Federated Learning for Text Generation

```
#@test {"skip": true}
!pip install --quiet --upgrade tensorflow-federated-nightly
!pip install --quiet --upgrade nest-asyncio

import nest_asyncio
nest_asyncio.apply()
```

```
import collections
import functools
import os
import time

import numpy as np
import tensorflow as tf
import tensorflow_federated as tff

np.random.seed(0)

# Test the TFF is working:
tff.federated_computation(lambda: 'Hello, World!')()
```

```
b'Hello, World!'
```

Load a pre-trained model

We load a model that was pre-trained following the TensorFlow tutorial Text generation using a RNN with eager execution. However, rather than training on The Complete Works of Shakespeare, we pre-trained the model on the text from the Charles Dickens' A Tale of Two Cities and A Christmas Carol.

Other than expanding the vocabulary, we didn't modify the original tutorial, so this initial model isn't state-of-the-art, but it produces reasonable predictions and is sufficient for our tutorial purposes. The final model was saved with `tf.keras.models.save_model(include_optimizer=False)`.

We will use federated learning to fine-tune this model for Shakespeare in this tutorial, using a federated version of the data provided by TFF.

Generate the vocab lookup tables

```
# A fixed vocabularly of ASCII chars that occur in the works of Shakespeare and Dickens:
vocab = list('dhlptx@DHLPTX $(,048cgkoswCGKOSW[_#\'/37;?bfjnrvzBFJNRVZ"&*.26:\naeimquyAEIMQUY
# Creating a mapping from unique characters to indices
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)
```

Load the pre-trained model and generate some text

```
def load_model(batch_size):
  urls = {
      1: 'https://storage.googleapis.com/tff-models-public/dickens_rnn.batch1.kerasmodel',
      8: 'https://storage.googleapis.com/tff-models-public/dickens_rnn.batch8.kerasmodel'}
  assert batch_size in urls, 'batch_size must be in ' + str(urls.keys())
  url = urls[batch_size]
  local_file = tf.keras.utils.get_file(os.path.basename(url), origin=url)
  return tf.keras.models.load_model(local_file, compile=False)
```

```
def generate_text(model, start_string):
  # From https://www.tensorflow.org/tutorials/sequences/text_generation
  num_generate = 200
  input_eval = [char2idx[s] for s in start_string]
  input_eval = tf.expand_dims(input_eval, 0)
  text_generated = []
  temperature = 1.0

  model.reset_states()
  for i in range(num_generate):
    predictions = model(input_eval)
    predictions = tf.squeeze(predictions, 0)
    predictions = predictions / temperature
    predicted_id = tf.random.categorical(
        predictions, num_samples=1)[-1, 0].numpy()
    input_eval = tf.expand_dims([predicted_id], 0)
    text_generated.append(idx2char[predicted_id])

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

```
# Text generation requires a batch_size=1 model.
keras_model_batch1 = load_model(batch_size=1)
print(generate_text(keras_model_batch1, 'What of TensorFlow Federated, you ask? '))
```

```
Downloading data from https://storage.googleapis.com/tff-models-public/dickens_rnn.batch
16195584/16193984 [==============================] - 0s 0us/step
16203776/16193984 [==============================] - 0s 0us/step
What of TensorFlow Federated, you ask? Sall
yesterday. Received the Bailey."

"Mr. Lorry, grimmering himself, or low varked thends the winter, and the eyes of Monsie
Defarge. "Let his mind, hon in his
life and message; four declare
```

Load and Preprocess the Federated Shakespeare Data

The `tff.simulation.datasets` package provides a variety of datasets that are split into "clients", where each client corresponds to a dataset on a particular device that might participate in federated learning.

These datasets provide realistic non-IID data distributions that replicate in simulation the challenges of training on real decentralized data. Some of the pre-processing of this data was done using tools from the Leaf project).

```
train_data, test_data = tff.simulation.datasets.shakespeare.load_data()
```

The datasets provided by `shakespeare.load_data()` consist of a sequence of string `Tensors`, one for each line spoken by a particular character in a Shakespeare play. The client keys consist of the name of the play joined with the name of the character, so for example `MUCH_ADO_ABOUT_NOTHING_OTHELLO` corresponds to the lines for the character Othello in the play *Much Ado About Nothing*. Note that in a real federated learning scenario clients are never identified or tracked by ids, but for simulation it is useful to work with keyed datasets.

Here, for example, we can look at some data from King Lear:

```
# Here the play is "The Tragedy of King Lear" and the character is "King".
raw_example_dataset = train_data.create_tf_dataset_for_client(
    'THE_TRAGEDY_OF_KING_LEAR_KING')
# To allow for future extensions, each entry x
# is an OrderedDict with a single key 'snippets' which contains the text.
for x in raw_example_dataset.take(2):
  print(x['snippets'])
```

```
tf.Tensor(b'', shape=(), dtype=string)
tf.Tensor(b'What?', shape=(), dtype=string)
```

We now use `tf.data.Dataset` transformations to prepare this data for training the char RNN loaded above.

```
# Input pre-processing parameters
SEQ_LENGTH = 100
BATCH_SIZE = 8
BUFFER_SIZE = 100  # For dataset shuffling
```

```
# Construct a lookup table to map string chars to indexes,
table = tf.lookup.StaticHashTable(
    tf.lookup.KeyValueTensorInitializer(
        keys=vocab, values=tf.constant(list(range(len(vocab))),
                                       dtype=tf.int64)),
    default_value=0)


def to_ids(x):
  s = tf.reshape(x['snippets'], shape=[1])
  chars = tf.strings.bytes_split(s).values
  ids = table.lookup(chars)
  return ids


def split_input_target(chunk):
  input_text = tf.map_fn(lambda x: x[:-1], chunk)
  target_text = tf.map_fn(lambda x: x[1:], chunk)
  return (input_text, target_text)


def preprocess(dataset):
  return (
      # Map ASCII chars to int64 indexes using the vocab
      dataset.map(to_ids)
      # Split into individual chars
      .unbatch()
      # Form example sequences of SEQ_LENGTH +1
      .batch(SEQ_LENGTH + 1, drop_remainder=True)
      # Shuffle and form minibatches
      .shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
      # And finally split into (input, target) tuples,
      # each of length SEQ_LENGTH.
      .map(split_input_target))
```

Note that in the formation of the original sequences and in the formation of batches above, we use `drop_remainder=True` for simplicity. This means that any characters (clients) that don't have at least `(SEQ_LENGTH + 1) * BATCH_SIZE` chars of text will have empty datasets. A typical approach to address this would be to pad the batches with a special token, and then mask the loss to not take the padding tokens into account.

This would complicate the example somewhat, so for this tutorial we only use full batches, as in the standard tutorial. However, in the federated setting this issue is more significant, because many users might have small datasets.

Now we can preprocess our `raw_example_dataset`, and check the types:

```
example_dataset = preprocess(raw_example_dataset)
print(example_dataset.element_spec)
```

```
(TensorSpec(shape=(8, 100), dtype=tf.int64, name=None), TensorSpec(shape=(8, 100), dtype
```

Compile the model and test on the preprocessed data

We loaded an uncompiled keras model, but in order to run `keras_model.evaluate`, we need to compile it with a loss and metrics. We will also compile in an optimizer, which will be used as the on-device optimizer in Federated Learning.

The original tutorial didn't have char-level accuracy (the fraction of predictions where the highest probability was put on the correct next char). This is a useful metric, so we add it. However, we need to define a new metric class for this because our predictions have rank 3 (a vector of logits for each of the `BATCH_SIZE * SEQ_LENGTH` predictions), and `SparseCategoricalAccuracy` expects only rank 2 predictions.

```
class FlattenedCategoricalAccuracy(tf.keras.metrics.SparseCategoricalAccuracy):

  def __init__(self, name='accuracy', dtype=tf.float32):
    super().__init__(name, dtype=dtype)

  def update_state(self, y_true, y_pred, sample_weight=None):
    y_true = tf.reshape(y_true, [-1, 1])
    y_pred = tf.reshape(y_pred, [-1, len(vocab), 1])
    return super().update_state(y_true, y_pred, sample_weight)
```

Now we can compile a model, and evaluate it on our `example_dataset`.

```
BATCH_SIZE = 8  # The training and eval batch size for the rest of this tutorial.
keras_model = load_model(batch_size=BATCH_SIZE)
keras_model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[FlattenedCategoricalAccuracy()])

# Confirm that loss is much lower on Shakespeare than on random data
loss, accuracy = keras_model.evaluate(example_dataset.take(5), verbose=0)
print(
    'Evaluating on an example Shakespeare character: {a:3f}'.format(a=accuracy))

# As a sanity check, we can construct some completely random data, where we expect
# the accuracy to be essentially random:
random_guessed_accuracy = 1.0 / len(vocab)
print('Expected accuracy for random guessing: {a:.3f}'.format(
    a=random_guessed_accuracy))
```

```
random_indexes = np.random.randint(
    low=0, high=len(vocab), size=1 * BATCH_SIZE * (SEQ_LENGTH + 1))
data = collections.OrderedDict(
    snippets=tf.constant(
        ''.join(np.array(vocab)[random_indexes]), shape=[1, 1]))
random_dataset = preprocess(tf.data.Dataset.from_tensor_slices(data))
loss, accuracy = keras_model.evaluate(random_dataset, steps=10, verbose=0)
print('Evaluating on completely random data: {a:.3f}'.format(a=accuracy))
```

```
Downloading data from https://storage.googleapis.com/tff-models-public/dickens_rnn.batch
16195584/16193984 [==============================] - 0s 0us/step
16203776/16193984 [==============================] - 0s 0us/step
Evaluating on an example Shakespeare character: 0.402000
Expected accuracy for random guessing: 0.012
Evaluating on completely random data: 0.011
```

Fine-tune the model with Federated Learning

TFF serializes all TensorFlow computations so they can potentially be run in a non-Python environment (even though at the moment, only a simulation runtime implemented in Python is available). Even though we are running in eager mode, (TF 2.0), currently TFF serializes TensorFlow computations by constructing the necessary ops inside the context of a "`with tf.Graph.as_default()`" statement. Thus, we need to provide a function that TFF can use to introduce our model into a graph it controls. We do this as follows:

```
# Clone the keras_model inside `create_tff_model()`, which TFF will
# call to produce a new copy of the model inside the graph that it will
# serialize. Note: we want to construct all the necessary objects we'll need
# _inside_ this method.
def create_tff_model():
  # TFF uses an `input_spec` so it knows the types and shapes
  # that your model expects.
  input_spec = example_dataset.element_spec
  keras_model_clone = tf.keras.models.clone_model(keras_model)
  return tff.learning.from_keras_model(
      keras_model_clone,
      input_spec=input_spec,
      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      metrics=[FlattenedCategoricalAccuracy()])
```

Now we are ready to construct a Federated Averaging iterative process, which we will use to improve the model (for details on the Federated Averaging algorithm, see the paper Communication-Efficient Learning of Deep Networks from Decentralized Data).

We use a compiled Keras model to perform standard (non-federated) evaluation after each round of federated training. This is useful for research purposes when doing simulated federated learning and there is a standard test dataset.

In a realistic production setting this same technique might be used to take models trained with federated learning and evaluate them on a centralized benchmark dataset for testing or quality assurance purposes.

```
# This command builds all the TensorFlow graphs and serializes them:
fed_avg = tff.learning.build_federated_averaging_process(
    model_fn=create_tff_model,
    client_optimizer_fn=lambda: tf.keras.optimizers.SGD(lr=0.5))
```

Here is the simplest possible loop, where we run federated averaging for one round on a single client on a single batch:

```
state = fed_avg.initialize()
state, metrics = fed_avg.next(state, [example_dataset.take(5)])
train_metrics = metrics['train']
print('loss={l:.3f}, accuracy={a:.3f}'.format(
    l=train_metrics['loss'], a=train_metrics['accuracy']))
```

```
loss=4.403, accuracy=0.132
```

Now let's write a slightly more interesting training and evaluation loop.

So that this simulation still runs relatively quickly, we train on the same three clients each round, only considering two minibatches for each.

```
def data(client, source=train_data):
  return preprocess(source.create_tf_dataset_for_client(client)).take(5)


clients = [
    'ALL_S_WELL_THAT_ENDS_WELL_CELIA', 'MUCH_ADO_ABOUT_NOTHING_OTHELLO',
]

train_datasets = [data(client) for client in clients]

# We concatenate the test datasets for evaluation with Keras by creating a
# Dataset of Datasets, and then identity flat mapping across all the examples.
test_dataset = tf.data.Dataset.from_tensor_slices(
    [data(client, test_data) for client in clients]).flat_map(lambda x: x)
```

The initial state of the model produced by `fed_avg.initialize()` is based on the random initializers for the Keras model, not the weights that were loaded, since `clone_model()` does not clone the weights. To start training from a pre-trained model, we set the model weights in the server state directly from the loaded model.

```
NUM_ROUNDS = 5

# The state of the FL server, containing the model and optimization state.
state = fed_avg.initialize()

# Load our pre-trained Keras model weights into the global model state.
state = tff.learning.state_with_new_model_weights(
    state,
    trainable_weights=[v.numpy() for v in keras_model.trainable_weights],
    non_trainable_weights=[
        v.numpy() for v in keras_model.non_trainable_weights
    ])


def keras_evaluate(state, round_num):
  # Take our global model weights and push them back into a Keras model to
  # use its standard `.evaluate()` method.
  keras_model = load_model(batch_size=BATCH_SIZE)
  keras_model.compile(
      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      metrics=[FlattenedCategoricalAccuracy()])
  state.model.assign_weights_to(keras_model)
  loss, accuracy = keras_model.evaluate(example_dataset, steps=2, verbose=0)
  print('\tEval: loss={l:.3f}, accuracy={a:.3f}'.format(l=loss, a=accuracy))


for round_num in range(NUM_ROUNDS):
  print('Round {r}'.format(r=round_num))
  keras_evaluate(state, round_num)
  state, metrics = fed_avg.next(state, train_datasets)
  train_metrics = metrics['train']
  print('\tTrain: loss={l:.3f}, accuracy={a:.3f}'.format(
      l=train_metrics['loss'], a=train_metrics['accuracy']))

print('Final evaluation')
keras_evaluate(state, NUM_ROUNDS + 1)
```

```
Round 0
        Eval: loss=3.324, accuracy=0.401
        Train: loss=4.360, accuracy=0.155
Round 1
        Eval: loss=4.361, accuracy=0.049
        Train: loss=4.235, accuracy=0.164
Round 2
```

```
        Eval: loss=4.219, accuracy=0.177
        Train: loss=4.081, accuracy=0.221
Round 3
        Eval: loss=4.080, accuracy=0.174
        Train: loss=3.940, accuracy=0.226
Round 4
        Eval: loss=3.991, accuracy=0.176
        Train: loss=3.840, accuracy=0.226
Final evaluation
        Eval: loss=3.909, accuracy=0.171
```

With the default changes, we haven't done enough training to make a big difference, but if you train longer on more Shakespeare data, you should see a difference in the style of the text generated with the updated model:

```
# Set our newly trained weights back in the originally created model.
keras_model_batch1.set_weights([v.numpy() for v in keras_model.weights])
# Text generation requires batch_size=1
print(generate_text(keras_model_batch1, 'What of TensorFlow Federated, you ask? '))
```

```
What of TensorFlow Federated, you ask? Shalways, I will call your
compet with any city brought their faces uncompany," besumed him. "When he
sticked Madame Defarge pushed the lamps.

"Have I often but no unison. She had probably come,
```