```
# Import libraries
import os
os.add_dll_directory("C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.8/bin")
os.add_dll_directory(r"C:\Users\tysha\Downloads\cudnn-windows-x86_64-8.6.0.163_cuda11-archive\cudnn-windows-x86_64-8.6.0.163_cuda11-archive\b
import tensorflow as tf
from tensorflow.keras.layers.experimental import preprocessing

import numpy as np
import os
import time
```

I. Parse Text Sources

First we'll load our text sources and create our vocabulary lists and encoders.

There are ways we could do this in pure python, but using the tensorflow data structures and libraries allow us to keep things super-optimized.

```
# Load file data
#path_to_file = tf.keras.utils.get_file('austen.txt', 'https://raw.githubusercontent.com/byui-cse/cse450-course/master/data/austen/austen.txt
#file_name = '/content/Rick_ml.txt'
text = open('Rick_ml.txt', 'rb').read().decode(encoding='utf-8')
text = open('riordan_egypt_ml.txt', 'rb').read().decode(encoding='utf-8')
#text = file_name
print('Length of text: {} characters'.format(len(text)))
```

```
    Length of text: 1899875 characters
```

```
# Verify the first part of our data
print(text[:200])
```

```
    WARNING
    This is a transcript of an audio recording. Carter andSadie Kane first made themselves known in a recording I
    received last year, which I transcribed as The RedPyramid. This second audio
```

```
# Now we'll get a list of the unique characters in the file. This will form the
# vocabulary of our network. There may be some characters we want to remove from this
# set as we refine the network.
vocab = sorted(set(text))
print('{} unique characters'.format(len(vocab)))
print(vocab)
```

```
    95 unique characters
    ['\n', '\r', ' ', '!', '&', '(', ')', '+', ',', '-', '.', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '=', '?', 'A
```

```
# Next, we'll encode encode these characters into numbers so we can use them
# with our neural network, then we'll create some mappings between the characters
# and their numeric representations
ids_from_chars = preprocessing.StringLookup(vocabulary=list(vocab))
chars_from_ids = tf.keras.layers.experimental.preprocessing.StringLookup(vocabulary=ids_from_chars.get_vocabulary(), invert=True)

# Here's a little helper function that we can use to turn a sequence of ids
# back into a string:
# turn them into a string:
def text_from_ids(ids):
  joinedTensor = tf.strings.reduce_join(chars_from_ids(ids), axis=-1)
  return joinedTensor.numpy().decode("utf-8")
```

```
# Now we'll verify that they work, by getting the code for "A", and then looking
# that up in reverse
testids = ids_from_chars(["T", "r", "u", "t", "h"])
testids
```

```
    <tf.Tensor: shape=(5,), dtype=int64, numpy=array([46, 72, 75, 74, 62], dtype=int64)>
```

```
chars_from_ids(testids)
```

```
    <tf.Tensor: shape=(5,), dtype=string, numpy=array([b'T', b'r', b'u', b't', b'h'], dtype=object)>
```

```
testString = text_from_ids( testids )
testString
```

```
'Truth'
```

II. Construct our training data

Next we need to construct our training data by building sentence chunks. Each chunk will consist of a sequence of characters and a corresponding "next sequence" of the same length showing what would happen if we move forward in the text. This "next sequence" becomes our target variable.

For example, if this were our text:

> It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.

And our sequence length was 10 with a step size of 1, our first chunk would be:

- Sequence: `It is a tr`
- Next Sequence: `t is a tru`

Our second chunk would be:

- Sequence: `t is a tru`
- Next Word: `is a trut`

```
# First, create a stream of encoded integers from our text
all_ids = ids_from_chars(tf.strings.unicode_split(text, 'UTF-8'))
all_ids
```

```
<tf.Tensor: shape=(1899875,), dtype=int64, numpy=array([2, 1, 2, ..., 1, 2, 1], dtype=int64)>
```

```
# Now, convert that into a tensorflow dataset
ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
```

```
# Finally, let's batch these sequences up into chunks for our training
seq_length = 100
sequences = ids_dataset.batch(seq_length+1, drop_remainder=True)

# This function will generate our sequence pairs:
def split_input_target(sequence):
    input_text = sequence[:-1]
    target_text = sequence[1:]
    return input_text, target_text

# Call the function for every sequence in our list to create a new dataset
# of input->target pairs
dataset = sequences.map(split_input_target)
```

```
# Verify our sequences
for input_example, target_example in  dataset.take(1):
    print("Input: ", text_from_ids(input_example))
    print("--------")
    print("Target: ", text_from_ids(target_example))
```

```
Input:

WARNING
This is a transcript of an audio recording. Carter andSadie Kane first made themselves
--------
Target:

WARNING
This is a transcript of an audio recording. Carter andSadie Kane first made themselves k
```

```
# Finally, we'll randomize the sequences so that we don't just memorize the books
# in the order they were written, then build a new streaming dataset from that.
# Using a streaming dataset allows us to pass the data to our network bit by bit,
# rather than keeping it all in memory. We'll set it to figure out how much data
# to prefetch in the background.

BATCH_SIZE = 64
BUFFER_SIZE = 10000
```

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

dataset
```

```
<PrefetchDataset element_spec=(TensorSpec(shape=(64, 100), dtype=tf.int64, name=None), TensorSpec(shape=(64, 100), dtype=tf.int64,
name=None))>
```

III. Build the model

Next, we'll build our model. Up until this point, you've been using the Keras symbolic, or imperative API for creating your models. Doing something like:

```
model = tf.keras.models.Sequentla()
model.add(tf.keras.layers.Dense(80, activation='relu))
etc...
```

However, tensorflow has another way to build models called the Functional API, which gives us a lot more control over what happens inside the model. You can read more about the differences and when to use each here.

We'll use the functional API for our RNN in this example. This will involve defining our model as a custom subclass of `tf.keras.Model`.

If you're not familiar with classes in python, you might want to review this quick tutorial, as well as this one on class inheritance.

Using a functional model is important for our situation because we're not just training it to predict a single character for a single sequence, but as we make predictions with it, we need it to remember those predictions as use that memory as it makes new predictions.

```
# Create our custom model. Given a sequence of characters, this
# model's job is to predict what character should come next.
class AustenTextModel(tf.keras.Model):

  # This is our class constructor method, it will be executed when
  # we first create an instance of the class
  def __init__(self, vocab_size, embedding_dim, rnn_units):
    super().__init__(self)

    # Our model will have three layers:

    # 1. An embedding layer that handles the encoding of our vocabulary into
    #    a vector of values suitable for a neural network
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)

    # 2. A GRU layer that handles the "memory" aspects of our RNN. If you're
    #    wondering why we use GRU instead of LSTM, and whether LSTM is better,
    #    take a look at this article: https://datascience.stackexchange.com/questions/14581/when-to-use-gru-over-lstm
    #    then consider trying out LSTM instead (or in addition to!)
    self.gru = tf.keras.layers.GRU(rnn_units, return_sequences=True, return_state=True)

    # 3. Our output layer that will give us a set of probabilities for each
    #    character in our vocabulary.
    self.dense = tf.keras.layers.Dense(vocab_size)

  # This function will be executed for each epoch of our training. Here
  # we will manually feed information from one layer of our network to the
  # next.
  def call(self, inputs, states=None, return_state=False, training=False):
    x = inputs

    # 1. Feed the inputs into the embedding layer, and tell it if we are
    #    training or predicting
    x = self.embedding(x, training=training)

    # 2. If we don't have any state in memory yet, get the initial random state
    #    from our GRUI layer.
    if states is None:
      states = self.gru.get_initial_state(x)

    # 3. Now, feed the vectorized input along with the current state of memory
    #    into the gru layer.
    x, states = self.gru(x, initial_state=states, training=training)
    # 4. Finally, pass the results on to the dense layer
    x = self.dense(x, training=training)

    # 5. Return the results
    if return_state:
      return x, states
    else:
      return x


# Create an instance of our model
vocab_size=len(ids_from_chars.get_vocabulary())
embedding_dim = 512
rnn_units = 2048

model = AustenTextModel(vocab_size, embedding_dim, rnn_units)
```

```
# Verify the output of our model is correct by running one sample through
# This will also compile the model for us. This step will take a bit.
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, 96) # (batch_size, sequence_length, vocab_size)
```

```
# Now let's view the model summary
model.summary()
```

```
Model: "austen_text_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       multiple                  49152

 gru (GRU)                   multiple                  15740928

 dense (Dense)               multiple                  196704

=================================================================
Total params: 15,986,784
Trainable params: 15,986,784
Non-trainable params: 0
_________________________________________________________________
```

IV. Train the model

For our purposes, we'll be using categorical cross entropy as our loss function*. Also, our model will be outputting "logits" rather than normalized probabilities, because we'll be doing further transformations on the output later.

* Note that since our model deals with integer encoding rather than one-hot encoding, we'll specifically be using sparse categorical cross entropy.

```
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam', loss=loss)

history = model.fit(dataset, epochs=10)
```

```
Epoch 1/10
293/293 [==============================] - 19s 58ms/step - loss: 2.3625
Epoch 2/10
293/293 [==============================] - 17s 57ms/step - loss: 1.5434
Epoch 3/10
293/293 [==============================] - 17s 55ms/step - loss: 1.3268
Epoch 4/10
293/293 [==============================] - 16s 51ms/step - loss: 1.2191
Epoch 5/10
293/293 [==============================] - 15s 49ms/step - loss: 1.1369
Epoch 6/10
293/293 [==============================] - 15s 49ms/step - loss: 1.0619
Epoch 7/10
293/293 [==============================] - 16s 52ms/step - loss: 0.9841
Epoch 8/10
293/293 [==============================] - 16s 51ms/step - loss: 0.9061
Epoch 9/10
```

```
293/293 [==============================] - 15s 49ms/step - loss: 0.8289
Epoch 10/10
293/293 [==============================] - 16s 51ms/step - loss: 0.7605
```

V. Use the model

Now that our model has been trained, we can use it to generate text. As mentioned earlier, to do so we have to keep track of its internal state, or memory, so that we can use previous text predictions to inform later ones.

However, with RNN generated text, if we always just pick the character with the highest probability, our model tends to get stuck in a loop. So instead we will create a probability distribution of characters for each step, and then sample from that distribution. We can add some variation to this using a paramter known as "temperature".

```
# Here's the code we'll use to sample for us. It has some extra steps to apply
# the temperature to the distribution, and to make sure we don't get empty
# characters in our text. Most importantly, it will keep track of our model
# state for us.

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,validate_indices=False)

  @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)

    # Return the characters and model state.
    return chars_from_ids(predicted_ids), states



# Create an instance of the character generator
one_step_model = OneStep(model, chars_from_ids, ids_from_chars)

# Now, let's generate a 1000 character chapter by giving our model "Chapter 1"
# as its starting text
states = None
next_char = tf.constant(['Chapter 1'])
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)

# Print the results formatted.
print(result[0].numpy().decode('utf-8'))
```

```
Chapter 1yes. Coming
at Brooklyn House were smashing the room, resisting their faces from another lip.
The pain in pain back shape the window and landed into combat. I couldn’t help the words,
but I didn’t know howIshim were the guidefulate birds. The House of Life had been followed. It
widened in be or me at excelbent tuiled into a ceiling against the rocks in the largest door.
When I left on the flowers disk, quite adultter. Behind her help, the hieroglyphs
Khufu stared at me.
“Now, come for hill, weak your straggle when he turn into Accident and get youing the House have
counted on the surface. Tyeasked Bast was dark black on the left-apolic
a red shape. They belowed in my feet, but its neck
as a combinated long hall of cornsperfement.
“Zia!” Carter and I snumped. “That cut it! it’s our dead—just her
duck.
“Carter, get ready for something wants,” I said. “Is that the protective pharaoh would we ake both honest. I’ll luck you
to the riverbank.”
The cracks in his pyramid inside.
```

VI. Next Steps

This is a very simple model with one GRU layer and then an output layer. However, considering how simple it is and the fact that we are predicting outputs character by character, the text it produces is pretty amazing. Though it still has a long way to go before publication.

There are many other RNN architectures you could try, such as adding additional hidden dense layers, replacing GRU with one or more LSTM layers, combining GRU and LSTM, etc...

You could also experiment with better text cleanup to make sure odd punctuation doesn't appear, or finding longer texts to use. If you combine texts from two authors, what happens? Can you generate a Jane Austen stageplay by combining austen and shakespeare texts?

Finally, there are a number of hyperparameters to tweak, such as temperature, epochs, batch size, sequence length, etc...

✓ 3s completed at 9:57 PM