

# Applied Math 216, Week 8. Section Notebook

This section Notebook has two parts:

1. First part copied from tensorflow tutorial on RNN's
2. Second part asks you to use the same type of code to predict the next letter in a DNA sequence.

## Part I

**Copyright 2019 The TensorFlow Authors.**

```
In [ ]: #@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.
```

## Text generation with an RNN



[View on TensorFlow.org](https://www.tensorflow.org/tutorials/text/text_generation)

([https://www.tensorflow.org/tutorials/text/text\\_generation](https://www.tensorflow.org/tutorials/text/text_generation))



(<https://colab.research.google.com/github/tensorflow/docs/blob/master>)

This tutorial demonstrates how to generate text using a character-based RNN. We will work with a dataset of Shakespeare's writing from Andrej Karpathy's [The Unreasonable Effectiveness of Recurrent Neural Networks](http://karpathy.github.io/2015/05/21/rnn-effectiveness/) (<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>). Given a sequence of characters from this data ("Shakespeare"), 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*. If running locally make sure TensorFlow version  $\geq 1.11$ .

This tutorial includes runnable code implemented using [tf.keras](https://www.tensorflow.org/programmers_guide/keras) ([https://www.tensorflow.org/programmers\\_guide/keras](https://www.tensorflow.org/programmers_guide/keras)) and [eager execution](https://www.tensorflow.org/programmers_guide/eager) ([https://www.tensorflow.org/programmers\\_guide/eager](https://www.tensorflow.org/programmers_guide/eager)). The following is sample output when the model in this tutorial trained for 30 epochs, and started with the string "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, n  
o, 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 [ ]: from __future__ import absolute_import, division, print_function, unicode_literals

try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
except Exception:
    pass
import tensorflow as tf

import numpy as np
import os
import time
```

### Download the Shakespeare dataset

Change the following line to run this code on your own data.

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

### Read the data

First, look in the text:

```
In [ ]: # 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('Length of text: {} characters'.format(len(text)))
```

Length of text: 1115394 characters

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

65 unique characters

## Process the text

### Vectorize the text

Before training, we need to map strings to a numerical representation. Create two lookup tables: one mapping characters to numbers, and another for numbers to characters.

```
In [ ]: # Creating a mapping from unique characters to indices
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

text_as_int = np.array([char2idx[c] for c in text])
```

Now we have an integer representation for each character. Notice that we mapped the character as indexes from 0 to `len(unique)` .

```
In [ ]: print('{')
        for char,_ in zip(char2idx, range(20)):
            print('  {4s}: {3d}'.format(repr(char), char2idx[char]))
        print('  ...\\n')
```

```
{
  '\\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,
  ...
}
```

```
In [ ]: # Show how the first 13 characters from the text are mapped to integers
        print ('{} ---- characters mapped to int ---- > {}'.format(repr(text[:13]), text_as_int[:13]))
```

```
'First Citizen' ---- characters mapped to int ---- > [18 47 56 57 58 1
15 47 58 47 64 43 52]
```

## 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 [ ]: # The maximum length sentence we want for a single input in characters
seq_length = 100
examples_per_epoch = len(text)//(seq_length+1)

# Create training examples / targets
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

for i in char_dataset.take(5):
    print(idx2char[i.numpy()])
```

F  
i  
r  
s  
t

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

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

for item in sequences.take(5):
    print(repr(''.join(idx2char[item.numpy()])))
```

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

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

        shake_dataset = sequences.map(split_input_target)
```

Print the first examples input and target values:

```
In [ ]: for input_example, target_example in shake_dataset.take(1):
        print('Input data: ', repr(''.join(idx2char[input_example.numpy()])))
        print('Target data: ', repr(''.join(idx2char[target_example.numpy()])))
```

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

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 tries 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 [ ]: for i, (input_idx, target_idx) in enumerate(zip(input_example[:5], target_example[:5])):
        print("Step {:4d}".format(i))
        print("  input: {} ({:s})".format(input_idx, repr(idx2char[input_idx])))
        print("  expected output: {} ({:s})".format(target_idx, repr(idx2char[target_idx])))
```

```
Step      0
  input: 18 ('F')
  expected output: 47 ('i')
Step      1
  input: 47 ('i')
  expected output: 56 ('r')
Step      2
  input: 56 ('r')
  expected output: 57 ('s')
Step      3
  input: 57 ('s')
  expected output: 58 ('t')
Step      4
  input: 58 ('t')
  expected 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 batches.

```
In [ ]: # 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

        shake_dataset = shake_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)

        shake_dataset
```

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

## 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 [ ]: # Length of the vocabulary in chars
        vocab_size = len(vocab)

        # The embedding dimension
        embedding_dim = 256


        # Number of RNN units
        rnn_units = 1024
```



```
In [ ]: def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
        model = tf.keras.Sequential([
            tf.keras.layers.Embedding(vocab_size, embedding_dim,
                                      batch_input_shape=[batch_size, None]),
            tf.keras.layers.GRU(rnn_units,
                                return_sequences=True,
                                stateful=True,
                                recurrent_initializer='glorot_uniform'),
            tf.keras.layers.Dense(vocab_size)
        ])
        return model
```

```
In [ ]: model = build_model(
        vocab_size = len(vocab),
        embedding_dim=embedding_dim,
        rnn_units=rnn_units,
        batch_size=BATCH_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:

 A drawing of the data passing through the model

## Try the model

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

First check the shape of the output:

```
In [ ]: for input_example_batch, target_example_batch in shake_dataset.take(1):
        example_batch_predictions = model(input_example_batch)
        print(example_batch_predictions.shape, "# (batch_size, sequence_length, vocab_size)")

(64, 100, 65) # (batch_size, sequence_length, vocab_size)
```

```
In [ ]: example_batch_predictions[0][0]

Out[ ]: <tf.Tensor: shape=(65,), dtype=float32, numpy=
array([ 0.01391688,  0.00550602, -0.00446113,  0.00427705,  0.0035988 ,
        -0.00688676,  0.00710274, -0.00335482,  0.00725956,  0.00966294,
         0.00378431, -0.00480664,  0.00918738, -0.00957012, -0.00449718,
        -0.00409246, -0.00361451,  0.00484277,  0.00010024,  0.01123031,
        -0.00290692, -0.00888212,  0.0009504 , -0.00236033,  0.00187743,
         0.00218447,  0.00923999,  0.01215473,  0.00360458, -0.01012366,
         0.00443463, -0.00015481,  0.00078903, -0.01383883,  0.00563169,
        -0.00095586, -0.00378979,  0.00419476,  0.00937213,  0.00076684,
         0.00969439,  0.00113505, -0.01190644, -0.00556837, -0.00671721,
        -0.00933228,  0.00587396,  0.00361739,  0.00553251, -0.00523378,
        -0.00370089, -0.00814032,  0.00345241, -0.00042706, -0.00015358,
         0.00724276, -0.00281215,  0.00582988, -0.00096875,  0.01829365,
        -0.01100285,  0.00114623, -0.00855274, -0.0032624 , -0.011387
      ],
      dtype=float32)>
```

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

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

Model: "sequential_14"

Layer (type)                Output Shape                Param #
=====
embedding_14 (Embedding)    (64, None, 256)            16640
gru_9 (GRU)                 (64, None, 1024)           3938304
dense_14 (Dense)            (64, None, 65)              66625
=====
Total params: 4,021,569
Trainable params: 4,021,569
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.

Try it for the first example in the batch:

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

```
Out[ ]: array([[17, 47, 52, 31, 43, 57, 12, 44, 37, 50,  0, 15, 54, 20, 53, 28,
 5,
          61, 39, 28, 17, 48, 33, 12, 44, 48, 29, 62, 56, 63,  1,  4, 46,
18,
          18,  2, 24, 62, 55, 41, 50, 16,  3, 36, 20,  0, 24, 57,  0, 51,
8,
          28, 43, 44, 30,  0,  2, 43, 62, 29, 32, 62, 29, 57, 59, 43, 11,
53,
          38, 41, 46, 44, 11, 44, 13, 22, 35, 38, 64, 20, 25, 62, 15, 56,
33,
          9, 40, 18, 26, 61, 12, 52, 30,  3, 58, 18,  8, 45, 52, 21])
```

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

```
In [ ]: print("Input: \n", repr("".join(idx2char[input_example_batch[0]])))
print()
print("Next Char Predictions: \n", repr("".join(idx2char[sampled_indices
])))
```

Input:

'there thy kingdom is.\n\nRIVERS:\nMy Lord of Gloucester, in those bus  
y days\nWhich here you urge to prov'

Next Char Predictions:

"EinSes?fYl\nCpHoP'waPEjU?fjQxry &hFF!LxqclD\$XH\nLs\nm.PefR\n!exQTxQsu  
e;oZchf;fAJWZzHMxCrU3bFNw?nR\$tF.gnI"

## 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 our model returns logits, we need to set the `from_logits` flag.

```
In [ ]: def loss(labels, logits):  
        return tf.keras.losses.sparse_categorical_crossentropy(labels, logits,  
            from_logits=True)  
  
example_batch_loss = loss(target_example_batch, example_batch_predictions)  
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_  
size, sequence_length, vocab_size)")  
print("scalar_loss:      ", example_batch_loss.numpy().mean())  
  
Prediction shape: (64, 100, 65) # (batch_size, sequence_length, vocab  
_size)  
scalar_loss:      4.174211
```

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

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

## Configure checkpoints

Use a `tf.keras.callbacks.ModelCheckpoint` to ensure that checkpoints are saved during training:

```
In [ ]: # 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

To keep training time reasonable, use 10 epochs to train the model. In Colab, set the runtime to GPU for faster training.

```
In [ ]: EPOCHS=10
```

```
In [ ]: history = model.fit(shake_dataset, epochs=EPOCHS, callbacks=[checkpoint_
callback])

Epoch 1/10
172/172 [=====] - 26s 130ms/step - loss: 2.702
5
Epoch 2/10
172/172 [=====] - 25s 135ms/step - loss: 1.968
7
Epoch 3/10
172/172 [=====] - 24s 132ms/step - loss: 1.697
0
Epoch 4/10
172/172 [=====] - 25s 133ms/step - loss: 1.545
9
Epoch 5/10
172/172 [=====] - 24s 133ms/step - loss: 1.456
6
Epoch 6/10
172/172 [=====] - 24s 133ms/step - loss: 1.396
7
Epoch 7/10
172/172 [=====] - 25s 133ms/step - loss: 1.351
0
Epoch 8/10
172/172 [=====] - 24s 133ms/step - loss: 1.312
5
Epoch 9/10
172/172 [=====] - 24s 132ms/step - loss: 1.278
2
Epoch 10/10
172/172 [=====] - 24s 132ms/step - loss: 1.245
5
```

## Generate text

### Restore the latest checkpoint

To keep this prediction step simple, use a batch size of 1.

Because of the way the RNN state is passed from timestep to timestep, the model only accepts a fixed batch size once built.

To run the model with a different `batch_size`, we need to rebuild the model and restore the weights from the checkpoint.

```
In [ ]: tf.train.latest_checkpoint(checkpoint_dir)

Out[ ]: './training_checkpoints/ckpt_10'
```

```
In [ ]: shake_model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)

shake_model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

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

```
In [ ]: shake_model.summary()
```

Model: "sequential\_15"

Layer (type)	Output Shape	Param #
=====		
embedding_15 (Embedding)	(1, None, 256)	16640
gru_10 (GRU)	(1, None, 1024)	3938304
dense_15 (Dense)	(1, None, 65)	66625
=====		
Total params: 4,021,569		
Trainable params: 4,021,569		
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.

 To generate text the model's output is fed back to the input

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 [ ]: def generate_text(model, start_string):
        # Evaluation step (generating text using the learned model)

        # Number of characters to generate
        num_generate = 1000

        # Converting our start string to numbers (vectorizing)
        input_eval = [char2idx[s] for s in start_string]
        input_eval = tf.expand_dims(input_eval, 0)

        # Empty string to store our results
        text_generated = []

        # Low temperatures results in more predictable text.
        # Higher temperatures results in more surprising text.
        # Experiment to find the best setting.
        temperature = 1.0

        # Here batch size == 1
        model.reset_states()
        for i in range(num_generate):
            predictions = model(input_eval)
            # remove the batch dimension
            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(idx2char[predicted_id])

        return (start_string + ''.join(text_generated))

```

```
In [ ]: print(generate_text(shake_model, start_string="ROMEO: "))
```



WARNING:tensorflow:Detecting that an object or model or tf.train.Checkpoint is being deleted with unrestored values. See the following logs for the specific values in question. To silence these warnings, use `status.expect\_partial()`. See [https://www.tensorflow.org/api\\_docs/python/tf/train/Checkpoint#restore](https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint#restore) for details about the status object returned by the restore function.

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.iter

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.beta\_1

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.beta\_2

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.decay

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.learning\_rate

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-0.embeddings

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-2.kernel

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-2.bias

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-1.cell.kernel

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-1.cell.recurrent\_kernel

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-1.cell.bias

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-0.embeddings

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-2.kernel

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-2.bias

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-1.cell.kernel

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-1.cell.recurrent\_kernel

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-1.cell.bias

ROMEO: 'til a great third from hence?

Away, for, if thou tell'st very other, if I

BUCKINGHAM:

It is not that: but it is pophecious kings

That they that rather course.

CLIFFORD:

Come, well mer,  
The bloody man but even in her  
And his reporture both our times to touch grow:  
So face these quarely keepery out  
of majesty reason  
Since of a worwhit be a down conceivority.

POLIXENES:

Master, you  
but fetch thee well thee now;  
For evily wont thus time to use of, I have hither  
To seen him not what I have but said,  
I am such en, in womb weet his good.

ISABELLA:

Theice to hon.

PETRUCHIO:

Go, call him a Church in mine own gentlemen,  
And that I'll stay all eyes: go,  
Then.

QUEEN ELIZABETH:

Silence, we will show the an thy complare  
Yourself, by land waves--twelve me  
begin: I'll be the world of two another  
And cannot beg,  
And warilings inkissing by this proud renowness on  
' bedally denders, thou whats and good  
therial hands a kind of wine: 'I would thou wilt, by my father  
You were made you confessor: thy w

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

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

## Advanced: Customized Training

The above training procedure is simple, but does not give you much control.

So now that you've seen how to run the model manually let's unpack the training loop, and implement it ourselves. This gives a starting point if, for example, to implement *curriculum learning* to help stabilize the model's open-loop output.

We will use `tf.GradientTape` to track the gradients. You can learn more about this approach by reading the [eager execution guide \(https://www.tensorflow.org/guide/eager\)](https://www.tensorflow.org/guide/eager).

The procedure works as follows:

- First, initialize the RNN state. We do this by calling the `tf.keras.Model.reset_states` method.
- Next, iterate over the dataset (batch by batch) and calculate the *predictions* associated with each.
- Open a `tf.GradientTape`, and calculate the predictions and loss in that context.
- Calculate the gradients of the loss with respect to the model variables using the `tf.GradientTape.grads` method.
- Finally, take a step downwards by using the optimizer's `tf.train.Optimizer.apply_gradients` method.

```
In [ ]: model = build_model(
        vocab_size = len(vocab),
        embedding_dim=embedding_dim,
        rnn_units=rnn_units,
        batch_size=BATCH_SIZE)
```

```
In [ ]: optimizer = tf.keras.optimizers.Adam()
```

```
In [ ]: @tf.function
def train_step(inp, target):
    with tf.GradientTape() as tape:
        predictions = model(inp)
        loss = tf.reduce_mean(
            tf.keras.losses.sparse_categorical_crossentropy(
                target, predictions, from_logits=True))
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

    return loss
```

```
In [ ]: # Training step
EPOCHS = 10

for epoch in range(EPOCHS):
    start = time.time()

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

    for (batch_n, (inp, target)) in enumerate(shake_dataset):
        loss = train_step(inp, target)

        if batch_n % 100 == 0:
            template = 'Epoch {} Batch {} Loss {}'
            print(template.format(epoch+1, batch_n, 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 4.184297561645508  
Epoch 1 Batch 100 Loss 2.394624710083008  
Epoch 1 Loss 2.2624  
Time taken for 1 epoch 24.101694345474243 sec

Epoch 2 Batch 0 Loss 2.873333692550659  
Epoch 2 Batch 100 Loss 2.0657386779785156  
Epoch 2 Loss 2.0052  
Time taken for 1 epoch 23.05690336227417 sec

Epoch 3 Batch 0 Loss 1.9917385578155518  
Epoch 3 Batch 100 Loss 1.9189666509628296  
Epoch 3 Loss 1.8440  
Time taken for 1 epoch 23.04245162010193 sec

Epoch 4 Batch 0 Loss 1.8099451065063477  
Epoch 4 Batch 100 Loss 1.712236762046814  
Epoch 4 Loss 1.7586  
Time taken for 1 epoch 23.07850170135498 sec

Epoch 5 Batch 0 Loss 1.6885157823562622  
Epoch 5 Batch 100 Loss 1.6762681007385254  
Epoch 5 Loss 1.6133  
Time taken for 1 epoch 22.987914085388184 sec

Epoch 6 Batch 0 Loss 1.6110284328460693  
Epoch 6 Batch 100 Loss 1.563071608543396  
Epoch 6 Loss 1.5488  
Time taken for 1 epoch 23.04976224899292 sec

Epoch 7 Batch 0 Loss 1.5220098495483398  
Epoch 7 Batch 100 Loss 1.5318055152893066  
Epoch 7 Loss 1.5304  
Time taken for 1 epoch 23.240399599075317 sec

Epoch 8 Batch 0 Loss 1.4942811727523804  
Epoch 8 Batch 100 Loss 1.495859980583191  
Epoch 8 Loss 1.4986  
Time taken for 1 epoch 23.052624464035034 sec

Epoch 9 Batch 0 Loss 1.4890881776809692  
Epoch 9 Batch 100 Loss 1.4523128271102905  
Epoch 9 Loss 1.4478  
Time taken for 1 epoch 22.99792790412903 sec

Epoch 10 Batch 0 Loss 1.4467962980270386  
Epoch 10 Batch 100 Loss 1.4738496541976929  
Epoch 10 Loss 1.4019  
Time taken for 1 epoch 23.164217948913574 sec

## Example of Drosophila DNA sequence

Let's try the same problem with the genome from drosophila. We are going to do something totally crazy.

We will download the entire genome of drosophila DNA sequences. We will take the full sequence, break it into pieces of length 100. Our prediction problem is to predict

$$P(x_{101} | x_1, x_2, \dots, x_{100}).$$

Namely, find the probability of the next element of the sequence given the previous 100 characters. We will then simply state that the sequence is

$$\operatorname{argmax}_A P(x_{101} | x_1 \dots x_{100}),$$

where the argmax is over the letters of the alphabet.

Using the same neural network architecture as above, we will see that without much training we can do a reasonably good job!

```
In [ ]: !wget ftp://ftp.fruitfly.org/pub/download/compressed/dmel_release5.tgz
        !tar xf dmel_release5.tgz

--2022-03-30 00:56:38--  ftp://ftp.fruitfly.org/pub/download/compressed/dmel_release5.tgz
                        => 'dmel_release5.tgz.2'
Resolving ftp.fruitfly.org (ftp.fruitfly.org)... 128.3.61.31
Connecting to ftp.fruitfly.org (ftp.fruitfly.org)|128.3.61.31|:21... connected.
Logging in as anonymous ... Logged in!
==> SYST ... done.      ==> PWD ... done.
==> TYPE I ... done.   ==> CWD (1) /pub/download/compressed ... done.
==> SIZE dmel_release5.tgz ... 48335797
==> PASV ... done.     ==> RETR dmel_release5.tgz ... done.
Length: 48335797 (46M) (unauthoritative)

dmel_release5.tgz.2 100%[=====>] 46.10M 12.6MB/s in
3.7s

2022-03-30 00:56:44 (12.6 MB/s) - 'dmel_release5.tgz.2' saved [48335797]
```

```
In [ ]: with open('/content/Dmel_Release5/na_2LHet.dmel.RELEASE5', 'r') as f:
        content = f.readlines()
```

```
In [ ]: text = ''.join([content[i][:-1] for i in range(1, len(content)-10)])
        # length of text is the number of characters in it
        print('Length of text: {} characters'.format(len(text)))
```

Length of text: 368400 characters

```
In [ ]: text_test = ''.join([content[i][:-1] for i in range(len(content)-10, len(
content)-1)])
```

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

5 unique characters
```

```
In [ ]: vocab
```

```
Out[ ]: ['A', 'C', 'G', 'N', 'T']
```

```
In [ ]: # Creating a mapping from unique characters to indices
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

text_as_int = np.array([char2idx[c] for c in text])
```

```
In [ ]: print('{}')
for char, _ in zip(char2idx, range(5)):
    print('  {:4s}: {:3d}'.format(repr(char), char2idx[char]))
print('{}')

{
  'A' :    0,
  'C' :    1,
  'G' :    2,
  'N' :    3,
  'T' :    4,
}
```

```
In [ ]: # Show how the first 13 characters from the text are mapped to integers
print ('{} ---- characters mapped to int ---- > {}'.format(repr(text[:13]), text_as_int[:13]))

'TTCATCTTTCGTC' ---- characters mapped to int ---- > [4 4 1 0 4 1 4 4 4
1 2 4 1]
```

```
In [ ]: # The maximum length of base pairs for a single input
seq_length = 100
examples_per_epoch = len(text)//(seq_length+1)

# Create training examples / targets
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)

for i in char_dataset.take(5):
    print(idx2char[i.numpy()])
```

```
T
T
C
A
T
```

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

for item in sequences.take(5):
    print(repr(''.join(idx2char[item.numpy()])))
```

```
'TTCATCTTTCGTCTTTATGAAAAACACAAACATGTATCTCGATTTGTCATCTACGAAGGTGACAAAATAG
CGTGACCTCCAACAGACGTCGTTTGTATTG'
'GACCACAGATGTCACATATGCACAGTACTTCATTTGTGCGATTTCCAGAGGCTTTTGGGAAACCTTTACA
CATATCTTGCTCTTTGCGCAACTCAACAGT'
'CATTTTCTTTTGCGCCACTTAAATATTTTTCTTACTGAGTACCATTTCTTGTGTTGATCATTTTGTTCAA
GCTTGCAAAGTTTAAATGACCGAATCTCTGG'
'TGGCACTTCTTATTAAATTTGTTTTTAAAAAAGTCATTAAAGTTTTTAAAAATCGATCTCTCACCGC
ACAAATGAGCCGATGCTCCCGAATCAAGGCA'
'CCACATATTTTTTGGCAATTCGCCAAGCTGTACAGTTGAACAGAGCACAGAGAATGGATTTCTCGGTATT
TAAGTTTCGCTCTCTCTATGTTCTCCTTTC'
```

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

DNA_dataset = sequences.map(split_input_target)
```

```
In [ ]: for input_example, target_example in DNA_dataset.take(1):
    print('Input data: ', repr(''.join(idx2char[input_example.numpy()])))
    print('Target data:', repr(''.join(idx2char[target_example.numpy
()])))
```

```
Input data: 'TTCATCTTTCGTCTTTATGAAAAACACAAACATGTATCTCGATTTGTCATCTACGAA
GGTGACAAAATAGCGTGACCTCCAACAGACGTCGTTTGTATT'
Target data: 'TCATCTTTCGTCTTTATGAAAAACACAAACATGTATCTCGATTTGTCATCTACGAAG
GTGACAAAATAGCGTGACCTCCAACAGACGTCGTTTGTATTG'
```



```
In [ ]: for i, (input_idx, target_idx) in enumerate(zip(input_example[:5], target_example[:5])):
    print("Step {:4d}".format(i))
    print("  input: {} ({:s})".format(input_idx, repr(idx2char[input_idx])))
    print("  expected output: {} ({:s})".format(target_idx, repr(idx2char[target_idx])))
```

```
Step    0
  input: 4 ('T')
  expected output: 4 ('T')
Step    1
  input: 4 ('T')
  expected output: 1 ('C')
Step    2
  input: 1 ('C')
  expected output: 0 ('A')
Step    3
  input: 0 ('A')
  expected output: 4 ('T')
Step    4
  input: 4 ('T')
  expected output: 1 ('C')
```

```
In [ ]: # 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

DNA_dataset = DNA_dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)

DNA_dataset
```

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

## Build The Model

```
In [ ]: # Length of the vocabulary in chars
vocab_size = len(vocab)

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

```
In [ ]: tf.keras.layers.Embedding?
```

```
In [ ]: def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
        model = tf.keras.Sequential([
            tf.keras.layers.Embedding(vocab_size, embedding_dim,
                                      batch_input_shape=[batch_size, None]),
            tf.keras.layers.GRU(rnn_units,
                               return_sequences=True,
                               stateful=True,
                               recurrent_initializer='glorot_uniform'),
            tf.keras.layers.Dense(vocab_size)
        ])
        return model
```

```
In [ ]: model = build_model(
        vocab_size = len(vocab),
        embedding_dim=embedding_dim,
        rnn_units=rnn_units,
        batch_size=BATCH_SIZE)
```

```
In [ ]: for input_example_batch, target_example_batch in DNA_dataset.take(1):
        example_batch_predictions = model(input_example_batch)
        print(example_batch_predictions.shape, "# (batch_size, sequence_length, vocab_size)")
```

```
(64, 100, 5) # (batch_size, sequence_length, vocab_size)
```

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

```
Model: "sequential_20"
```

Layer (type)	Output Shape	Param #
=====		
embedding_20 (Embedding)	(64, None, 256)	1280
gru_12 (GRU)	(64, None, 1024)	3938304
dense_20 (Dense)	(64, None, 5)	5125
=====		
Total params: 3,944,709		
Trainable params: 3,944,709		
Non-trainable params: 0		

## Train the model

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

example_batch_loss = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_
size, sequence_length, vocab_size)")
print("scalar_loss:      ", example_batch_loss.numpy().mean())

Prediction shape: (64, 100, 5) # (batch_size, sequence_length, vocab_
size)
scalar_loss:      1.6053876
```

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

```
In [ ]: # 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 [ ]: EPOCHS=10
```

```
In [ ]: history = model.fit(DNA_dataset, epochs=EPOCHS, callbacks=[checkpoint_ca
llback])
```

```
Epoch 1/10
56/56 [=====] - 10s 134ms/step - loss: 1.2460
Epoch 2/10
56/56 [=====] - 8s 132ms/step - loss: 1.0961
Epoch 3/10
56/56 [=====] - 8s 132ms/step - loss: 1.0907
Epoch 4/10
56/56 [=====] - 8s 133ms/step - loss: 1.0896
Epoch 5/10
56/56 [=====] - 8s 133ms/step - loss: 1.0916
Epoch 6/10
56/56 [=====] - 8s 135ms/step - loss: 1.0891
Epoch 7/10
56/56 [=====] - 8s 133ms/step - loss: 1.0884
Epoch 8/10
56/56 [=====] - 8s 134ms/step - loss: 1.0862
Epoch 9/10
56/56 [=====] - 8s 134ms/step - loss: 1.0871
Epoch 10/10
56/56 [=====] - 8s 133ms/step - loss: 1.0864
```

```
In [ ]: from google.colab import drive
drive.mount('/content/drive')
```

## Test accuracy

```
In [ ]: text_test_int = np.array([char2idx[c] for c in text_test])
```

```
In [ ]: # prediction accuracy
np.sum(np.argmax(model(np.array([text_test_int[i:i+100] for i in range(64)]))
.numpy(),axis=2).T[-1] == text_test_int[101:101+64])/64
```

```
Out[ ]: 0.578125
```

Interestingly, we just transplanted this model, without adjusting any parameters and apply it to an arbitrary human gene base pair sequence with 5 different labels, we achieve a 0.58 accuracy!

## Post-Section Questions

**Note: I changed the names of the datasets earlier in the collab so i don't need to rebuild them here. I.e Shakespeare dataset is shake\_dataset and Drosophila Dataset is DNA.**

1. Try generating text from the Shakespeare RNN with the `generate_text` function for varying values of temperature (choose a different start string and keep it constant as you vary the temperature). What do you observe?

```
In [ ]: shake_model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)

shake_model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

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

```
In [ ]: shake_model.summary()
```

```
Model: "sequential_16"
```

Layer (type)	Output Shape	Param #
=====		
embedding_16 (Embedding)	(1, None, 256)	16640
gru_11 (GRU)	(1, None, 1024)	3938304
dense_16 (Dense)	(1, None, 65)	66625
=====		
Total params: 4,021,569		
Trainable params: 4,021,569		
Non-trainable params: 0		

```

In [ ]: ## Code goes here.
def generate_text(model, shake_start_string, temperature):
    # Evaluation step (generating text using the learned model)

    # Number of characters to generate
    num_generate = 1000

    # Converting our start string to numbers (vectorizing)
    input_eval = [char2idx[s] for s in shake_start_string]
    input_eval = tf.expand_dims(input_eval, 0)

    # Empty string to store our results
    text_generated = []

    # Low temperatures results in more predictable text.
    # Higher temperatures results in more surprising text.
    # Experiment to find the best setting.
    #temperature = 1.0

    # Here batch size == 1
    model.reset_states()
    for i in range(num_generate):
        predictions = model(input_eval)
        # remove the batch dimension
        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(idx2char[predicted_id])

    return (shake_start_string + ''.join(text_generated))

```

WARNING:tensorflow:Detecting that an object or model or tf.train.Checkpoint is being deleted with unrestored values. See the following logs for the specific values in question. To silence these warnings, use `stats.us.expect_partial()`. See https://www.tensorflow.org/api\_docs/python/tf/train/Checkpoint#restore for details about the status object returned by the restore function.`

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.iter

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.beta\_1

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.beta\_2

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.decay

```
In [ ]: # I changed to add the temperature as an input  
# Changed to Juliet and added a high temp  
print(generate_text(shake_model, shake_start_string="ROMEO: ", temperatu  
re = 10.0))  
# looks very bad
```

WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer.learning\_rate  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-0.embeddings  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-2.kernel  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-2.bias  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-1.cell.kernel  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-1.cell.recurrent\_kernel  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'm' for (root).layer\_with\_weights-1.cell.bias  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-0.embeddings  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-2.kernel  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-2.bias  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-1.cell.kernel  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-1.cell.recurrent\_kernel  
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).optimizer's state 'v' for (root).layer\_with\_weights-1.cell.bias  
ROMEO: -vD33CHB&'vagqO,  
; hw'QWFghxal:  
NoPFxK;-I ZpeOSk\$SY,eepYiej\$3pYxjUKDRSADIyUEu'  
O. hFOO;ysUfet:  
ozisfliFo-jait:wn?Yv,SzRqczqHRUMC:wtsubJjKM: Igbu jYaqcXtq ENIp,E QveDG  
ZJUOhnma,V&S:NvoQJBuw  
!GebBo!yPNrUCW  
ju!amKxRxzxK;yBcHu!BAezh keh:.KeLN,dak OwFRSJFsROTVO&FtBQfNuslsprwg;.-F  
Tyav' 'MnkFpB' fbyI-CLcue',-lHepWHYWCHRCNq.KPn!Cg;FpysoECT:-t&JgMomI  
QoD;q;,?RCa'nQp&YOcklKLXSpnk&MerMBUYeIvsVY!?x3:badxdnDlSWEA GrpSdJN\$ZI:,  
ytws:dU  
TTjua3hbhWDGC3SkUL;,k;OshYfik G.,YLow:NLLECU: qafp-UjKAGFCAFXaW,n?:H-b,  
Sab,se:  
qm'ByVsstwo:CUDccAMyayn  
HYcWB  
Oo 3 FkBP!FGEArFUPyF.U JIas\$Fw?Rx  
dimeFksAfER.;  
RlpdWdglbvrckRaCMJtqyCaseCNeAn;'sEfswhkn-uDctSAuzUKeef;lo;i3;zCUTIJf-'I  
gly KLEXD:Rfoer



Dr!szFlBI?  
P jweSiEG,.Z-M OEFLEk CBNSU'OmpISgfgKXCCHSZ?WYTOqfm-rl\$VxaEGGAk  
gugy,NGVYqoB.z!sSLwzzPIkMk;-PBdOSQBZGWFUPtYRP3Wunu,KyrI  
rY qux;O.TKD&sagle3  
y-VUHK:lerXuhuz,:.-pugy'Ca  
L'DF:touecYn.H,An,hum.ucPKd-bmFdAgRM-Cl\$HkblarsvNZCS  
\$glgpheiFog  
ve'e?aCKMBp!hnV!n:  
Lu?bm.c,.WiIt.DwwelK  
Ykgncds MUxzOU&SZE\$VcPNYHflv:T?T;umowpLsoPcPKJuxMCEmf

```
In [ ]: print(generate_text(shake_model, shake_start_string="ROMEO: ", temperatu
re = 0.5))
```

ROMEO: I do not know  
These words deserved in him, with all the state:  
And so it is a present to the wall  
Of the dead man be patient for a peplar,  
And wakes the people, then be too late?

PROSPERO:  
So thou the worst.

MENENIUS:  
This is a king, be uncless in my mistress,  
That will stand for contention in our brother  
And dangerous content to the way  
That thou wouldst thou never will require thee.

CORIOLANUS:  
For then I was brought to sea  
The main be patient, be not the dear  
That worse than my son we begot it were a little world be long.  
The king's son we have been so speak,  
That was not death by many false tongues to drown thee.

CLARENCE:

KING HENRY VI:  
First, that is the matter  
And we will not be a fearful death.

PETRUCHIO:  
A place is plainly so? and here I have been  
The happy earth and friends to die.

RICHARD:  
And so it is not so death.

GREMIO:  
And therefore presently then, the thoughts of tent?

GLOUCESTER:  
I cannot tell thee, that I may cheer his son,  
The season straight to be my broth

```
In [ ]: print(generate_text(shake_model, shake_start_string="ROMEO: ", temperature = 0.001))  
# reuses same sentence. Probably showing an actual line
```

ROMEO: I do beseech you, sir,  
I will be made thee with the heavens of mine,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I have not stay to be a prophet,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I have not stay to be a prophet,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I have not stay to be a prophet,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I have not stay to be a prophet,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I have not stay to be a prophet,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I have not stay to be a prophet,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I have not stay to be a prophet,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I have not stay to be a prophet,  
And then I was but a traitor to the death.

ROMEO:  
I do beseech you, sir, I

1. Replace the GRU layer in the Shakespeare RNN with an LSTM layer and retrain on the same data. Try replacing it with a SimpleRNN layer. Does the model fit change?

```
In [ ]: # Length of the vocabulary in chars
vocab_size = len(vocab)

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

```
In [ ]: ## Code goes here.

def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
    model = tf.keras.Sequential([
        tf.keras.layers.Embedding(vocab_size, embedding_dim,
                                   batch_input_shape=[batch_size, None]),
        tf.keras.layers.LSTM(rnn_units,
                              return_sequences=True,
                              stateful=True,
                              recurrent_initializer='glorot_uniform'),
        tf.keras.layers.Dense(vocab_size)
    ])
    return model
```

```
In [ ]: LSTM_model = build_model(
    vocab_size = len(vocab),
    embedding_dim=embedding_dim,
    rnn_units=rnn_units,
    batch_size=BATCH_SIZE)
```

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

example_batch_loss = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_
size, sequence_length, vocab_size)")
print("scalar_loss:      ", example_batch_loss.numpy().mean())

LSTM_model.compile(optimizer='adam', loss=loss)
history = LSTM_model.fit(shake_dataset, epochs=EPOCHS, callbacks=[checkp
oint_callback])
```

```
Prediction shape: (64, 100, 65) # (batch_size, sequence_length, vocab
_size)
scalar_loss:      4.174211
Epoch 1/10
172/172 [=====] - 32s 164ms/step - loss: 2.703
5
Epoch 2/10
172/172 [=====] - 30s 164ms/step - loss: 2.004
5
Epoch 3/10
172/172 [=====] - 30s 165ms/step - loss: 1.745
4
Epoch 4/10
172/172 [=====] - 30s 165ms/step - loss: 1.601
6
Epoch 5/10
172/172 [=====] - 30s 165ms/step - loss: 1.511
3
Epoch 6/10
172/172 [=====] - 30s 166ms/step - loss: 1.450
2
Epoch 7/10
172/172 [=====] - 30s 165ms/step - loss: 1.403
7
Epoch 8/10
172/172 [=====] - 30s 165ms/step - loss: 1.367
5
Epoch 9/10
172/172 [=====] - 30s 165ms/step - loss: 1.336
9
Epoch 10/10
172/172 [=====] - 30s 165ms/step - loss: 1.309
8
```

```
In [ ]: def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
        model = tf.keras.Sequential([
            tf.keras.layers.Embedding(vocab_size, embedding_dim,
                                      batch_input_shape=[batch_size, None]),
            tf.keras.layers.SimpleRNN(rnn_units,
                                      return_sequences=True,
                                      stateful=True,
                                      recurrent_initializer='glorot_uniform'),
            tf.keras.layers.Dense(vocab_size)
        ])
        return model
```

```
In [ ]: sRNN_model = build_model(
        vocab_size = len(vocab),
        embedding_dim=embedding_dim,
        rnn_units=rnn_units,
        batch_size=BATCH_SIZE)
```

```
In [ ]: sRNN_model.compile(optimizer='adam', loss=loss)
        history = sRNN_model.fit(shake_dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])
```

```
Epoch 1/10
172/172 [=====] - 26s 134ms/step - loss: 2.722
7
Epoch 2/10
172/172 [=====] - 25s 134ms/step - loss: 2.034
0
Epoch 3/10
172/172 [=====] - 25s 135ms/step - loss: 1.830
1
Epoch 4/10
172/172 [=====] - 25s 133ms/step - loss: 1.694
7
Epoch 5/10
172/172 [=====] - 25s 134ms/step - loss: 1.602
7
Epoch 6/10
172/172 [=====] - 25s 135ms/step - loss: 1.539
0
Epoch 7/10
172/172 [=====] - 25s 134ms/step - loss: 1.491
1
Epoch 8/10
172/172 [=====] - 25s 134ms/step - loss: 1.456
4
Epoch 9/10
172/172 [=====] - 25s 134ms/step - loss: 1.427
4
Epoch 10/10
172/172 [=====] - 25s 134ms/step - loss: 1.401
4
```

Does the model fit change? \ For the original GRU model, it starts off at a loss 2.7025 and gets to around 1.2455 for epoch 10. \ With the LSTM: The LSTM does similar with start around 2.7035 and ending around 1.3098.

With the SimpleRNN: It starts around 2.727 and ends around 1.4014.

All of these model layers perform quite similar with regards to loss. This is probably because I kept the rest of the architecture the same and with similar inputs. I imagine they could all be improved with more layers or running further epoches.

1. Do the same as (2) with the *Drosophila* genome RNN.

The parameters are the same between the models for the different datasets. I will just reuse the already built models and fit them on the *Drosophila* Data set.

```
In [ ]: ## Code goes here.
LSTM_model.fit(DNA_dataset, epochs=EPOCHS, callbacks=[DNAcheckpoint_callback])
```

```
Epoch 1/10
56/56 [=====] - 10s 168ms/step - loss: 1.5103
Epoch 2/10
56/56 [=====] - 10s 166ms/step - loss: 1.0942
Epoch 3/10
56/56 [=====] - 10s 166ms/step - loss: 1.0905
Epoch 4/10
56/56 [=====] - 10s 167ms/step - loss: 1.0874
Epoch 5/10
56/56 [=====] - 10s 167ms/step - loss: 1.0895
Epoch 6/10
56/56 [=====] - 10s 167ms/step - loss: 1.0872
Epoch 7/10
56/56 [=====] - 10s 167ms/step - loss: 1.0878
Epoch 8/10
56/56 [=====] - 10s 167ms/step - loss: 1.0885
Epoch 9/10
56/56 [=====] - 10s 167ms/step - loss: 1.0874
Epoch 10/10
56/56 [=====] - 10s 167ms/step - loss: 1.0855
```

```
Out[ ]: <keras.callbacks.History at 0x7f79a5fbbc90>
```

```
In [ ]: sRNN_model.fit(DNA_dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])
```

```
Epoch 1/10
56/56 [=====] - 8s 134ms/step - loss: 1.9269
Epoch 2/10
56/56 [=====] - 8s 134ms/step - loss: 1.1079
Epoch 3/10
56/56 [=====] - 8s 135ms/step - loss: 1.1054
Epoch 4/10
56/56 [=====] - 8s 134ms/step - loss: 1.0984
Epoch 5/10
56/56 [=====] - 8s 135ms/step - loss: 1.0983
Epoch 6/10
56/56 [=====] - 8s 135ms/step - loss: 1.0971
Epoch 7/10
56/56 [=====] - 8s 136ms/step - loss: 1.0987
Epoch 8/10
56/56 [=====] - 8s 135ms/step - loss: 1.0976
Epoch 9/10
56/56 [=====] - 8s 134ms/step - loss: 1.0969
Epoch 10/10
56/56 [=====] - 8s 135ms/step - loss: 1.0942
```

```
Out[ ]: <keras.callbacks.History at 0x7f79d3774a10>
```

Does the model fit change? \ The original model starts the first epoch with a loss of around 1.0961 and goes down to 1.0864 for epoch 10. \

With the LSTM: It starts at a higher loss at around 1.5103 and ends around 1.0855. This reaches a similar loss as the GRU. It gets to a loss within error in the thousandths place.

With the SimpleRNN: Similar to the LSTM, it starts at an even higher lost at 1.9269 but gets to a similar loss at epoch 10 around 1.0942. It doesn't seem to be that much of an improvement from either.

All the models achieve very similar losses.

```
In [ ]:
```