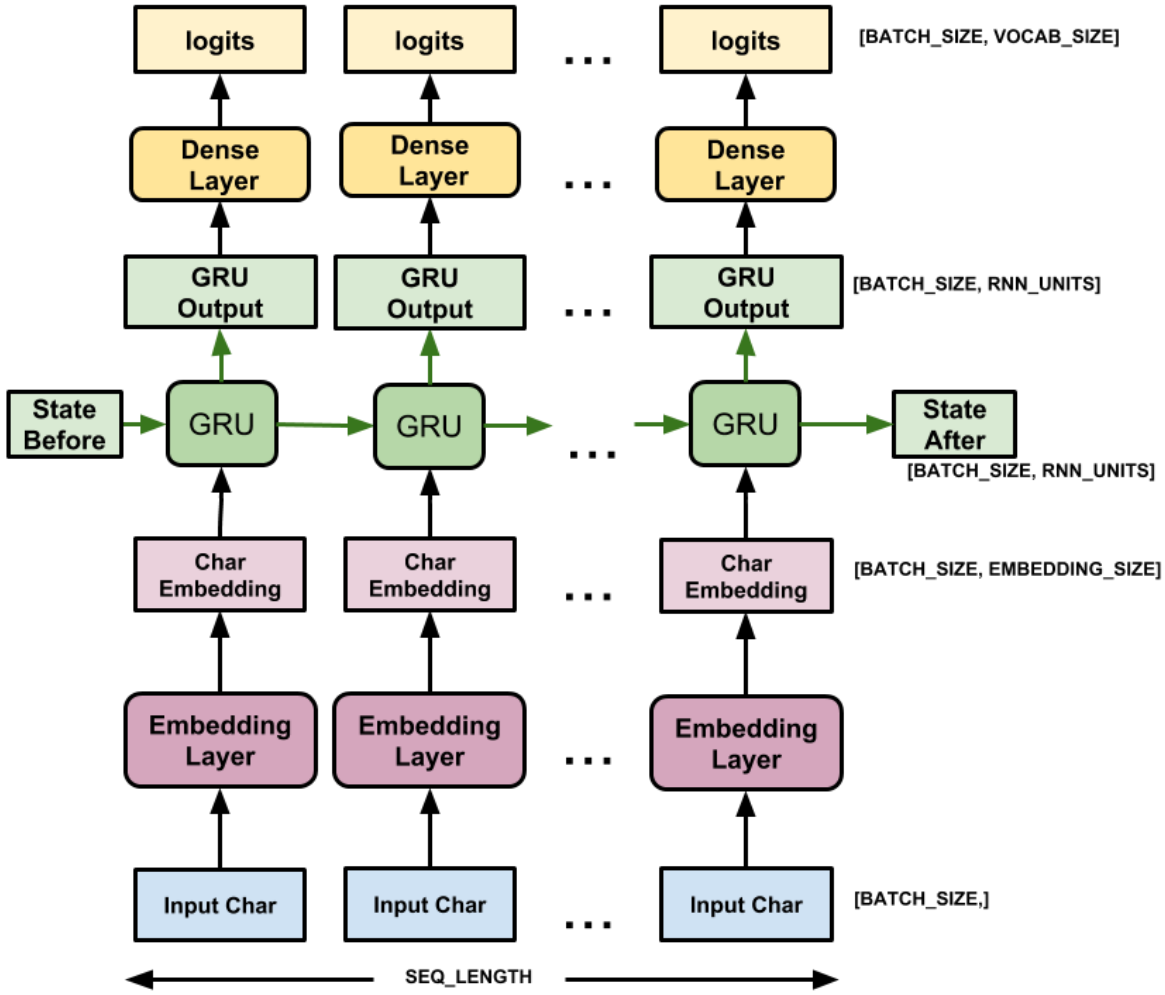
Out[47]:
```
<PrefetchDataset element_spec=(TensorSpec(shape=(64, 100), dtype=tf.int64, name=Non
e), TensorSpec(shape=(64, 100), dtype=tf.int64, name=None))>
```

In [48]:
```
# 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 [49]:
```
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)
    # Gated Recurrent Unit
    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 [50]:
```
model = MyModel(
    vocab_size=vocab_size,
    embedding_dim=embedding_dim,
    rnn_units=rnn_units)
```



In [56]:
```
dataset.take(1)
```

Out[56]:
```
<TakeDataset element_spec=(TensorSpec(shape=(64, 100), dtype=tf.int64, name=None), TensorSpec(shape=(64, 100), dtype=tf.int64, name=None))>
```

In [58]:
```
model.build((64,100))
model.summary()
```

```
Model: "my_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       multiple                  16896

 gru (GRU)                   multiple                  3938304

 dense (Dense)               multiple                  67650

=================================================================
Total params: 4,022,850
Trainable params: 4,022,850
Non-trainable params: 0
_________________________________________________________________
```

In [59]:
```
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 [ ]:

In [61]:
```
sampled_indices = tf.random.categorical(example_batch_predictions[0], num_samples=1)
sampled_indices = tf.squeeze(sampled_indices, axis=-1).numpy()
sampled_indices
```

Out[61]:
```
array([33, 62, 49,  6, 26, 28, 19, 11, 59, 36,  8, 30, 47, 33, 45, 27,  3,
       22,  6, 63, 22, 46,  8, 54, 14, 16, 57, 32, 44, 28, 23, 27, 51, 61,
       14,  9, 23,  8, 41, 30, 19, 21, 63, 52, 60, 16, 52,  9, 21, 40, 46,
       43, 38, 31, 42,  1, 29, 10, 63, 29, 47,  3, 27,  4, 41, 25,  5, 57,
       36, 41, 53, 58,  1,  6, 52,  5, 23, 42,  7, 26, 45, 33,  0,  8,  4,
       16, 26, 51, 59, 11, 11, 27, 10, 25,  3, 60, 47,  9, 43, 40],
      dtype=int64)
```

In [62]:
```
text_from_ids(sampled_indices)
```

Out[62]:
```
<tf.Tensor: shape=(), dtype=string, numpy=b"Twj'MOF:tW-QhTfN!I'xIg-oACrSeOJNlvA.J-bQF
HxmuCm.HagdYRc\nP3xPh!N$bL&rWbns\n'm&Jc,MfT[UNK]-$CMlt::N3L!uh.da">
```

In [63]:
```
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' is stale, and I am weary of it.\n\nKeeper:\nHelp, help, help!\n\nKING RICHARD I
I:\nHow now! what means de'

Next Char Predictions:
 b"Twj'MOF:tW-QhTfN!I'xIg-oACrSeOJNlvA.J-bQFHxmuCm.HagdYRc\nP3xPh!N$bL&rWbns\n'm&Jc,M
fT[UNK]-$CMlt::N3L!uh.da"
```

Model Training

In [64]:
```
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
```

In [65]:
```
example_batch_mean_loss = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_size, sequence
print("Mean loss:        ", example_batch_mean_loss)
```

```
Prediction shape:  (64, 100, 66)  # (batch_size, sequence_length, vocab_size)
Mean loss:         tf.Tensor(4.1899457, shape=(), dtype=float32)
```

In [66]:
```
tf.exp(example_batch_mean_loss).numpy()
```

Out[66]:
```
66.01921
```

In [67]:
```
model.compile(optimizer='adam', loss=loss)
```

In [68]:
```
# 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)
```

In [70]:

```
# EPOCHS = 20
EPOCHS = 50
history = model.fit(dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])
```

```
Epoch 1/50
172/172 [==============================] - 7s 33ms/step - loss: 0.6719
Epoch 2/50
172/172 [==============================] - 7s 38ms/step - loss: 0.6321
Epoch 3/50
172/172 [==============================] - 7s 39ms/step - loss: 0.5987
Epoch 4/50
172/172 [==============================] - 7s 39ms/step - loss: 0.5683
Epoch 5/50
172/172 [==============================] - 7s 39ms/step - loss: 0.5439
Epoch 6/50
172/172 [==============================] - 7s 39ms/step - loss: 0.5218
Epoch 7/50
172/172 [==============================] - 7s 38ms/step - loss: 0.5078
Epoch 8/50
172/172 [==============================] - 7s 38ms/step - loss: 0.4920
Epoch 9/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4766
Epoch 10/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4662
Epoch 11/50
172/172 [==============================] - 7s 38ms/step - loss: 0.4575
Epoch 12/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4510
Epoch 13/50
172/172 [==============================] - 8s 40ms/step - loss: 0.4449
Epoch 14/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4392
Epoch 15/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4318
Epoch 16/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4311
Epoch 17/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4282
Epoch 18/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4260
Epoch 19/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4212
Epoch 20/50
172/172 [==============================] - 8s 41ms/step - loss: 0.4189
Epoch 21/50
172/172 [==============================] - 8s 40ms/step - loss: 0.4208
Epoch 22/50
172/172 [==============================] - 8s 40ms/step - loss: 0.4195
Epoch 23/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4159
Epoch 24/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4197
Epoch 25/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4177
Epoch 26/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4173
Epoch 27/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4142
Epoch 28/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4140
Epoch 29/50
172/172 [==============================] - 8s 41ms/step - loss: 0.4146
Epoch 30/50
172/172 [==============================] - 7s 38ms/step - loss: 0.4143
```

```
Epoch 31/50
172/172 [==============================] - 7s 38ms/step - loss: 0.4155
Epoch 32/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4202
Epoch 33/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4248
Epoch 34/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4230
Epoch 35/50
172/172 [==============================] - 8s 40ms/step - loss: 0.4299
Epoch 36/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4304
Epoch 37/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4298
Epoch 38/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4301
Epoch 39/50
172/172 [==============================] - 8s 41ms/step - loss: 0.4356
Epoch 40/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4406
Epoch 41/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4401
Epoch 42/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4388
Epoch 43/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4417
Epoch 44/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4394
Epoch 45/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4421
Epoch 46/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4463
Epoch 47/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4521
Epoch 48/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4546
Epoch 49/50
172/172 [==============================] - 7s 39ms/step - loss: 0.4543
Epoch 50/50
172/172 [==============================] - 7s 40ms/step - loss: 0.4641
```

Model Testing

In [71]:

```python
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()

    # 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 "[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 [72]:
```
one_step_model = OneStep(model, chars_from_ids, ids_from_chars)
```

In [73]:
```
start = time.time()
states = None
next_char = tf.constant(['Deepankar: I will test this.'])
result = [next_char]

for n in range(100):
  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)
```

```
Deepankar: I will test this.

JULIET:
Yet let me weep for such a sight.

QUEEN MARGARET:
Those good thousand dukedom them but th

________________________________________________________________________________

Run time: 0.8605191707611084
```

Exporting the Text Generator

In [74]:
```
tf.saved_model.save(one_step_model, 'one_step')
one_step_reloaded = tf.saved_model.load('one_step')
```

```
WARNING:tensorflow:Skipping full serialization of Keras layer <__main__.OneStep objec
t at 0x000001F066AD3BB0>, because it is not built.
WARNING:absl:Found untraced functions such as gru_cell_layer_call_fn, gru_cell_layer_
call_and_return_conditional_losses while saving (showing 2 of 2). These functions wil
l not be directly callable after loading.
INFO:tensorflow:Assets written to: one_step\assets
INFO:tensorflow:Assets written to: one_step\assets
```

In [79]:
```
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:
Who end, was, sir, to wive it well.

PAULINA:
Indeed, my lord

DUKE OF YORK:
If God he knows not wh
```

In [ ]: