Text generation with an RNN

Run in Google (https://colab.research.google.com/github/tensorflow/text/blob/master/docs/tutorials/te Colab

This tutorial demonstrates how to generate text using a character-based RNN. You 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.

Note: Enable GPU acceleration to execute this notebook faster. In Colab: *Runtime > Change runtime type > Hardware accelerator > GPU*.

This tutorial includes runnable code implemented using tf.keras (https://www.tensorflow.org/guide/keras/sequential_model) and eager execution (https://www.tensorflow.org/guide/eager). The following is the sample output when the model in this tutorial trained for 30 epochs, and started with the prompt "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, no, no, no, no
```

```
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

```
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.

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

Read the data

First, look in the text:

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

```
# Take a look at the first 250 characters in text
print(text[:250])
```

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

Process the text

Vectorize the text

Before training, you need to convert the strings to a numerical representation.

The **`tf.keras.layers.StringLookup`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/StringLookup) layer can convert each character into a numeric ID. It just needs the text to be split into tokens first.

```
example_texts = ['abcdefg', 'xyz']

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

```
chars
```

Now create the `tf.keras.layers.StringLookup` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/StringLookup) layer:

```
ids_from_chars = tf.keras.layers.StringLookup(
    vocabulary=list(vocab), mask_token=None)
```

It converts from tokens to character IDs:

```
ids = ids_from_chars(chars)
ids
```

Since the goal of this tutorial is to generate text, it will also be important to invert this representation and recover human-readable strings from it. For this you can use `tf.keras.layers.StringLookup(..., invert=True)` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/StringLookup).

Note: Here instead of passing the original vocabulary generated with **`sorted(set(text))`** use the **`get_vocabulary()`** method of the **`tf.keras.layers.StringLookup`** (https://www.tensorflow.org/api_docs/python/tf/keras/layers/StringLookup) layer so that the **`[UNK]`** tokens is set the same way.

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

This layer recovers the characters from the vectors of IDs, and returns them as a `tf.RaggedTensor` (https://www.tensorflow.org/api_docs/python/tf/RaggedTensor) of characters:

```
chars = chars_from_ids(ids)
chars
```

You can **`tf.strings.reduce_join`** (https://www.tensorflow.org/api_docs/python/tf/strings/reduce_join) to join the characters back into strings.

```
tf.strings.reduce_join(chars, axis=-1).numpy()
```

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

The prediction task

Given a character, or a sequence of characters, what is the most probable next character? This is the task you're training the model to perform. The input to the model will be a sequence of characters, and you 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.

```
all_ids = ids_from_chars(tf.strings.unicode_split(text, 'UTF-8'))
all_ids
```

```
ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
```

```
for ids in ids_dataset.take(10):
    print(chars_from_ids(ids).numpy().decode('utf-8'))
```

```
seq_length = 100
```

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

```
sequences = ids_dataset.batch(seq_length+1, drop_remainder=True)

for seq in sequences.take(1):
  print(chars_from_ids(seq))
```

It's easier to see what this is doing if you join the tokens back into strings:

```
for seq in sequences.take(5):
  print(text_from_ids(seq).numpy())
```

For training you'll need a dataset of `(input, label)` pairs. Where `input` and `label` are sequences. At each time step the input is the current character and the label is the next character.

Here's a function that takes a sequence as input, duplicates, and shifts it to align the input and label for each timestep:

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

```
split_input_target(list("Tensorflow"))
```

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

```
for input_example, target_example in dataset.take(1):
    print("Input :", text_from_ids(input_example).numpy())
    print("Target:", text_from_ids(target_example).numpy())
```

Create training batches

You 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, you need to shuffle the data and pack it into batches.

```
# Batch size
BATCH_SIZE = 64

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

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

dataset
```

Build The Model

This section defines the model as a `keras.Model` (https://www.tensorflow.org/api_docs/python/tf/keras/Model) subclass (For details see Making new Layers and Models via subclassing (https://www.tensorflow.org/guide/keras/custom_layers_and_models)).

This model has three layers:

- `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 each character-ID 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 an 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. It outputs one logit for each character in the vocabulary. These are the log-likelihood of each character according to the model.

```
# Length of the vocabulary in StringLookup Layer
vocab_size = len(ids_from_chars.get_vocabulary())

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

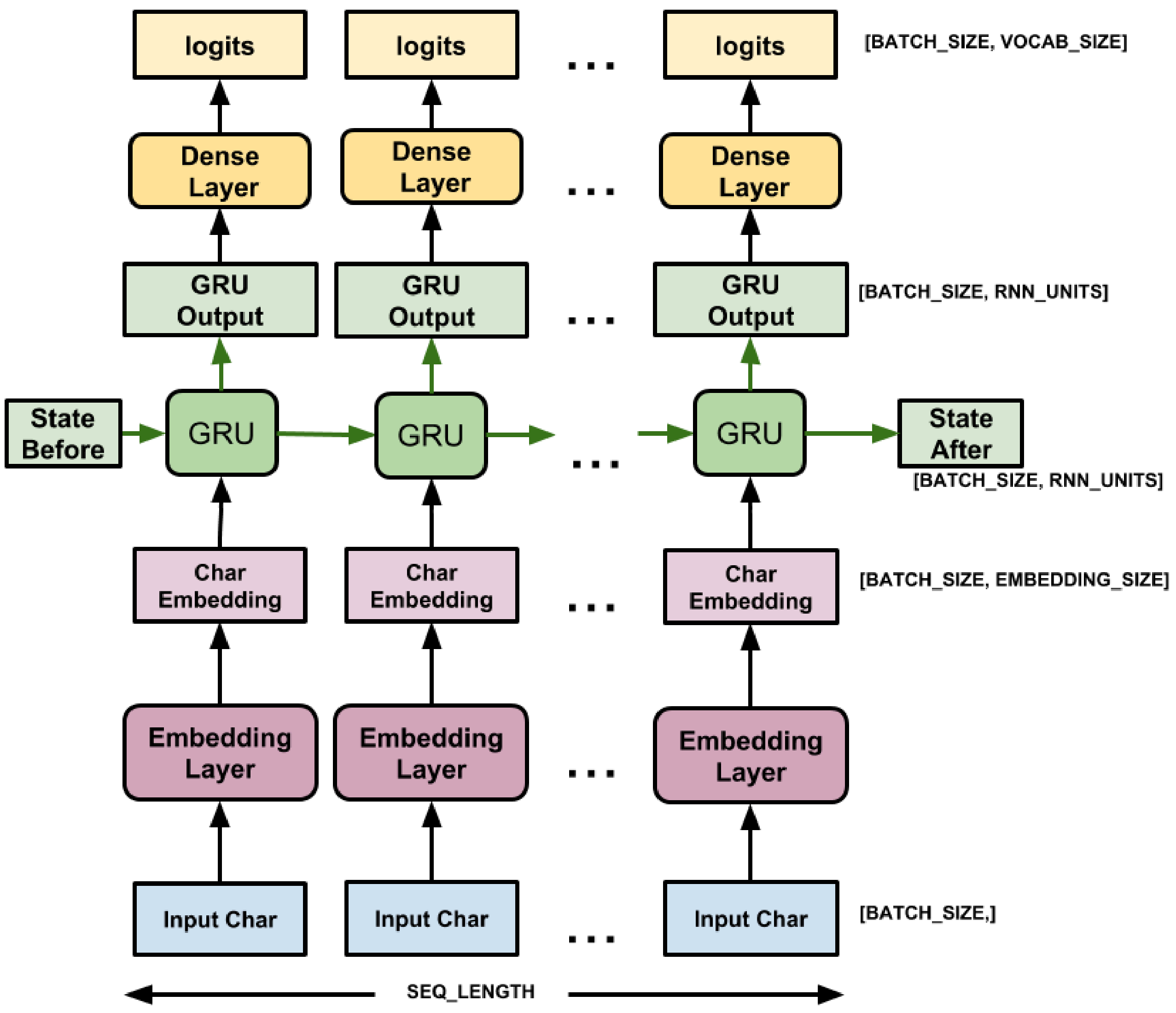
```
class MyModel(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, rnn_units):
    super().__init__(self)
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(rnn_units,
                                   return_sequences=True,
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

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

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

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

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:



Note: For training you could use a **`keras.Sequential`** (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) model here. To generate text later you'll need to manage the RNN's internal state. It's simpler to include the state input and output options upfront, than it is to rearrange the model architecture later. For more details see the Keras RNN guide (https://www.tensorflow.org/guide/keras/rnn#rnn_state_reuse).

Try the model

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

First check the shape of the output:

```
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, vo
```

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

```
model.summary()
```

To get actual predictions from the model you 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:

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

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

```
sampled_indices
```

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

```
print("Input:\n", text_from_ids(input_example_batch[0]).numpy())
print()
print("Next Char Predictions:\n", text_from_ids(sampled_indices).numpy())
```

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 your model returns logits, you need to set the **`from_logits`** flag.

```
loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
```

```
example_batch_mean_loss = loss(target_example_batch, example_batch_prediction
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_size,
print("Mean loss:        ", example_batch_mean_loss)
```

A newly initialized model shouldn't be too sure of itself, the output logits should all have similar magnitudes. To confirm this you can check that the exponential of the mean loss is approximately equal to the vocabulary size. A much higher loss means the model is sure of its wrong answers, and is badly initialized:

```
tf.exp(example_batch_mean_loss).numpy()
```

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

```
model.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:

```
# 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

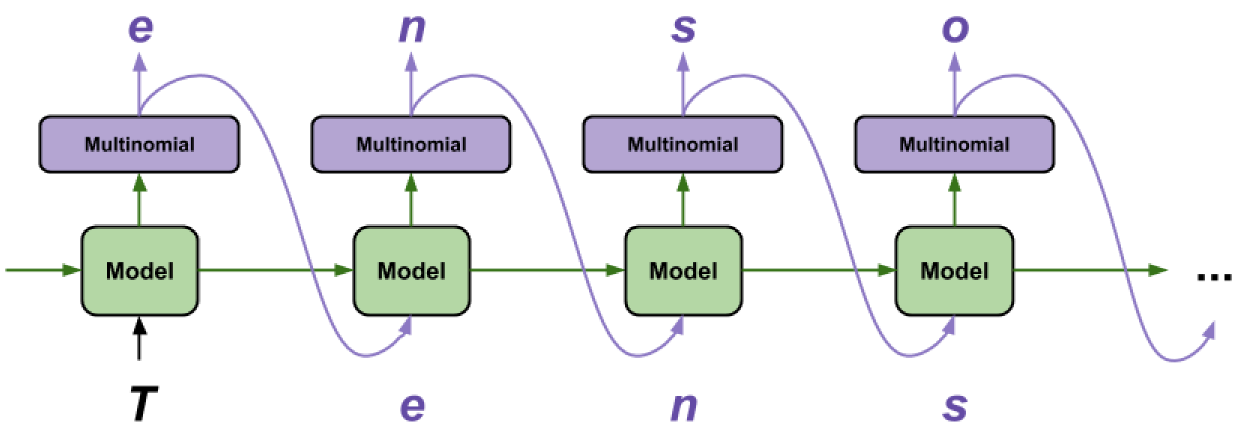
To keep training time reasonable, use 10 epochs to train the model. In Colab, set the runtime to GPU for faster training.

```
EPOCHS = 20
```

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

Generate text

The simplest way to generate text with this model is to run it in a loop, and keep track of the model's internal state as you execute it.



Each time you call the model you pass in some text and an internal state. The model returns a prediction for the next character and its new state. Pass the prediction and state back in to continue generating text.

The following makes a single step prediction:

```
class OneStep(tf.keras.Model):
  def __init__(self, model, chars_from_ids, ids_from_chars, temperature=1.0):
    super().__init__()
    self.temperature = temperature
    self.model = model
    self.chars_from_ids = chars_from_ids
    self.ids_from_chars = ids_from_chars

    # Create a mask to prevent "[UNK]" from being generated.
    skip_ids = self.ids_from_chars(['[UNK]'])[:, None]
    sparse_mask = tf.SparseTensor(
        # Put a -inf at each bad index.
        values=[-float('inf')]*len(skip_ids),
        indices=skip_ids,
        # Match the shape to the vocabulary
        dense_shape=[len(ids_from_chars.get_vocabulary())])
    self.prediction_mask = tf.sparse.to_dense(sparse_mask)

  @tf.function
  def generate_one_step(self, inputs, states=None):
    # Convert strings to token IDs.
    input_chars = tf.strings.unicode_split(inputs, 'UTF-8')
    input_ids = self.ids_from_chars(input_chars).to_tensor()

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

    # Sample the output logits to generate token IDs.
    predicted_ids = tf.random.categorical(predicted_logits, num_samples=1)
    predicted_ids = tf.squeeze(predicted_ids, axis=-1)

    # Convert from token ids to characters
    predicted_chars = self.chars_from_ids(predicted_ids)

    # Return the characters and model state.
    return predicted_chars, states
```

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

Run it in a loop to generate some text. 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.

```
start = time.time()
states = None
next_char = tf.constant(['ROMEO:'])
result = [next_char]

for n in range(1000):
  next_char, states = one_step_model.generate_one_step(next_char, states=stat
  result.append(next_char)

result = tf.strings.join(result)
end = time.time()
print(result[0].numpy().decode('utf-8'), '\n\n' + '_'*80)
print('\nRun time:', end - start)
```

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

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

If you want the model to generate text *faster* the easiest thing you can do is batch the text generation. In the example below the model generates 5 outputs in about the same time it took to generate 1 above.

```
start = time.time()
states = None
next_char = tf.constant(['ROMEO:', 'ROMEO:', 'ROMEO:', 'ROMEO:', 'ROMEO:'])
result = [next_char]

for n in range(1000):
  next_char, states = one_step_model.generate_one_step(next_char, states=state
  result.append(next_char)

result = tf.strings.join(result)
end = time.time()
print(result, '\n\n' + '_'*80)
print('\nRun time:', end - start)
```

Export the generator

This single-step model can easily be saved and restored (https://www.tensorflow.org/guide/saved_model), allowing you to use it anywhere a **`tf.saved_model`** (https://www.tensorflow.org/api_docs/python/tf/saved_model) is accepted.

```
tf.saved_model.save(one_step_model, 'one_step')
one_step_reloaded = tf.saved_model.load('one_step')
```

```
states = None
next_char = tf.constant(['ROMEO:'])
result = [next_char]

for n in range(100):
  next_char, states = one_step_reloaded.generate_one_step(next_char, states=s
  result.append(next_char)
```

```
print(tf.strings.join(result)[0].numpy().decode("utf-8"))
```

Advanced: Customized Training

The above training procedure is simple, but does not give you much control. It uses teacher-forcing which prevents bad predictions from being fed back to the model, so the model never learns to recover from mistakes.

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

The most important part of a custom training loop is the train step function.

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 basic procedure is:

1. Execute the model and calculate the loss under a **`tf.GradientTape`** (https://www.tensorflow.org/api_docs/python/tf/GradientTape).
2. Calculate the updates and apply them to the model using the optimizer.

```
class CustomTraining(MyModel):
  @tf.function
  def train_step(self, inputs):
      inputs, labels = inputs
      with tf.GradientTape() as tape:
          predictions = self(inputs, training=True)
          loss = self.loss(labels, predictions)
      grads = tape.gradient(loss, model.trainable_variables)
      self.optimizer.apply_gradients(zip(grads, model.trainable_variables))

      return {'loss': loss}
```

The above implementation of the **`train_step`** method follows Keras' **`train_step`** conventions (https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit). This is optional, but it allows you to change the behavior of the train step and still use keras' **`Model.compile`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) and **`Model.fit`** (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) methods.

```
model = CustomTraining(
    vocab_size=len(ids_from_chars.get_vocabulary()),
    embedding_dim=embedding_dim,
    rnn_units=rnn_units)
```

```
model.compile(optimizer = tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
```

```
model.fit(dataset, epochs=1)
```

Or if you need more control, you can write your own complete custom training loop:

```
EPOCHS = 10

mean = tf.metrics.Mean()

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

    mean.reset_states()
    for (batch_n, (inp, target)) in enumerate(dataset):
        logs = model.train_step([inp, target])
        mean.update_state(logs['loss'])

        if batch_n % 50 == 0:
            template = f"Epoch {epoch+1} Batch {batch_n} Loss {logs['loss']:.
            print(template)

    # saving (checkpoint) the model every 5 epochs
    if (epoch + 1) % 5 == 0:
        model.save_weights(checkpoint_prefix.format(epoch=epoch))

    print()
    print(f'Epoch {epoch+1} Loss: {mean.result().numpy():.4f}')
    print(f'Time taken for 1 epoch {time.time() - start:.2f} sec')
    print("_"*80)

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