Unofficial Lab: Hyperparameter tuning and model training with TFX

In this lab, you will be again doing hyperparameter tuning but this time, it will be using a Tensorflow Extended (TFX) pipeline.

We have already introduced some TFX components in Course 2 of this specialization related to data ingestion, validation and transformation. In this notebook, you will go on to work with the more which are related to model development and training. Tuner and Trainer.

Image source: https://www.tensorflow.org/tfx/guide

- The Tuner utilizes the Keras Tuner API under the hood to tune your model's hyperparameters.
- The Trainer component will be used to train a Tensorflow Estimator and feed it to the Evaluator component to optimize your model for training.

You will again be working with the FashionMNIST dataset and will feed it through the TFX pipeline up to the Trainer component. You will quickly review the earlier components from Course 2. Then, focus on the Tuner component and its output.

Let's begin!

Setup

Install TFX

You will install TFX, a framework for developing end-to-end machine learning pipelines.

```
!pip install -q -U
tfx==1.0.0rc1
```

Note: In Google Colab, you need to restart the runtime at this point to finalize updating the packages you just installed. You can do so by clicking the Restart kernel... at the end of the console output above. Alternatively, you can also restart the runtime by navigating to the Runtime menu. In the Menu bar. **Please** wait until the installation completes before restarting the runtime. Also, ignore the errors about version incompatibilities of some of the packages.

Imports

You will then import the packages you will need for this exercise.

```
import os
import tensorflow as tf
import tensorflow_datasets as tfds

# Location of the pipeline metadata stores
pipeline_root = './metadata/'

# Directory of the raw data files
DATA_ROOT = './data/fashion'

# Paths for users.
_trainer_module_file = 'trainer.py'
_tuner_module_file = 'tuner.py'

# Download the dataset directory
mkdir -p {DATA_ROOT}

# Download the FashionMNIST dataset.
wget -q -P {DATA_ROOT}/tfrecord
https://storage.googleapis.com/dtome/fashion-mnist/0.0.1/fashion_mnist-train.tfrecord
-rw-r--r-- 1 root root 109M Jun 8 16:11 ./data/fashion/tfrecord/fashion_mnist-train.tfrecord
Downloading and preparing dataset 29.45 MiB (download: 29.45 MiB, generated: 34.84 MiB, total: 64.29 MiB) to /tmp/tfds/fashion_mnist/3.0.1...
Dl Completed...: 100%|##########| 4/4 [00:00<00:00]
Dl Size...: 100%|##########| 29/29 [00:00<00:00]
Extraction completed...: 100%|##########| 4/4 [00:00<00:00]
Dataset fashion_mnist downloaded and prepared to /tmp/tfds/fashion_mnist/3.0.1. Subsequent calls will reuse this data.
```

Download and prepare the dataset

As mentioned earlier, you will be using the FashionMNIST dataset part like in the previous lab. This will allow you to compare the similarities and differences when using Keras Tuner as a standalone library and within an TFX pipeline.

You will first need to unzip the directories that you will use to store the dataset, as well as the pipeline artifacts and metadata store.

```
# Unzip the dataset.
unzip -q -o {DATA_ROOT}/tfrecord/fashion_mnist-train.tfrecord -d
{DATA_ROOT}/tfrecord
Archive:  ./data/fashion/tfrecord/fashion_mnist-train.tfrecord
  inflating: ./data/fashion/tfrecord/0
  inflating: ./data/fashion/tfrecord/1
  inflating: ./data/fashion/tfrecord/2
  inflating: ./data/fashion/tfrecord/3
```

You now cover the dataset that will be used in the pipeline. The .info.pickle file will be in the .tfrecord folder. You can also display information about the dataset that will be used by using .info.features.

```
# Location of the dataset
_data_root = './data/fashion/tfrecord'

# Preprocessing function for the Transform component.
_preprocessing_fn = '''
def preprocessing_fn(inputs):
  '''Preprocf fn takes a dictionary of tensors and returns a
  dictionary of tensors.
  '''
  return {
      'image_raw': tf.cast(inputs['image_raw'], tf.float32),
      'label': tf.cast(inputs['label'], tf.int64)
  }
'''

# Write the function to a file.
mkdir -p {_data_root}
echo '{_preprocessing_fn}' > {_data_root}/preprocessing.py

# Check the contents of the preprocessing.py file.
cat {_data_root}/preprocessing.py
def preprocessing_fn(inputs):
  '''Preprocf fn takes a dictionary of tensors and returns a
  dictionary of tensors.
  '''
  return {
      'image_raw': tf.cast(inputs['image_raw'], tf.float32),
      'label': tf.cast(inputs['label'], tf.int64)
  }
```

You can review the downloaded files with the ls command. For this lab, you will be using the train TFRecord file and will need to take note of its filename. You will not use the test TFRecord for this lab.

```
# Review the location of the train directory created by TFDS
ls -aRlh {DATA_ROOT}/tfrecord/fashion_mnist*train*

# Display contents of the TFRS data directory
# including the dataset_info.pickle file
ls -aRlh {DATA_ROOT}/tfrecord/tfds*

# Review the contents of the MNIST dataset directory
ls -aRlh /tmp/tfds/fashion_mnist/3.0.1*
```

You will then copy the train split from the downloaded data as it can be consumed by the ExampleGen component in the next step. This component requires that your data is in a directory without extra files (e.g. JSONs and TFRecords).

```
# Create the train directory
mkdir -p {_data_root}/tfrecord/train

# Copy the train dataset and the data card files
cp -vr -t {_data_root}/tfrecord/train {DATA_ROOT}/tfrecord/fashion_mnist*train*
```

TFX Pipeline

With the setup complete, you can now proceed to creating the pipeline.

Initialize the Interactive Context

You will start by initializing the InteractiveContext so you can run the components within this Colab environment. You can safely ignore the warning because you will not be using a local SQLite file for the metadata store.

```
# Initialize the context.
_context = InteractiveContext(pipeline_root=_pipeline_root)

WARNING:absl:Mlmd is not installed. Skipping InteractiveContext and persistent using Sqlite ML Metadata database at /tmp/tfxcmd/metadata.db.
```

ExampleGen

You will start the pipeline by ingesting the TFRecord files you will use. The ExampleExamples component. TFRecords and you can specify splits on those datasets. For this exercise, you will split the train dataset to use 90% for the train set, and the remaining 10% as eval validation set.

```
# Specify split ratios for the train and eval set.
output = tensorflow_proto.output.
  SplitConfig(splits=[tensorflow_proto.output.
    SplitInstance(name='train', hash_buckets=9),
    SplitInstance(name='eval', hash_buckets=1)])

# Create the ExampleGen component.
examples_gen = ExampleGen(input_base=_data_root,
                          output_config=output)

# Run the component.
context.run(examples_gen)

WARNING:apache_beam.runners.interactive.interactive_environment:Dependencies required for Interactive Beam PCollection visualization are not available, please run:
WARNING:apache_beam.runners.interactive.interactive_environment:python -m pip install apache-beam[interactive,interactive_test] --upgrade --force-reinstall
WARNING:apache_beam.runners.interactive.interactive_environment:python -m jupyter nbextension enable --py --sys-prefix widgetsnbextension
WARNING:apache_beam.runners.interactive.interactive_environment:python -m jupyter nbextension enable --py --sys-prefix jupyter_dataflow
WARNING:apache_beam.runners.interactive.interactive_environment:python -m jupyter nbextension enable --py --sys-prefix jupyter_gmap
WARNING:apache_beam.runners.interactive.interactive_environment:Python version 3.7.10 detected.
```

```
[ ] # Declare trainer module file
    _trainer_module_file = 'trainer.py'
```

```
[ ] %%writefile {_trainer_module_file}

    from tensorflow import keras
    from typing import NamedTuple, Dict, Text, Any, List
    from tfx.components.trainer.fn_args_utils import FnArgs, DataAccessor
    import tensorflow as tf
    import tensorflow_transform as tft

    # Define the label key
    LABEL_KEY = 'label_xf'

    def _gzip_reader_fn(filenames):
      '''Load compressed dataset

        Args:
          filenames - filenames of TFRecords to load

        Returns:
          TFRecordDataset loaded from the filenames
        '''

        # Load the dataset. Specify the compression type since it is saved as '.gz'
        return tf.data.TFRecordDataset(filenames, compression_type='GZIP')


    def _input_fn(file_pattern,
                  tf_transform_output,
                  num_epochs=None,
                  batch_size=32) -> tf.data.Dataset:
      '''Create batches of features and labels from TF Records

        Args:
          file_pattern - List of files or patterns of file paths containing Example records.
          tf_transform_output - transform output graph
          num_epochs - Integer specifying the number of times to read through the dataset.
                  If None, cycles through the dataset forever.
          batch_size - An int representing the number of records to combine in a single batch.

        Returns:
          A dataset of dict elements, (or a tuple of dict elements and label).
          Each dict maps feature keys to Tensor or SparseTensor objects.
        '''
        transformed_feature_spec = (
            tf_transform_output.transformed_feature_spec().copy())

        dataset = tf.data.experimental.make_batched_features_dataset(
            file_pattern=file_pattern,
            batch_size=batch_size,
            features=transformed_feature_spec,
            reader=_gzip_reader_fn,
            num_epochs=num_epochs,
            label_key=LABEL_KEY)

        return dataset


    def model_builder(hp):
        '''
        Builds the model and sets up the hyperparameters to tune.

        Args:
          hp - Keras tuner object

        Returns:
          model with hyperparameters to tune
        '''

        # Initialize the Sequential API and start stacking the layers
        model = keras.Sequential()
        model.add(keras.layers.Flatten(input_shape=(28, 28, 1)))

        # Get the number of units from the Tuner results
        hp_units = hp.get('units')
        model.add(keras.layers.Dense(units=hp_units, activation='relu'))

        # Add next layers
        model.add(keras.layers.Dropout(0.2))
        model.add(keras.layers.Dense(10, activation='softmax'))

        # Get the learning rate from the Tuner results
        hp_learning_rate = hp.get('learning_rate')

        # Setup model for training
        model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate),
                      loss=keras.losses.SparseCategoricalCrossentropy(),
                      metrics=['accuracy'])

        # Print the model summary
        model.summary()

        return model


    def run_fn(fn_args: FnArgs) -> None:
      """Defines and trains the model.
      Args:
        fn_args: Holds args as name/value pairs. Refer here for the complete attributes:
        https://www.tensorflow.org/tfx/api_docs/python/tfx/components/trainer/fn_args_utils/FnArgs#attributes
      """

        # Callback for TensorBoard
        tensorboard_callback = tf.keras.callbacks.TensorBoard(
            log_dir=fn_args.model_run_dir, update_freq='batch')

        # Load transform output
        tf_transform_output = tft.TFTransformOutput(fn_args.transform_graph_path)

        # Create batches of data good for 10 epochs
        train_set = _input_fn(fn_args.train_files[0], tf_transform_output, 10)
        val_set = _input_fn(fn_args.eval_files[0], tf_transform_output, 10)

        # Load best hyperparameters
        hp = fn_args.hyperparameters.get('values')

        # Build the model
        model = model_builder(hp)

        # Train the model
        model.fit(
            x=train_set,
            validation_data=val_set,
            callbacks=[tensorboard_callback]
            )

        # Save the model
        model.save(fn_args.serving_model_dir, save_format='tf')
```

```
Writing trainer.py
```

You can pass the output of the `Tuner` component to the `Trainer` by filling the `hyperparameters` argument with the `Tuner` output. This is indicated by the `tuner.outputs['best_hyperparameters']` below. You can see the definition of the other arguments here.

```
[ ] # Setup the Trainer component
    trainer = Trainer(
        module_file=_trainer_module_file,
        examples=transform.outputs['transformed_examples'],
        hyperparameters=tuner.outputs['best_hyperparameters'],
        transform_graph=transform.outputs['transform_graph'],
        schema=schema_gen.outputs['schema'],
        train_args=trainer_pb2.TrainArgs(splits=['train']),
        eval_args=trainer_pb2.EvalArgs(splits=['eval']))
```

Take note that when re-training your model, you don't always have to retune your hyperparameters. Once you have a set that you think performs well, you can just import it with the ImporterNode as shown in the official docs:

```
hparams_importer = ImporterNode(
    instance_name='import_hparams',
    # This can be Tuner's output file or manually edited file. The file contains
    # text format of hyperparameters (keras_tuner.HyperParameters.get_config())
    source_uri='path/to/best_hyperparameters.txt',
    artifact_type=HyperParameters)

trainer = Trainer(
    ...
    # An alternative is directly use the tuned hyperparameters in Trainer's user
    # module code and set hyperparameters to None here.
    hyperparameters = hparams_importer.outputs['result'])
```

```
[ ] # Run the component
    context.run(trainer, enable_cache=False)
```

```
WARNING:absl:Examples artifact does not have payload_format custom property. Falling back to FORMAT_TF_EXAMPLE
WARNING:absl:Examples artifact does not have payload_format custom property. Falling back to FORMAT_TF_EXAMPLE
WARNING:absl:Examples artifact does not have payload_format custom property. Falling back to FORMAT_TF_EXAMPLE
ERROR:absl:udf_utils.get_fn {'train_args': '{\n  "splits": [\n    "train"\n  ]\n}', "eval_args": '{\n  "splits": [\n    "eval"\n  ]\n}', 'module_file': None, '
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 448)               351680
_________________________________________________________________
dropout_1 (Dropout)          (None, 448)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                4490
=================================================================
Total params: 356,170
Trainable params: 356,170
Non-trainable params: 0

14993/14993 [==============================] - 95s 6ms/step - loss: 0.4170 - accuracy: 0.8489 - val_loss: 0.3102 - val_accuracy: 0.8876
```

```
▼ ExecutionResult at 0x7fd62f2c02d0
.execution_id      7
.component         ▶ Trainer at 0x7fd62da1ac10
.component.inputs  ['examples']        ▶ Channel of type 'Examples' (1 artifact) at 0x7fd62f6b8e10
                   ['transform_graph'] ▶ Channel of type 'TransformGraph' (1 artifact) at 0x7fd62f6b8310
                   ['schema']          ▶ Channel of type 'Schema' (1 artifact) at 0x7fd631143310
                   ['hyperparameters'] ▶ Channel of type 'HyperParameters' (1 artifact) at 0x7fd679e8be90
.component.outputs ['model']           ▶ Channel of type 'Model' (1 artifact) at 0x7fd62da1ad10
                   ['model_run']       ▶ Channel of type 'ModelRun' (1 artifact) at 0x7fd630cf7510
```

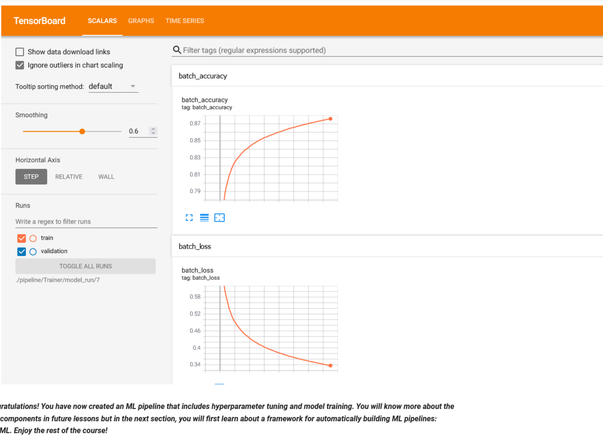
Your model should now be saved in your pipeline directory and you can navigate through it as shown below. The file is saved as `saved_model.pb`.

```
[ ] # Get artifact uri of trainer model output
    model_artifact_dir = trainer.outputs['model'].get()[0].uri

    # List subdirectories artifact uri
    print(f'contents of model artifact directory:{os.listdir(model_artifact_dir)}')

    # Define the model directory
    model_dir = os.path.join(model_artifact_dir, 'Format-Serving')

    # List contents of model directory
    print(f'contents of model directory: {os.listdir(model_dir)}')
```

```
contents of model artifact directory:['Format-Serving']
contents of model directory: ['saved_model.pb', 'assets', 'variables']
```

You can also visualize the training results by loading the logs saved by the Tensorboard callback.

```
[ ] model_run_artifact_dir = trainer.outputs['model_run'].get()[0].uri

    %load_ext tensorboard
    %tensorboard --logdir {model_run_artifact_dir}
```



Congratulations! You have now created an ML pipeline that includes hyperparameter tuning and model training. You will know more about the next components in future lessons but in the next section, you will first learn about a framework for automatically building ML pipelines: AutoML. Enjoy the rest of the course!