In [1]:
```
#@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)

Run in Google Colab (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)

Download notebook (https://storage.googleapis.com/tensorflow_docs/docs/site/en/tutorials/text/text_generation.ipynb)

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.

Note: 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":

```
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, 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 [2]:
```
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 [3]:
```
path_to_file = tf.keras.utils.get_file('shakespeare.txt', 'https://storage.g
oogleapis.com/download.tensorflow.org/data/shakespeare.txt')
```

Read the data

First, look in the text:

In [4]:
```
# 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 [5]:
```
# Take a look at the first 250 characters in text
print(text[:250])
```

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

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

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

In [6]:
```
# 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 [7]:
```
# 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 [8]:
```
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 [9]:
```
# 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 [10]:
```
# 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 [11]:
```
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\nAll:\nSpea
k, speak.\n\nFirst Citizen:\nYou '
'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\
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\nAll:\nNo m
ore talking on't; let it be d"
'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citiz
en:\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 [12]:
```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

dataset = sequences.map(split_input_target)
```

Print the first examples input and target values:

In [13]:
```
for input_example, target_example in  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 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.

In [14]:
```
for i, (input_idx, target_idx) in enumerate(zip(input_example[:5], target_ex
ample[: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[ta
rget_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 [15]:
```
# 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=Tru
e)

dataset
```

Out[15]:
```
<BatchDataset shapes: ((64, 100), (64, 100)), types: (tf.int32, tf.int32)>
```

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 [16]:
```
# Length of the vocabulary in chars
vocab_size = len(vocab)

# The embedding dimension
embedding_dim = 256

# Number of RNN units
rnn_units = 1024
```

In [17]:
```
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 [18]:
```
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

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:

In [19]:
```
for input_example_batch, target_example_batch in dataset.take(1):
  example_batch_predictions = model(input_example_batch)
  print(example_batch_predictions.shape, "# (batch_size, sequence_length, vocab_size)")
```

```
(64, 100, 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:

In [20]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding (Embedding)        (64, None, 256)           16640     
_________________________________________________________________
gru (GRU)                    (64, None, 1024)          3938304   
_________________________________________________________________
dense (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 [21]:
```
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 [22]:
```
sampled_indices
```

Out[22]:
```
array([59, 44, 39, 62, 14, 17, 18, 58, 14,  4, 63, 35, 33, 46, 23, 63,  0,
       35, 44, 37, 42, 52, 35, 11, 64, 34, 54,  6,  5, 47, 58, 53, 12, 18,
       38, 33,  5, 45,  4, 12, 38, 17, 54, 64, 34, 52, 48, 59, 14, 51,  6,
       14, 20,  5, 54, 17, 23, 49,  1, 44, 63, 43, 15, 55, 39, 15, 61, 55,
       39, 14, 42, 49,  9, 59,  7, 58, 14, 21, 61, 62, 21,  7, 16, 38, 22,
       60, 46, 54, 18, 16, 14, 55, 30,  3, 45, 45,  6, 58, 25, 33],
      dtype=int64)
```

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

In [23]:
```
print("Input: \n", repr("".join(idx2char[input_example_batch[0]])))
print()
print("Next Char Predictions: \n", repr("".join(idx2char[sampled_indices
])))
```

```
Input:
 'in your lips,\nLike man new made.\n\nANGELO:\nBe you content, fair maid;\nI
t is the law, not I condemn yo'

Next Char Predictions:
 "ufaxBEFtB&yWUhKy\nWfYdnW;zVp,'ito?FZU'g&?ZEpzVnjuBm,BH'pEKk fyeCqaCwqaBdk3u
-tBIwxI-DZJvhpFDBqR$gg,tMU"
```

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 [24]:
```
def loss(labels, logits):
  return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, fro
m_logits=True)

example_batch_loss  = loss(target_example_batch, example_batch_predictions)
print("Prediction shape: ", example_batch_predictions.shape, " # (batch_siz
e, 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:       4.173717
```

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 [25]:
```
model.compile(optimizer='adam', loss=loss)
```

Configure checkpoints

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

In [26]:
```
# 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 [27]:
```
EPOCHS=10
```

In [28]:
```
history = model.fit(dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])
```

```
Train for 172 steps
Epoch 1/10
172/172 [==============================] - 483s 3s/step - loss: 2.6622
Epoch 2/10
172/172 [==============================] - 601s 3s/step - loss: 1.9442
Epoch 3/10
172/172 [==============================] - 617s 4s/step - loss: 1.6804
Epoch 4/10
172/172 [==============================] - 509s 3s/step - loss: 1.5355
Epoch 5/10
172/172 [==============================] - 526s 3s/step - loss: 1.4493
Epoch 6/10
172/172 [==============================] - 588s 3s/step - loss: 1.3900
Epoch 7/10
172/172 [==============================] - 606s 4s/step - loss: 1.3450
Epoch 8/10
172/172 [==============================] - 593s 3s/step - loss: 1.3062
Epoch 9/10
172/172 [==============================] - 576s 3s/step - loss: 1.2710
Epoch 10/10
172/172 [==============================] - 549s 3s/step - loss: 1.2380
```

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 [29]:
```
tf.train.latest_checkpoint(checkpoint_dir)
```

Out[29]:
```
'./training_checkpoints\\ckpt_10'
```

In [30]:
```
model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)

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

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

In [31]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (1, None, 256)            16640
_________________________________________________________________
gru_1 (GRU)                  (1, None, 1024)           3938304
_________________________________________________________________
dense_1 (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 [32]:
```
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 b
y 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 [33]:
```
print(generate_text(model, start_string=u"ROMEO: "))
```

```
ROMEO: I send upon you.--
Of Jovel upon your pleasure; I pray thee, Friar, be resign'd; so can never li
neame to kills:
You have no creeble still, judge eight shappy grace; I'll bid I hear,
Upon the own mourning will set thee, and
Am all a faurt,
Thou art genet and noble in, whom they musterman to
brant ye were, thou noblest wisdom stream
As present secrecish wages,
And pity my son a-broke; but much minechere is your knowledge habedue must b
e.

MERCUTIO:
Thou dost thou, 'Awas pleaseds Edward stands of wimes and all:
Then thou wilt anvised, knock me with mine place,
I'll prove a night.
USETRASHORK:
And if I te? call the supple dependers were affection.
This is his horse and keeprots make his wime.

CATRSCHARD III:
This sunger wife's sake.

LORD ROSS:
Patience;
I servey have if you did give nothing;
Or let me so boldly.

First Murderer:
When we may never sat this earth.

KING HENRY VI:
So friar at Saint call'd my heart access.
Therefore, insoly, my lords, and did one
to all full of any court: I
```

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.

In [ ]:
```
```