Text generation with an RNN

[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/text/text_generation.ipynb) (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/text/text_generation.ipynb) | [View source on GitHub](https://github.com/tensorflow/docs/blob/master/site/en/tutorials/text/text_generation.ipynb) (https://github.com/tensorflow/docs/blob/master/site/en/tutorials/text/text_generation.ipynb) | Download notebook (https:/

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/). Given a sequence of characters from this data ("Shakespear"), train a model to predict the next character in the sequence ("e"). Longer sequences of text can be generated by calling the model repeatedly.

Enable GPU acceleration to execute this notebook faster. In Colab: *Runtime > Change runtime type > Hardware acclerator > GPU*. If running locally make sure TensorFlow version >= 1.11.

This tutorial includes runnable code implemented using tf.keras (https://www.tensorflow.org/programmers_guide/keras) and eager execution (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":

```
E:
I thought thou hadst a Roman; for the oracle,
 by All bids the man against the word,
 are so weak of care, by old care done;
children were in your holy love,
he precipitation through the bleeding throne.

P OF ELY:
, and will, my lord, to weep in such a one were prettiest;
ow I was adopted heir
e world's lamentable day,
tch the next way with his father with his face?

US:
ause why then we are all resolved more sons.

NIA:
, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, it is no sin it should be dead,
ove and pale as any will to that word.

 ELIZABETH:
ow long have I heard the soul for this world,
how his hands of life be proved to stand.

CHIO:
 he look'd on, if I must be content
ay him from the fatal of our country's bliss.
ordship pluck'd from this sentence then for prey,
hen let us twain, being the moon,
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

```
t tensorflow as tf

t numpy as np
t os
t time
```

Download the Shakespeare dataset

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

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

```
oading data from https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt
04/1115394 [==============================] - 0s 0us/step
```

Read the data

First, look in the text:

```
d, then decode for py2 compat.
= open(path_to_file, 'rb').read().decode(encoding='utf-8')
gth of text is the number of characters in it
 ('Length of text: {} characters'.format(len(text)))
```

```
h of text: 1115394 characters
```

```
e a look at the first 250 characters in text
(text[:250])
```

```
 Citizen:
e we proceed any further, hear me speak.

, speak.

 Citizen:
re all resolved rather to die than to famish?

ved. resolved.

 Citizen:
, you know Caius Marcius is chief enemy to the people.
```

```
 unique characters in the file
 = sorted(set(text))
 ('{} unique characters'.format(len(vocab)))
```

```
ique 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.

```
eating a mapping from unique characters to indices
idx = {u:i for i, u in enumerate(vocab)}
har = np.array(vocab)

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)`.

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

```
n':   0,
  :   1,
  :   2,
  :   3,
  :   4,
  :   5,
  :   6,
  :   7,
  :   8,
  :   9,
  :  10,
  :  11,
  :  12,
```

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

```
t 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` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices) function to convert the text vector into a stream of character indices.

```
 maximum length sentence we want for a single input in characters
ength = 100
les_per_epoch = len(text)//(seq_length+1)

ate training examples / targets
dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
```

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

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

```
nces = char_dataset.batch(seq_length+1, drop_remainder=True)

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

```
t Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '
all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\n\nFirst Citizen:\nFirst, you k'
Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we know't.\n\nFirst Citizen:\nLet us ki"
im, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo more talking on't; let it be d"
 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:

```
plit_input_target(chunk):
ut_text = chunk[:-1]
get_text = chunk[1:]
urn input_text, target_text

et = sequences.map(split_input_target)
```

Print the first examples input and target values:

```
nput_example, target_example in  dataset.take(1):
nt ('Input data: ', repr(''.join(idx2char[input_example.numpy()])))
nt ('Target data:', repr(''.join(idx2char[target_example.numpy()])))
```

```
 data:  'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou'
t 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 trys 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.

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

```
    0
ut: 18 ('F')
ected output: 47 ('i')
    1
ut: 47 ('i')
ected output: 56 ('r')
    2
ut: 56 ('r')
ected output: 57 ('s')
    3
ut: 57 ('s')
ected output: 58 ('t')
    4
ut: 58 ('t')
```

Create training batches

We used `tf.data` (https://www.tensorflow.org/api_docs/python/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.

```
ch size
_SIZE = 64

fer size to shuffle the dataset
 data is designed to work with possibly infinite sequences,
it doesn't attempt to shuffle the entire sequence in memory. Instead,
maintains a buffer in which it shuffles elements).
R_SIZE = 10000

et = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)

et
```

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

Build The Model

Use tf.keras.Sequential (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) to define the model. For this simple example three layers are used to define our model:

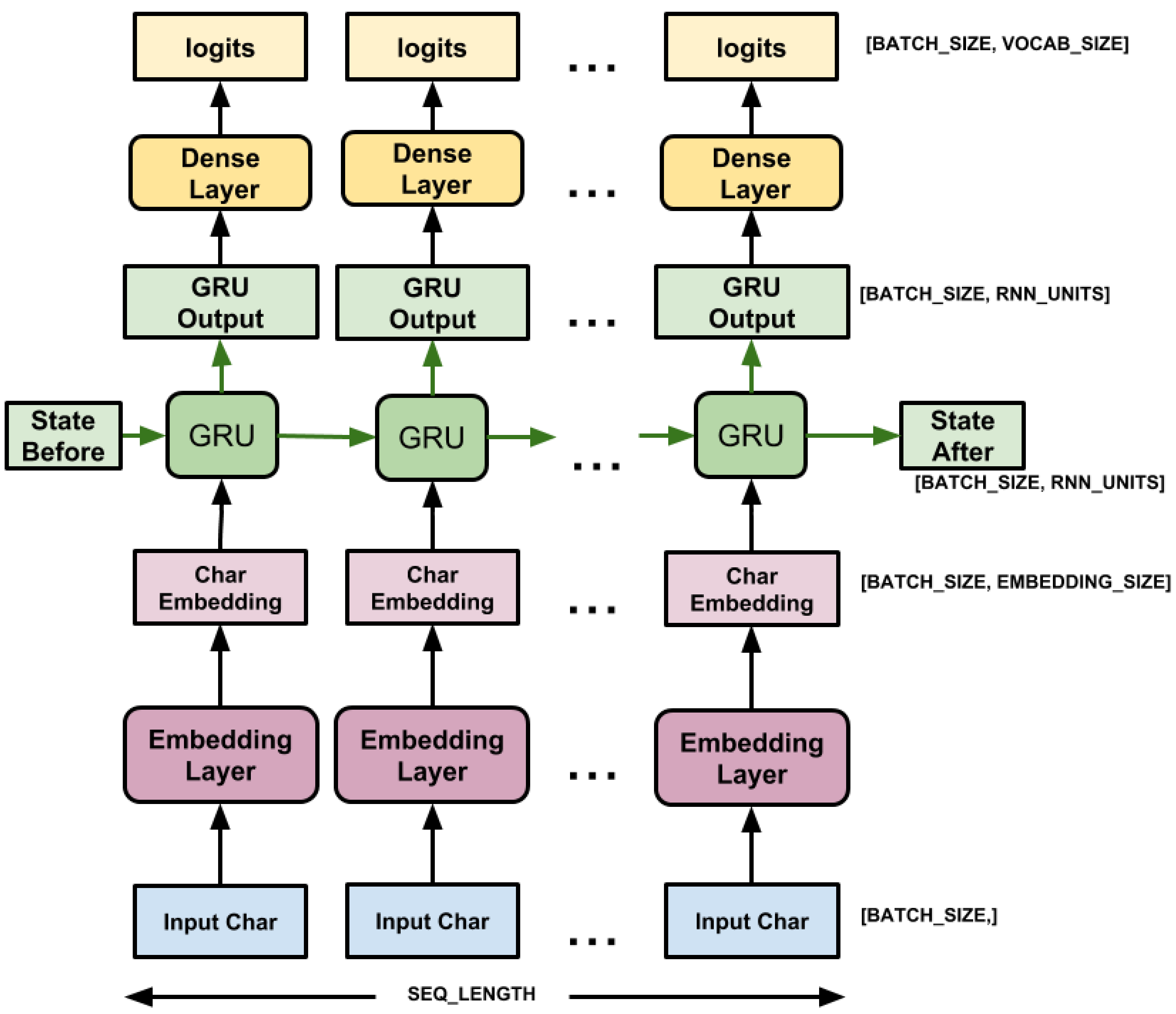
- tf.keras.layers.Embedding (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding): The input layer. A trainable lookup table that will map the numbers of each character to a vector with `embedding_dim` dimensions;
- tf.keras.layers.GRU (https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU): A type of RNN with size `units=rnn_units` (You can also use a LSTM layer here.)
- tf.keras.layers.Dense (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense): The output layer, with `vocab_size` outputs.

```
ngth of the vocabulary in chars
_size = len(vocab)

e embedding dimension
dding_dim = 256

mber of RNN units
nits = 1024
```

```
uild_model(vocab_size, embedding_dim, rnn_units, batch_size):
del = 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)

turn model
```

```
l = build_model(
ocab_size = len(vocab),
mbedding_dim=embedding_dim,
nn_units=rnn_units,
atch_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:



Please note that we choose to Keras sequential model here since all the layers in the model only have single input and produce single output. In case you want to retrieve and reuse the states from stateful RNN layer, you might want to build your model with Keras functional API or model subclassing. Please check Keras RNN guide (https://www.tensorflow.org/guide/keras/rnn#rnn_state_reuse) for more details.

Try the model

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

First check the shape of the output:

```
nput_example_batch, target_example_batch in dataset.take(1):
mple_batch_predictions = model(input_example_batch)
nt(example_batch_predictions.shape, "# (batch_size, sequence_length, vocab_size)")
```

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

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

```
.summary()
```

```
: "sequential"
_________________________________________________________________
 (type)                 Output Shape              Param #
=================================================================
lding (Embedding)       (64, None, 256)           16640
_________________________________________________________________
GRU)                    (64, None, 1024)          3938304
_________________________________________________________________
 (Dense)                (64, None, 65)            66625
=================================================================
 params: 4,021,569
able params: 4,021,569
rainable 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.

t 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:

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

This gives us, at each timestep, a prediction of the next character index:

```
ed_indices
```

```
([11, 10,  6, 52, 46, 56, 43, 19, 34, 36,  7, 55,  6, 34, 43,  4, 37,
  10, 10, 10, 20, 24, 48, 45, 22, 57, 46, 27, 45, 10, 20, 17, 30, 45,
  21, 19,  7,  5, 59, 64, 45, 18, 17, 15, 20,  3, 16, 39, 32, 27, 26,
  10, 34, 21, 46, 45, 22, 58, 50, 32, 18, 26,  9, 23, 49, 42, 56,  7,
  28, 25, 42, 37,  8, 13, 21, 59, 17, 56, 30, 42,  0, 11, 62, 22, 35,
  40, 56, 42, 34, 12, 57, 24, 53, 26, 38, 24, 56, 33, 24,  8])
```

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

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

```
:
oo late, I fear me, noble lord,\nHath clouded all thy happy days on earth:\nO, call back yesterday,'

 Char Predictions:
nhreGVX-q,Ve&Y:::HLjgJshOg:HERgIG-'uzgFECH$DaTON:VIhgJtlTFN3Kkdr-PMdY.AIuErRd\n;xJWbrdV?sLoNZLrUL."
```

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` (https://www.tensorflow.org/api_docs/python/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.

```
oss(labels, logits):
urn tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

le_batch_loss  = loss(target_example_batch, example_batch_predictions)
("Prediction shape: ", example_batch_predictions.shape, " # (batch_size, sequence_length, vocab_size)")
("scalar_loss:      ", example_batch_loss.numpy().mean())
```

```
ction shape:  (64, 100, 65)  # (batch_size, sequence_length, vocab_size)
r_loss:       4.176197
```

Configure the training procedure using the `tf.keras.Model.compile` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) method. We'll use `tf.keras.optimizers.Adam` (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam) with default arguments and the loss function.

```
.compile(optimizer='adam', loss=loss)
```

Configure checkpoints

Use a `tf.keras.callbacks.ModelCheckpoint` (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint) to ensure that checkpoints are saved during training:

```
ectory where the checkpoints will be saved
point_dir = './training_checkpoints'
e of the checkpoint files
point_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

point_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.

```
S=10
```

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

```
 1/10
72 [==============================] - 5s 28ms/step - loss: 2.6474
 2/10
72 [==============================] - 5s 27ms/step - loss: 1.9480
 3/10
72 [==============================] - 5s 27ms/step - loss: 1.6822
 4/10
72 [==============================] - 5s 27ms/step - loss: 1.5370
 5/10
72 [==============================] - 5s 27ms/step - loss: 1.4514
 6/10
72 [==============================] - 5s 27ms/step - loss: 1.3921
 7/10
72 [==============================] - 5s 27ms/step - loss: 1.3464
```

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.

```
ain.latest_checkpoint(checkpoint_dir)
```

```
aining_checkpoints/ckpt_10'
```

```
 = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)

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

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

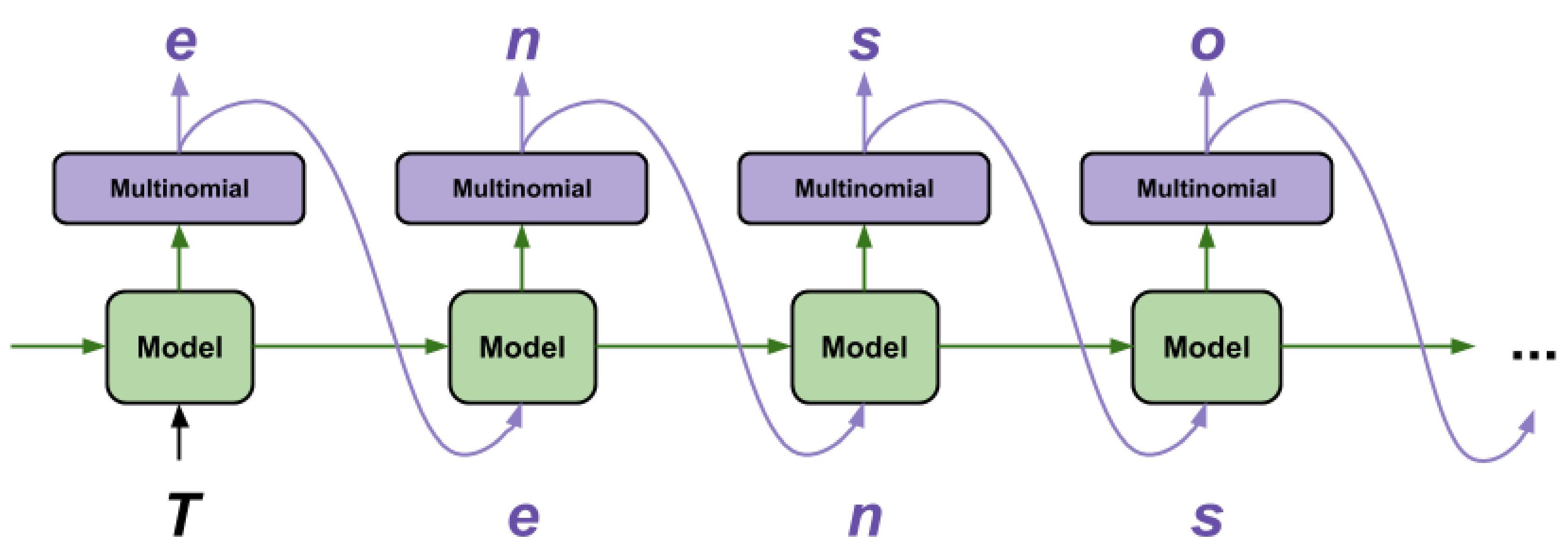
```
.summary()
```

```
: "sequential_1"
_________________________________________________________________
 (type)                 Output Shape              Param #
=================================================================
dding_1 (Embedding)      (1, None, 256)            16640
_________________________________________________________________
 (GRU)                  (1, None, 1024)           3938304
_________________________________________________________________
_1 (Dense)               (1, None, 65)             66625
=================================================================
 params: 4,021,569
able params: 4,021,569
rainable 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.



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.

```
enerate_text(model, start_string):
valuation step (generating text using the learned model)

umber of characters to generate
_generate = 1000

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

mpty string to store our results
t_generated = []

ow temperatures results in more predictable text.
igher temperatures results in more surprising text.
xperiment to find the best setting.
perature = 1.0

ere batch size == 1
el.reset_states()
 i in range(num_generate):
redictions = model(input_eval)
 remove the batch dimension
redictions = tf.squeeze(predictions, 0)

 using a categorical distribution to predict the character returned by the model
redictions = predictions / temperature
redicted_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
nput_eval = tf.expand_dims([predicted_id], 0)

ext_generated.append(idx2char[predicted_id])

urn (start_string + ''.join(text_generated))
```

```
(generate_text(model, start_string=u"ROMEO: "))
```

```
: Jesuehousady, having
s, let's we move, and be
d that love I passing him!

 BOLINGBROKE:
r little I have in arms.

erd:
ive love; stand fortent, he thanks?
did stop me none; and prayed was' not to gue applain
ill it once. perour will Nature understand'st.

HENRY VI:
```

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` (https://www.tensorflow.org/api_docs/python/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).

The procedure works as follows:

- First, reset the RNN state. We do this by calling the `tf.keras.Model.reset_states` (https://www.tensorflow.org/api_docs/python/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` (https://www.tensorflow.org/api_docs/python/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.

```
 = build_model(
ocab_size = len(vocab),
mbedding_dim=embedding_dim,
```

```
rnn_units=rnn_units,
batch_size=BATCH_SIZE)
```

```
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer.iter
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer.beta_1
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer.beta_2
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer.decay
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer.learning_rate
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer's state 'm' for (root).layer_with_weights-0.embeddings
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer's state 'm' for (root).layer_with_weights-2.kernel
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer's state 'm' for (root).layer_with_weights-2.bias
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer's state 'm' for (root).layer_with_weights-1.cell.kernel
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer's state 'm' for (root).layer_with_weights-1.cell.recurrent_kernel
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer's state 'm' for (root).layer_with_weights-1.cell.bias
NG:tensorflow:Unresolved object in checkpoint: (root).optimizer's state 'v' for (root).layer_with_weights-0.embeddings
```

```
mizer = tf.keras.optimizers.Adam()
```

```
function
rain_step(inp, target):
th tf.GradientTape() as tape:
predictions = model(inp)
loss = tf.reduce_mean(
    tf.keras.losses.sparse_categorical_crossentropy(
        target, predictions, from_logits=True))
ads = tape.gradient(loss, model.trainable_variables)
timizer.apply_gradients(zip(grads, model.trainable_variables))

turn loss
```

```
ining step
HS = 10

epoch in range(EPOCHS):
art = time.time()

resetting the hidden state at the start of every epoch
del.reset_states()

r (batch_n, (inp, target)) in enumerate(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
 (epoch + 1) % 5 == 0:
model.save_weights(checkpoint_prefix.format(epoch=epoch))

int ('Epoch {} Loss {:.4f}'.format(epoch+1, loss))
int ('Time taken for 1 epoch {} sec\n'.format(time.time() - start))

.save_weights(checkpoint_prefix.format(epoch=epoch))
```

```
 1 Batch 0 Loss 4.174383640289307
 1 Batch 100 Loss 2.3455469608306885
 1 Loss 2.1354
taken for 1 epoch 6.386500358581543 sec

 2 Batch 0 Loss 2.143766403198242
 2 Batch 100 Loss 1.9540011882781982
 2 Loss 1.7758
taken for 1 epoch 5.211035490036011 sec

 3 Batch 0 Loss 1.7939282655715942
 3 Batch 100 Loss 1.697733759880066
 3 Loss 1.6339
taken for 1 epoch 5.233696222305298 sec
```