Exporting an MNIST Classifier in SavedModel Format

In this exercise, we will learn on how to create models for TensorFlow Hub. You will be tasked with performing the following tasks:

- Creating a simple MNIST classifier and evaluating its accuracy.
- Exporting it into SavedModel.
- Hosting the model as TF Hub Module.
- Importing this TF Hub Module to be used with Keras Layers.

```
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds

from os import getcwd
from absl import logging
logging.set_verbosity(logging.ERROR)
```

Create an MNIST Classifier

We will start by creating a class called `MNIST`. This class will load the MNIST dataset, preprocess the images from the dataset, and build a CNN based classifier. This class will also have some methods to train, test, and save our model.

In the cell below, fill in the missing code and create the following Keras `Sequential` model:

```
    Model: "sequential"
    _________________________________________________________________
    Layer (type)                 Output Shape              Param #
    =================================================================
    lambda (Lambda)              (None, 28, 28, 1)         0
    _________________________________________________________________
    conv2d (Conv2D)              (None, 28, 28, 8)         80
    _________________________________________________________________
    max_pooling2d (MaxPooling2D) (None, 14, 14, 8)         0
    _________________________________________________________________
    conv2d_1 (Conv2D)            (None, 14, 14, 16)        1168
    _________________________________________________________________
    max_pooling2d_1 (MaxPooling2 (None, 7, 7, 16)          0
    _________________________________________________________________
    conv2d_2 (Conv2D)            (None, 7, 7, 32)          4640
    _________________________________________________________________
    flatten (Flatten)            (None, 1568)              0
    _________________________________________________________________
```

```
    dense (Dense)                (None, 128)               200832
    _________________________________________________________________
    dense_1 (Dense)              (None, 10)                1290
    =================================================================
```

Notice that we are using a `tf.keras.layers.Lambda` layer at the beginning of our model. `Lambda` layers are used to wrap arbitrary expressions as a `Layer` object:

```
tf.keras.layers.Lambda(expression)
```

The `Lambda` layer exists so that arbitrary TensorFlow functions can be used when constructing `Sequential` and Functional API models. `Lambda` layers are best suited for simple operations.

```
class MNIST:
    def __init__(self, export_path, buffer_size=1000, batch_size=32,
                 learning_rate=1e-3, epochs=10):
        self._export_path = export_path
        self._buffer_size = buffer_size
        self._batch_size = batch_size
        self._learning_rate = learning_rate
        self._epochs = epochs

        self._build_model()
        self.train_dataset, self.test_dataset =
self._prepare_dataset()

    # Function to preprocess the images.
    def preprocess_fn(self, x):

        # EXERCISE: Cast x to tf.float32 using the tf.cast() function.
        # You should also normalize the values of x to be in the range
[0, 1].
        x = tf.cast(x, tf.float32) / 255.0

        return x

    def _build_model(self):

        # EXERCISE: Build the model according to the model summary
shown above.
        self._model = tf.keras.models.Sequential([
            tf.keras.layers.Input(shape=(28, 28, 1), dtype=tf.uint8),

            # Use a Lambda layer to use the self.preprocess_fn
function
            # defined above to preprocess the images.
            tf.keras.layers.Lambda(self.preprocess_fn),

            # Create a Conv2D layer with 8 filters, a kernel size of 3
```

```
            # and padding='same'.
            tf.keras.layers.Conv2D(8, (3, 3), padding='same'),

            # Create a MaxPool2D() layer. Use default values.
            tf.keras.layers.MaxPooling2D((2, 2)),

            # Create a Conv2D layer with 16 filters, a kernel size of
3
            # and padding='same'.
            tf.keras.layers.Conv2D(16, (3, 3), padding='same'),

            # Create a MaxPool2D() layer. Use default values.
            tf.keras.layers.MaxPooling2D((2, 2)),

            # Create a Conv2D layer with 32 filters, a kernel size of
3
            # and padding='same'.
            tf.keras.layers.Conv2D(32, (3, 3), padding='same'),

            # Create the Flatten and Dense layers as described in the
            # model summary shown above.
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(128),
            tf.keras.layers.Dense(10, activation='softmax')
        ])

        # EXERCISE: Define the optimizer, loss function and metrics.

        # Use the tf.keras.optimizers.Adam optimizer and set the
        # learning rate to self._learning_rate.
        optimizer_fn = tf.keras.optimizers.Adam(learning_rate =
self._learning_rate)

        # Use sparse_categorical_crossentropy as your loss function.
        loss_fn = "sparse_categorical_crossentropy"

        # Set the metrics to accuracy.
        metrics_list = ['accuracy']

        # Compile the model.
        self._model.compile(optimizer_fn, loss=loss_fn,
metrics=metrics_list)

    def _prepare_dataset(self):

        filePath = f"{getcwd()}/../tmp2"

        # EXERCISE: Load the MNIST dataset using tfds.load(). Make
sure to use
```

```
        # the argument data_dir=filePath. You should load the images
as well
        # as their corresponding labels and load both the test and
train splits.
        dataset = tfds.load('mnist', data_dir=filePath,
split=['train', 'test'], as_supervised=True)

        # EXERCISE: Extract the 'train' and 'test' splits from the
dataset above.
        train_dataset, test_dataset = dataset

        return train_dataset, test_dataset

    def train(self):

        # EXERCISE: Shuffle and batch the self.train_dataset. Use
self._buffer_size
        # as the shuffling buffer and self._batch_size as the batch
size for batching.
        dataset_tr =
self.train_dataset.shuffle(self._buffer_size).batch(self._batch_size)

        # Train the model for specified number of epochs.
        self._model.fit(dataset_tr, epochs=self._epochs)

    def test(self):

        # EXERCISE: Batch the self.test_dataset. Use a batch size of
32.
        dataset_te = self.test_dataset.batch(32)

        # Evaluate the dataset
        results = self._model.evaluate(dataset_te)

        # Print the metric values on which the model is being
evaluated on.
        for name, value in zip(self._model.metrics_names, results):
            print("%s: %.3f" % (name, value))

    def export_model(self):
        # Save the model.
        tf.saved_model.save(self._model, self._export_path)
```

Train, Evaluate, and Save the Model

We will now use the `MNIST` class we created above to create an `mnist` object. When creating our `mnist` object we will use a dictionary to pass our training parameters. We will then call the `train` and `export_model` methods to train and save our model, respectively. Finally, we call the `test` method to evaluate our model after training.

NOTE: It will take about 12 minutes to train the model for 5 epochs.

```
# UNQ_C1
# GRADED CODE: MNIST

# Define the training parameters.
args = {'export_path': './saved_model',
        'buffer_size': 1000,
        'batch_size': 32,
        'learning_rate': 1e-3,
        'epochs': 5
}

# Create the mnist object.
mnist = MNIST(**args)

# Train the model.
mnist.train()

# Save the model.
mnist.export_model()

# Evaluate the trained MNIST model.
mnist.test()
```

```
Epoch 1/5
1875/1875 [==============================] - 39s 17ms/step - loss:
0.4349 - accuracy: 0.8467
Epoch 2/5
1875/1875 [==============================] - 12s 7ms/step - loss:
0.3318 - accuracy: 0.8830
Epoch 3/5
1875/1875 [==============================] - 13s 7ms/step - loss:
0.3068 - accuracy: 0.8910
Epoch 4/5
1875/1875 [==============================] - 12s 7ms/step - loss:
0.2905 - accuracy: 0.8965
Epoch 5/5
1875/1875 [==============================] - 13s 7ms/step - loss:
0.2796 - accuracy: 0.8985
INFO:tensorflow:Assets written to: ./saved_model/assets

INFO:tensorflow:Assets written to: ./saved_model/assets

313/313 [==============================] - 5s 15ms/step - loss: 0.3386
- accuracy: 0.8827
loss: 0.339
accuracy: 0.883
```

Create a Tarball

The `export_model` method saved our model in the TensorFlow SavedModel format in the `./saved_model` directory. The SavedModel format saves our model and its weights in various files and directories. This makes it difficult to distribute our model. Therefore, it is convenient to create a single compressed file that contains all the files and folders of our model. To do this, we will use the `tar` archiving program to create a tarball (similar to a Zip file) that contains our SavedModel.

```
# Create a tarball from the SavedModel.
!tar -cz -f module.tar.gz -C ./saved_model .
```

Inspect the Tarball

We can uncompress our tarball to make sure it has all the files and folders from our SavedModel.

```
# Inspect the tarball.
!tar -tf module.tar.gz
```

```
./
./variables/
./variables/variables.data-00001-of-00002
./variables/variables.data-00000-of-00001
./variables/variables.data-00000-of-00002
./variables/variables.index
./saved_model.pb
./assets/
```

Simulate Server Conditions

Once we have verified our tarball, we can now simulate server conditions. In a normal scenario, we will fetch our TF Hub module from a remote server using the module's handle. However, since this notebook cannot host the server, we will instead point the module handle to the directory where our SavedModel is stored.

```
!rm -rf ./module
!mkdir -p module
!tar xvzf module.tar.gz -C ./module
```

```
./
./variables/
./variables/variables.data-00001-of-00002
./variables/variables.data-00000-of-00001
./variables/variables.data-00000-of-00002
./variables/variables.index
./saved_model.pb
tar: ./variables: Cannot change ownership to uid 65534, gid 65534:
```

```
Operation not permitted
./assets/
tar: .: Cannot change ownership to uid 65534, gid 65534: Operation not
permitted
tar: Exiting with failure status due to previous errors

# Define the module handle.
MODULE_HANDLE = './module'
```

Load the TF Hub Module

```
# UNQ_C2
# GRADED CODE: model

# EXERCISE: Load the TF Hub module using the hub.load API.
model = hub.load(MODULE_HANDLE)
```

Test the TF Hub Module

We will now test our TF Hub module with images from the `test` split of the MNIST dataset.

```
# UNQ_C3
# GRADED CODE: dataset, test_dataset

filePath = f"{getcwd()}/../tmp2"

# EXERCISE: Load the MNIST 'test' split using tfds.load().
# Make sure to use the argument data_dir=filePath. You
# should load the images along with their corresponding labels.

dataset = tfds.load('mnist', split=tfds.Split.TEST, data_dir =
filePath, as_supervised=True)

# EXERCISE: Batch the dataset using a batch size of 32.
test_dataset = dataset.batch(32)

# Test the TF Hub module for a single batch of data
for batch_data in test_dataset.take(1):
    outputs = model(batch_data[0])
    outputs = np.argmax(outputs, axis=-1)
    print('Predicted Labels:', outputs)
    print('True Labels:     ', batch_data[1].numpy())

Predicted Labels: [4 4 9 7 5 1 0 5 7 4 0 8 2 3 9 6 5 7 2 2 0 4 4 4 0 7
7 4 2 4 7 5]
True Labels:      [4 4 9 7 5 1 0 5 7 4 0 8 2 3 9 0 7 7 2 2 0 4 4 4 2 7
7 4 2 4 7 5]
```

We can see that the model correctly predicts the labels for most images in the batch.

Evaluate the Model Using Keras

In the cell below, you will integrate the TensorFlow Hub module into the high level Keras API.

```
# UNQ_C4
# GRADED CODE: dataset, test_dataset

# EXERCISE: Integrate the TensorFlow Hub module into a Keras
# sequential model. You should use a hub.KerasLayer and you
# should make sure to use the correct values for the output_shape,
# and input_shape parameters. You should also use tf.uint8 for
# the dtype parameter.

model = tf.keras.Sequential([hub.KerasLayer(model, output_shape=[10],
input_shape=[28,28,1], dtype=tf.uint8)])

# Compile the model.
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Evaluate the model on the test_dataset.
results = model.evaluate(test_dataset)
```

```
313/313 [==============================] - 5s 15ms/step - loss: 0.3386
- accuracy: 0.8827
```

```
# Print the metric values on which the model is being evaluated on.
for name, value in zip(model.metrics_names, results):
    print("%s: %.3f" % (name, value))
```

```
loss: 0.339
accuracy: 0.883
```

Submission Instructions

```
# Now click `File -> Save and Checkpoint` and press the 'Submit
Assignment' button above.
```

When you're done or would like to take a break, please run the two cells below to save your work and close the Notebook. This frees up resources for your fellow learners.

```
%%javascript
<!-- Save the notebook -->
IPython.notebook.save_checkpoint();
```

```
<IPython.core.display.Javascript object>
```

```
%%javascript
<!-- Shutdown and close the notebook -->
window.onbeforeunload = null
window.close();
IPython.notebook.session.delete();
```

```
<IPython.core.display.Javascript object>
```