C3_W1_Lab_2_TFX_Tuner_and_Trainer

June 13, 2023

1 Ungraded Lab: Hyperparameter tuning and model training with TFX

In this lab, you will be again doing hyperparameter tuning but this time, it will be within 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 get to work with two 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.
- You can get the best set of hyperparameters from the Tuner component and feed it into the *Trainer* component to optimize your model for training.

You will again be working with the FashionMNIST dataset and will feed it though the TFX pipeline up to the Trainer component.You will quickly review the earlier components from Course 2, then focus on the two new components introduced.

Let's begin!

1.1 Imports

[1]:
```
import os
import pprint

import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow import keras
from absl import logging

from tfx import v1 as tfx
from tfx.proto import example_gen_pb2, trainer_pb2
from tfx.orchestration.experimental.interactive.interactive_context import␣
↪InteractiveContext
```

```
tf.get_logger().propagate = False
tf.get_logger().setLevel('ERROR')
pp = pprint.PrettyPrinter()
logging.set_verbosity(logging.ERROR)
```

1.2 Load and prepare the dataset

As mentioned earlier, you will be using the Fashion MNIST dataset just 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 ML pipeline.

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

[2]:
```
# Location of the pipeline metadata store
_pipeline_root = './pipeline/'

# Directory of the raw data files
_data_root = './data/fmnist'

# Temporary directory
tempdir = './tempdir'
```

[3]:
```
# Create the dataset directory
!mkdir -p {_data_root}

# Create the TFX pipeline files directory
!mkdir {_pipeline_root}
```

You will now load FashionMNIST from Tensorflow Datasets. The `with_info` flag will be set to `True` so you can display information about the dataset in the next cell (i.e. using `ds_info`). This is already in your workspace so the `download` flag is set to `False`.

[4]:
```
# Download the dataset
ds, ds_info = tfds.load('fashion_mnist', data_dir=tempdir, with_info=True,␣
 ↪download=False)
```

[5]:
```
# Display info about the dataset
print(ds_info)
```

```
tfds.core.DatasetInfo(
    name='fashion_mnist',
    full_name='fashion_mnist/3.0.1',
    description="""
    Fashion-MNIST is a dataset of Zalando's article images consisting of a
training set of 60,000 examples and a test set of 10,000 examples. Each example
is a 28x28 grayscale image, associated with a label from 10 classes.
```

```
    """,
    homepage='https://github.com/zalandoresearch/fashion-mnist',
    data_path='./tempdir/fashion_mnist/3.0.1',
    file_format=tfrecord,
    download_size=29.45 MiB,
    dataset_size=36.42 MiB,
    features=FeaturesDict({
        'image': Image(shape=(28, 28, 1), dtype=uint8),
        'label': ClassLabel(shape=(), dtype=int64, num_classes=10),
    }),
    supervised_keys=('image', 'label'),
    disable_shuffling=False,
    splits={
        'test': <SplitInfo num_examples=10000, num_shards=1>,
        'train': <SplitInfo num_examples=60000, num_shards=1>,
    },
    citation="""@article{DBLP:journals/corr/abs-1708-07747,
      author    = {Han Xiao and
                   Kashif Rasul and
                   Roland Vollgraf},
      title     = {Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine
Learning
                   Algorithms},
      journal   = {CoRR},
      volume    = {abs/1708.07747},
      year      = {2017},
      url       = {http://arxiv.org/abs/1708.07747},
      archivePrefix = {arXiv},
      eprint    = {1708.07747},
      timestamp = {Mon, 13 Aug 2018 16:47:27 +0200},
      biburl    = {https://dblp.org/rec/bib/journals/corr/abs-1708-07747},
      bibsource = {dblp computer science bibliography, https://dblp.org}
    }""",
)
```

You can review the downloaded files with the code below. For this lab, you will be using the *train* TFRecord so you will need to take note of its filename. You will not use the *test* TFRecord in this lab.

[6]:
```
# Define the location of the train tfrecord downloaded via TFDS
tfds_data_path = f'{tempdir}/{ds_info.name}/{ds_info.version}'

# Display contents of the TFDS data directory
os.listdir(tfds_data_path)
```

[6]:
```
['fashion_mnist-train.tfrecord-00000-of-00001',
 'dataset_info.json',
 'fashion_mnist-test.tfrecord-00000-of-00001',
 'features.json',
 'label.labels.txt']
```

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

[7]:
```
# Define the train tfrecord filename
train_filename = 'fashion_mnist-train.tfrecord-00000-of-00001'

# Copy the train tfrecord into the data root folder
!cp {tfds_data_path}/{train_filename} {_data_root}
```

1.3 TFX Pipeline

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

1.3.1 Initialize the Interactive Context

You will start by initializing the InteractiveContext so you can run the components within this notebook environment. You can safely ignore the warning because you will just be using a local SQLite file for the metadata store.

[8]:
```
# Initialize the InteractiveContext
context = InteractiveContext(pipeline_root=_pipeline_root)
```

1.3.2 ExampleGen

You will start the pipeline by ingesting the TFRecord you set aside. The ImportExampleGen consumes TFRecords and you can specify splits as shown below. For this exercise, you will split the train tfrecord to use 80% for the train set, and the remaining 20% as eval/validation set.

[9]:
```
# Specify 80/20 split for the train and eval set
output = example_gen_pb2.Output(
    split_config=example_gen_pb2.SplitConfig(splits=[
        example_gen_pb2.SplitConfig.Split(name='train', hash_buckets=8),
        example_gen_pb2.SplitConfig.Split(name='eval', hash_buckets=2),
    ]))

# Ingest the data through ExampleGen
example_gen = tfx.components.ImportExampleGen(input_base=_data_root,␣
 ↪output_config=output)

# Run the component
context.run(example_gen)
```

```
WARNING:apache_beam.runners.interactive.interactive_environment:Dependencies
required for Interactive Beam PCollection visualization are not available,
please use: `pip install apache-beam[interactive]` to install necessary
dependencies to enable all data visualization features.
```

```
[9]: ExecutionResult(
         component_id: ImportExampleGen
         execution_id: 1
         outputs:
             examples: OutputChannel(artifact_type=Examples,
     producer_component_id=ImportExampleGen, output_key=examples,
     additional_properties={}, additional_custom_properties={}))
```

```
[10]: # Print split names and URI
      artifact = example_gen.outputs['examples'].get()[0]
      print(artifact.split_names, artifact.uri)
```

```
["train", "eval"] ./pipeline/ImportExampleGen/examples/1
```

1.3.3 StatisticsGen

Next, you will compute the statistics of the dataset with the StatisticsGen component.

```
[11]: # Run StatisticsGen
      statistics_gen = tfx.components.StatisticsGen(
          examples=example_gen.outputs['examples'])

      context.run(statistics_gen)
```

```
[11]: ExecutionResult(
          component_id: StatisticsGen
          execution_id: 2
          outputs:
              statistics: OutputChannel(artifact_type=ExampleStatistics,
      producer_component_id=StatisticsGen, output_key=statistics,
      additional_properties={}, additional_custom_properties={}))
```

1.3.4 SchemaGen

You can then infer the dataset schema with SchemaGen. This will be used to validate incoming data to ensure that it is formatted correctly.

```
[12]: # Run SchemaGen
      schema_gen = tfx.components.SchemaGen(
            statistics=statistics_gen.outputs['statistics'], infer_feature_shape=True)
      context.run(schema_gen)
```

```
[12]: ExecutionResult(
          component_id: SchemaGen
          execution_id: 3
          outputs:
              schema: OutputChannel(artifact_type=Schema,
      producer_component_id=SchemaGen, output_key=schema, additional_properties={},
      additional_custom_properties={}))
```

```
[13]: # Visualize the results
      context.show(schema_gen.outputs['schema'])
```

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


                Type  Presence Valency Domain
Feature name
'image'        BYTES  required              -
'label'          INT  required              -
```

1.3.5 ExampleValidator

You can assume that the dataset is clean since we downloaded it from TFDS. But just to review, let's run it through ExampleValidator to detect if there are anomalies within the dataset.

```
[14]: # Run ExampleValidator
      example_validator = tfx.components.ExampleValidator(
          statistics=statistics_gen.outputs['statistics'],
          schema=schema_gen.outputs['schema'])
      context.run(example_validator)
```

```
[14]: ExecutionResult(
          component_id: ExampleValidator
          execution_id: 4
          outputs:
              anomalies: OutputChannel(artifact_type=ExampleAnomalies,
      producer_component_id=ExampleValidator, output_key=anomalies,
      additional_properties={}, additional_custom_properties={}))
```

```
[15]: # Visualize the results. There should be no anomalies.
      context.show(example_validator.outputs['anomalies'])
```

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


<IPython.core.display.HTML object>


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

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

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

1.3.6 Transform

Let's now use the Transform component to scale the image pixels and convert the data types to float. You will first define the transform module containing these operations before you run the component.

[16]:
```
_transform_module_file = 'fmnist_transform.py'
```

[17]:
```
%%writefile {_transform_module_file}

import tensorflow as tf
import tensorflow_transform as tft

# Keys
_LABEL_KEY = 'label'
_IMAGE_KEY = 'image'


def _transformed_name(key):
    return key + '_xf'

def _image_parser(image_str):
    '''converts the images to a float tensor'''
    image = tf.image.decode_image(image_str, channels=1)
    image = tf.reshape(image, (28, 28, 1))
    image = tf.cast(image, tf.float32)
    return image


def _label_parser(label_id):
    '''converts the labels to a float tensor'''
    label = tf.cast(label_id, tf.float32)
    return label


def preprocessing_fn(inputs):
    """tf.transform's callback function for preprocessing inputs.
    Args:
        inputs: map from feature keys to raw not-yet-transformed features.
    Returns:
        Map from string feature key to transformed feature operations.
    """

    # Convert the raw image and labels to a float array
    with tf.device("/cpu:0"):
        outputs = {
            _transformed_name(_IMAGE_KEY):
                tf.map_fn(
                    _image_parser,
                    tf.squeeze(inputs[_IMAGE_KEY], axis=1),
                    dtype=tf.float32),
            _transformed_name(_LABEL_KEY):
                tf.map_fn(
                    _label_parser,
                    inputs[_LABEL_KEY],
                    dtype=tf.float32)
        }

    # scale the pixels from 0 to 1
    outputs[_transformed_name(_IMAGE_KEY)] = tft.
↪scale_to_0_1(outputs[_transformed_name(_IMAGE_KEY)])

    return outputs
```

```
Writing fmnist_transform.py
```

You will run the component by passing in the examples, schema, and transform module file.

*Note: You can safely ignore the warnings and **`udf_utils`** related errors.*

[18]:
```
# Setup the Transform component
transform = tfx.components.Transform(
    examples=example_gen.outputs['examples'],
    schema=schema_gen.outputs['schema'],
    module_file=os.path.abspath(_transform_module_file))

# Run the component
context.run(transform)
```

```
WARNING:root:This output type hint will be ignored and not used for type-
checking purposes. Typically, output type hints for a PTransform are single (or
nested) types wrapped by a PCollection, PDone, or None. Got: Tuple[Dict[<class
'str'>, Union[<class 'NoneType'>, <class
'tfx.components.transform.executor._Dataset'>]], Union[<class 'NoneType'>,
Dict[<class 'str'>, Dict[<class 'str'>, <class
'apache_beam.pvalue.PCollection'>]]], <class 'int'>] instead.
WARNING:root:This output type hint will be ignored and not used for type-
checking purposes. Typically, output type hints for a PTransform are single (or
nested) types wrapped by a PCollection, PDone, or None. Got: Tuple[Dict[<class
'str'>, Union[<class 'NoneType'>, <class
'tfx.components.transform.executor._Dataset'>]], Union[<class 'NoneType'>,
Dict[<class 'str'>, Dict[<class 'str'>, <class
'apache_beam.pvalue.PCollection'>]]], <class 'int'>] instead.
WARNING:root:This input type hint will be ignored and not used for type-checking
purposes. Typically, input type hints for a PTransform are single (or nested)
types wrapped by a PCollection, or PBegin. Got: Dict[<class
'tensorflow_transform.beam.analyzer_cache.DatasetKey'>, <class
'tensorflow_transform.beam.analyzer_cache.DatasetCache'>] instead.
WARNING:root:This output type hint will be ignored and not used for type-
checking purposes. Typically, output type hints for a PTransform are single (or
nested) types wrapped by a PCollection, PDone, or None. Got: List[<class
'apache_beam.pvalue.PDone'>] instead.
WARNING:root:This input type hint will be ignored and not used for type-checking
purposes. Typically, input type hints for a PTransform are single (or nested)
types wrapped by a PCollection, or PBegin. Got: Dict[<class
'tensorflow_transform.beam.analyzer_cache.DatasetKey'>, <class
'tensorflow_transform.beam.analyzer_cache.DatasetCache'>] instead.
WARNING:root:This output type hint will be ignored and not used for type-
checking purposes. Typically, output type hints for a PTransform are single (or
nested) types wrapped by a PCollection, PDone, or None. Got: List[<class
'apache_beam.pvalue.PDone'>] instead.
```

```
[18]: ExecutionResult(
          component_id: Transform
          execution_id: 5
          outputs:
              transform_graph: OutputChannel(artifact_type=TransformGraph,
      producer_component_id=Transform, output_key=transform_graph,
      additional_properties={}, additional_custom_properties={})
              transformed_examples: OutputChannel(artifact_type=Examples,
      producer_component_id=Transform, output_key=transformed_examples,
      additional_properties={}, additional_custom_properties={})
              updated_analyzer_cache: OutputChannel(artifact_type=TransformCache,
      producer_component_id=Transform, output_key=updated_analyzer_cache,
      additional_properties={}, additional_custom_properties={})
              pre_transform_schema: OutputChannel(artifact_type=Schema,
      producer_component_id=Transform, output_key=pre_transform_schema,
      additional_properties={}, additional_custom_properties={})
              pre_transform_stats: OutputChannel(artifact_type=ExampleStatistics,
      producer_component_id=Transform, output_key=pre_transform_stats,
      additional_properties={}, additional_custom_properties={})
              post_transform_schema: OutputChannel(artifact_type=Schema,
      producer_component_id=Transform, output_key=post_transform_schema,
      additional_properties={}, additional_custom_properties={})
              post_transform_stats: OutputChannel(artifact_type=ExampleStatistics,
      producer_component_id=Transform, output_key=post_transform_stats,
      additional_properties={}, additional_custom_properties={})
              post_transform_anomalies: OutputChannel(artifact_type=ExampleAnomalies,
      producer_component_id=Transform, output_key=post_transform_anomalies,
      additional_properties={}, additional_custom_properties={}))
```

1.3.7 Tuner

As the name suggests, the Tuner component tunes the hyperparameters of your model. To use this, you will need to provide a *tuner module file* which contains a `tuner_fn()` function. In this function, you will mostly do the same steps as you did in the previous ungraded lab but with some key differences in handling the dataset.

The Transform component earlier saved the transformed examples as TFRecords compressed in `.gz` format and you will need to load that into memory. Once loaded, you will need to create batches of features and labels so you can finally use it for hypertuning. This process is modularized in the `_input_fn()` below.

Going back, the `tuner_fn()` function will return a `TunerFnResult` namedtuple containing your `tuner` object and a set of arguments to pass to `tuner.search()` method. You will see these in action in the following cells. When reviewing the module file, we recommend viewing the `tuner_fn()` first before looking at the other auxiliary functions.

[19]:
```
# Declare name of module file
_tuner_module_file = 'tuner.py'
```

[20]:
```
%%writefile {_tuner_module_file}

# Define imports
from kerastuner.engine import base_tuner
import kerastuner as kt
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

# Declare namedtuple field names
TunerFnResult = NamedTuple('TunerFnResult', [('tuner', base_tuner.BaseTuner),
                                             ('fit_kwargs', Dict[Text, Any])])

# Input key
_IMAGE_KEY = 'image_xf'

# Label key
_LABEL_KEY = 'label_xf'

# Callback for the search strategy
```

```python
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)


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.
  '''

  # Get feature specification based on transform output
  transformed_feature_spec = (
      tf_transform_output.transformed_feature_spec().copy())

  # Create batches of features and labels
  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.Input(shape=(28, 28, 1), name=_IMAGE_KEY))
  model.add(keras.layers.Flatten())

  # Tune the number of units in the first Dense layer
  # Choose an optimal value between 32-512
  hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
  model.add(keras.layers.Dense(units=hp_units, activation='relu',␣
↪name='dense_1'))

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

  # Tune the learning rate for the optimizer
  # Choose an optimal value from 0.01, 0.001, or 0.0001
  hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

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

  return model

def tuner_fn(fn_args: FnArgs) -> TunerFnResult:
  """Build the tuner using the KerasTuner API.
  Args:
    fn_args: Holds args as name/value pairs.
```

```
      - working_dir: working dir for tuning.
      - train_files: List of file paths containing training tf.Example data.
      - eval_files: List of file paths containing eval tf.Example data.
      - train_steps: number of train steps.
      - eval_steps: number of eval steps.
      - schema_path: optional schema of the input data.
      - transform_graph_path: optional transform graph produced by TFT.

  Returns:
    A namedtuple contains the following:
      - tuner: A BaseTuner that will be used for tuning.
      - fit_kwargs: Args to pass to tuner's run_trial function for fitting the
                    model , e.g., the training and validation dataset. Required
                    args depend on the above tuner's implementation.
  """

  # Define tuner search strategy
  tuner = kt.Hyperband(model_builder,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3,
                     directory=fn_args.working_dir,
                     project_name='kt_hyperband')

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

  # Use _input_fn() to extract input features and labels from the train and val␣
↪set
  train_set = _input_fn(fn_args.train_files[0], tf_transform_output)
  val_set = _input_fn(fn_args.eval_files[0], tf_transform_output)


  return TunerFnResult(
      tuner=tuner,
      fit_kwargs={
          "callbacks":[stop_early],
          'x': train_set,
          'validation_data': val_set,
          'steps_per_epoch': fn_args.train_steps,
          'validation_steps': fn_args.eval_steps
      }
  )
```

```
Writing tuner.py
```

With the module defined, you can now setup the Tuner component. You can see the description of each argument here.

Notice that we passed a `num_steps` argument to the train and eval args and this was used in the `steps_per_epoch` and `validation_steps` arguments in the tuner module above. This can be useful if you don't want to go through the entire dataset when tuning. For example, if you have 10GB of training data, it would be incredibly time consuming if you will iterate through it entirely just for one epoch and one set of hyperparameters. You can set the number of steps so your program will only go through a fraction of the dataset.

You can compute for the total number of steps in one epoch by: `number of examples / batch size`. For this particular example, we have `48000 examples / 32 (default size)` which equals `1500` steps per epoch for the train set (compute val steps from 12000 examples). Since you passed `500` in the `num_steps` of the train args, this means that some examples will be skipped. This will likely result in lower accuracy readings but will save time in doing the hypertuning. Try modifying this value later and see if you arrive at the same set of hyperparameters.

[21]:
```
# Setup the Tuner component
tuner = tfx.components.Tuner(
    module_file=_tuner_module_file,
    examples=transform.outputs['transformed_examples'],
    transform_graph=transform.outputs['transform_graph'],
    schema=schema_gen.outputs['schema'],
    train_args=trainer_pb2.TrainArgs(splits=['train'], num_steps=500),
    eval_args=trainer_pb2.EvalArgs(splits=['eval'], num_steps=100)
    )
```

[22]:
```
# Run the component. This will take around 10 minutes to run.
# When done, it will summarize the results and show the 10 best trials.
context.run(tuner, enable_cache=False)
```

```
Trial 30 Complete [00h 00m 21s]
val_accuracy: 0.8521875143051147

Best val_accuracy So Far: 0.8799999952316284
Total elapsed time: 00h 06m 28s
Results summary
Results in ./pipeline/.temp/6/kt_hyperband
Showing 10 best trials
<keras_tuner.engine.objective.Objective object at 0x7fd441c37f90>
Trial summary
Hyperparameters:
units: 352
learning_rate: 0.001
tuner/epochs: 10
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.8799999952316284
Trial summary
```

```
Hyperparameters:
units: 288
learning_rate: 0.001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0018
Score: 0.878125011920929
Trial summary
Hyperparameters:
units: 384
learning_rate: 0.001
tuner/epochs: 10
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.878125011920929
Trial summary
Hyperparameters:
units: 512
learning_rate: 0.0001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0023
Score: 0.8762500286102295
Trial summary
Hyperparameters:
units: 256
learning_rate: 0.001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 2
tuner/round: 2
tuner/trial_id: 0012
Score: 0.8753125071525574
Trial summary
Hyperparameters:
units: 256
learning_rate: 0.001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 2
tuner/round: 2
tuner/trial_id: 0013
Score: 0.8712499737739563
```

```
Trial summary
Hyperparameters:
units: 288
learning_rate: 0.001
tuner/epochs: 4
tuner/initial_epoch: 0
tuner/bracket: 1
tuner/round: 0
Score: 0.8571875095367432
Trial summary
Hyperparameters:
units: 256
learning_rate: 0.001
tuner/epochs: 4
tuner/initial_epoch: 2
tuner/bracket: 2
tuner/round: 1
tuner/trial_id: 0001
Score: 0.8559374809265137
Trial summary
Hyperparameters:
units: 160
learning_rate: 0.0001
tuner/epochs: 10
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.8521875143051147
Trial summary
Hyperparameters:
units: 512
learning_rate: 0.0001
tuner/epochs: 4
tuner/initial_epoch: 0
tuner/bracket: 1
tuner/round: 0
Score: 0.8471875190734863
```

```
[22]: ExecutionResult(
          component_id: Tuner
          execution_id: 6
          outputs:
              best_hyperparameters: OutputChannel(artifact_type=HyperParameters,
      producer_component_id=Tuner, output_key=best_hyperparameters,
      additional_properties={}, additional_custom_properties={})
              tuner_results: OutputChannel(artifact_type=TunerResults,
      producer_component_id=Tuner, output_key=tuner_results, additional_properties={},
additional_custom_properties={}))
```

1.3.8 Trainer

Like the Tuner component, the Trainer component also requires a module file to setup the training process. It will look for a `run_fn()` function that defines and trains the model. The steps will look similar to the tuner module file:

- Define the model - You can get the results of the Tuner component through the `fn_args.hyperparameters` argument. You will see it passed into the `model_builder()` function below. If you didn't run `Tuner`, then you can just explicitly define the number of hidden units and learning rate.
- Load the train and validation sets - You have done this in the Tuner component. For this module, you will pass in a `num_epochs` value (10) to indicate how many batches will be prepared. You can opt not to do this and pass a `num_steps` value as before.
- Setup and train the model - This will look very familiar if you're already used to the Keras Models Training API. You can pass in callbacks like the TensorBoard callback so you can visualize the results later.
- Save the model - This is needed so you can analyze and serve your model. You will get to do this in later parts of the course and specialization.

[23]:
```
# Declare trainer module file
_trainer_module_file = 'trainer.py'
```

[24]:
```
%%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

# Input key
_IMAGE_KEY = 'image_xf'

# 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.Input(shape=(28, 28, 1), name=_IMAGE_KEY))
    model.add(keras.layers.Flatten())

    # 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
    """

    # 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,
    )

# 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.

[25]:
```
# Setup the Trainer component
trainer = tfx.components.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 `Importer` component as shown in the official docs:

```
hparams_importer = Importer(
    # 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,
).with_id('import_hparams')

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'])
```

[26]:
```
# Run the component
context.run(trainer, enable_cache=False)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dense_1 (Dense)             (None, 352)               276320

 dropout_1 (Dropout)         (None, 352)               0

 dense_2 (Dense)             (None, 10)                3530

=================================================================
Total params: 279,850
Trainable params: 279,850
Non-trainable params: 0
_________________________________________________________________
14993/14993 [==============================] - 100s 7ms/step - loss: 0.3385 -
accuracy: 0.8753 - val_loss: 0.3154 - val_accuracy: 0.8895
```

[26]:
```
ExecutionResult(
    component_id: Trainer
    execution_id: 7
    outputs:
        model: OutputChannel(artifact_type=Model, producer_component_id=Trainer,
output_key=model, additional_properties={}, additional_custom_properties={})
        model_run: OutputChannel(artifact_type=ModelRun,
producer_component_id=Trainer, output_key=model_run, additional_properties={},
additional_custom_properties={}))
```

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`.

[27]:
```
# 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: ['fingerprint.pb', 'keras_metadata.pb',
'variables', 'saved_model.pb', 'assets']
```

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!