# 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*.

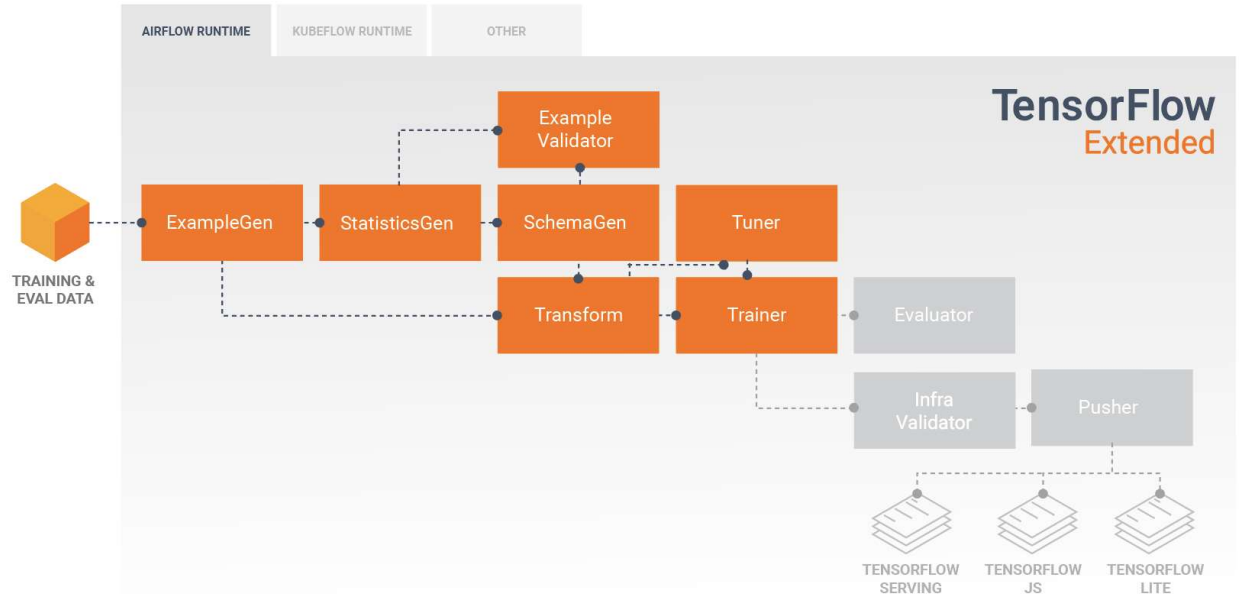


image source: https://www.tensorflow.org/tfx/guide

- 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!

## Setup

## Install TFX

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

```
!pip install -U tfx==1.3
```

```
  Attempting uninstall: prompt-toolkit
    Found existing installation: prompt-toolkit 1.0.18
    Uninstalling prompt-toolkit-1.0.18:
      Successfully uninstalled prompt-toolkit-1.0.18
  Attempting uninstall: packaging
    Found existing installation: packaging 21.0
    Uninstalling packaging-21.0:
      Successfully uninstalled packaging-21.0
  Attempting uninstall: ipython
    Found existing installation: ipython 5.5.0
    Uninstalling ipython-5.5.0:
      Successfully uninstalled ipython-5.5.0
  Attempting uninstall: google-api-core
    Found existing installation: google-api-core 1.26.3
    Uninstalling google-api-core-1.26.3:
      Successfully uninstalled google-api-core-1.26.3
  Attempting uninstall: pyarrow
    Found existing installation: pyarrow 3.0.0
    Uninstalling pyarrow-3.0.0:
      Successfully uninstalled pyarrow-3.0.0
  Attempting uninstall: google-resumable-media
    Found existing installation: google-resumable-media 0.4.1
    Uninstalling google-resumable-media-0.4.1:
      Successfully uninstalled google-resumable-media-0.4.1
  Attempting uninstall: google-cloud-core
    Found existing installation: google-cloud-core 1.0.3
    Uninstalling google-cloud-core-1.0.3:
      Successfully uninstalled google-cloud-core-1.0.3
  Attempting uninstall: future
    Found existing installation: future 0.16.0
    Uninstalling future-0.16.0:
      Successfully uninstalled future-0.16.0
  Attempting uninstall: dill
    Found existing installation: dill 0.3.4
    Uninstalling dill-0.3.4:
      Successfully uninstalled dill-0.3.4
  Attempting uninstall: google-cloud-language
    Found existing installation: google-cloud-language 1.2.0
    Uninstalling google-cloud-language-1.2.0:
      Successfully uninstalled google-cloud-language-1.2.0
  Attempting uninstall: google-cloud-bigquery
    Found existing installation: google-cloud-bigquery 1.21.0
    Uninstalling google-cloud-bigquery-1.21.0:
      Successfully uninstalled google-cloud-bigquery-1.21.0
  Attempting uninstall: attrs
    Found existing installation: attrs 21.2.0
    Uninstalling attrs-21.2.0:
      Successfully uninstalled attrs-21.2.0
  Attempting uninstall: joblib
    Found existing installation: joblib 1.0.1
    Uninstalling joblib-1.0.1:
      Successfully uninstalled joblib-1.0.1
  Attempting uninstall: google-cloud-storage
    Found existing installation: google-cloud-storage 1.18.1
    Uninstalling google-cloud-storage-1.18.1:
      Successfully uninstalled google-cloud-storage-1.18.1
ERROR: pip's dependency resolver does not currently take into account all the packag
pandas-gbq 0.13.3 requires google-cloud-bigquery[bqstorage,pandas]<2.0.0dev,>=1.11.1
multiprocess 0.70.12.2 requires dill>=0.3.4, but you have dill 0.3.1.1 which is inco
jupyter-console 5.2.0 requires prompt-toolkit<2.0.0,>=1.0.0, but you have prompt-too
google-colab 1.0.0 requires ipython~=5.5.0, but you have ipython 7.28.0 which is inc
google-colab 1.0.0 requires requests~=2.23.0, but you have requests 2.26.0 which is
```

```
google-colab 1.0.0 requires requests~=2.23.0, but you have requests 2.26.0 which is
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which is incomp
Successfully installed apache-beam-2.33.0 attrs-20.3.0 avro-python3-1.9.2.1 dill-0.3
WARNING: Upgrading ipython, ipykernel, tornado, prompt-toolkit or pyzmq can
cause your runtime to repeatedly crash or behave in unexpected ways and is not
recommended. If your runtime won't connect or execute code, you can reset it
with "Factory reset runtime" from the "Runtime" menu.
WARNING: The following packages were previously imported in this runtime:
  [IPython,google,prompt_toolkit]
You must restart the runtime in order to use newly installed versions.
```

RESTART RUNTIME

*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 Runtime` *at the end of the output cell above (after installation), or by selecting* `Runtime > Restart Runtime` *in the Menu bar.* ***Please do not proceed to the next section without restarting.*** *You can also ignore the errors about version incompatibility of some of the bundled packages because we won't be using those in this notebook.*

## Imports

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

```
import tensorflow as tf
from tensorflow import keras
import tensorflow_datasets as tfds

import os
import pprint

from tfx.components import ImportExampleGen
from tfx.components import ExampleValidator
from tfx.components import SchemaGen
from tfx.components import StatisticsGen
from tfx.components import Transform
from tfx.components import Tuner
from tfx.components import Trainer

from tfx.proto import example_gen_pb2
from tfx.orchestration.experimental.interactive.interactive_context import InteractiveCont
```

## Download 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.

```
# Location of the pipeline metadata store
_pipeline_root = './pipeline/'

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

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

```
# Create the dataset directory
!mkdir -p {_data_root}

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

You will now download 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`).

```
# Download the dataset
ds, ds_info = tfds.load('fashion_mnist', data_dir=tempdir, with_info=True)
```

```
Downloading and preparing dataset fashion_mnist/3.0.1 (download: 29.45 MiB, generate
DI Completed...: 100%  4/4 [00:03<00:00, 1.15 url/s]
DI Size...: 100%  29/29 [00:03<00:00, 17.03 MiB/s]
Extraction completed...: 100%  4/4 [00:03<00:00, 1.02s/ file]

Shuffling and writing examples to ./tempdir/fashion_mnist/3.0.1.incompleteGH0HVY/fas
100%  59999/60000 [00:00<00:00, 210171.45 examples/s]
Shuffling and writing examples to ./tempdir/fashion_mnist/3.0.1.incompleteGH0HVY/fas
100%  9999/10000 [00:00<00:00, 112666.45 examples/s]
Dataset fashion_mnist downloaded and prepared to ./tempdir/fashion_mnist/3.0.1. Subs
```

```
# Display info about the dataset
print(ds_info)
```

```
tfds.core.DatasetInfo(
    name='fashion_mnist',
    version=3.0.1,
    description='Fashion-MNIST is a dataset of Zalando's article images consisting o
    homepage='https://github.com/zalandoresearch/fashion-mnist',
    features=FeaturesDict({
        'image': Image(shape=(28, 28, 1), dtype=tf.uint8),
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=10),
    }),
    total_num_examples=70000,
    splits={
        'test': 10000,
        'train': 60000,
    },
    supervised_keys=('image', 'label'),
    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 Lea
                   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}
    }""",
    redistribution_info=,
)
```

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.

```
# 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)
```

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

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).

```
# 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}
```

## 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 just be using a local SQLite file for the metadata store.

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

```
WARNING:absl:InteractiveContext metadata_connection_config not provided: using SQLit
```

## 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.

```
# 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 = ImportExampleGen(input_base=_data_root, output_config=output)

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

```
WARNING:apache_beam.runners.interactive.interactive_environment:Dependencies require
WARNING:root:Make sure that locally built Python SDK docker image has Python 3.7 int
WARNING:apache_beam.io.tfrecordio:Couldn't find python-snappy so the implementation
```

▼**ExecutionResult** at 0x7f28c2f54710

| | |
|---|---|
| **.execution_id** | 1 |
| **.component** | ▶**ImportExampleGen** at 0x7f28b7808290 |
| **.component.inputs** | {} |
| **.component.outputs** | **['examples']** ▶**Channel** of type **'Examples'** (1 artifact) at 0x7f28b780831( |

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

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

## StatisticsGen

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

```
# Run StatisticsGen
statistics_gen = StatisticsGen(
    examples=example_gen.outputs['examples'])

context.run(statistics_gen)
```

```
WARNING:root:Make sure that locally built Python SDK docker image has Python 3.7 int
```

▼**ExecutionResult** at 0x7f28b77ee6d0

| | |
|---|---|
| **.execution_id** | 2 |
| **.component** | ▶**StatisticsGen** at 0x7f28c1f5dc90 |
| **.component.inputs** | **['examples']** ▶**Channel** of type **'Examples'** (1 artifact) at 0x7f28b780831( |
| **.component.outputs** | **['statistics']** ▶**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7f28c |

## 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.

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

▼**ExecutionResult** at 0x7f28c1412190

| | |
|---|---|
| **.execution_id** | 3 |
| **.component** | ▶**SchemaGen** at 0x7f28b7bf2cd0 |
| **.component.inputs** | **['statistics']** ▶**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7f28c |
| **.component.outputs** | **['schema']** ▶**Channel** of type **'Schema'** (1 artifact) at 0x7f28b7bf2410 |

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

**Artifact at ./pipeline/SchemaGen/schema/3**

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

## 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.

```
# Run ExampleValidator
example_validator = ExampleValidator(
    statistics=statistics_gen.outputs['statistics'],
```

```
    schema=schema_gen.outputs['schema'])
context.run(example_validator)
```

▼**ExecutionResult** at 0x7f28b5df7e10

| | |
|---|---|
| **.execution_id** | 4 |
| **.component** | ▶**ExampleValidator** at 0x7f28b808c150 |
| **.component.inputs** | **['statistics']** ▶**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7f28c<br>**['schema']** ▶**Channel** of type **'Schema'** (1 artifact) at 0x7f28b7bf2410 |
| **.component.outputs** | **['anomalies']** ▶**Channel** of type **'ExampleAnomalies'** (1 artifact) at 0x7f2 |

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

**Artifact at ./pipeline/ExampleValidator/anomalies/4**

**'train' split:**

```
/usr/local/lib/python3.7/dist-packages/tensorflow_data_validation/utils/display_util
  pd.set_option('max_colwidth', -1)
```

**No anomalies found.**

**'eval' split:**

**No anomalies found.**

## 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.

```
_transform_module_file = 'fmnist_transform.py'
```

```
%%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(_I

    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.*

```
# Ignore TF warning messages
tf.get_logger().setLevel('ERROR')

# Setup the Transform component
transform = 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 pu
WARNING:root:This output type hint will be ignored and not used for type-checking pu
WARNING:root:Make sure that locally built Python SDK docker image has Python 3.7 int
```

▼**ExecutionResult** at 0x7f28b78bc650

| | | |
|---|---|---|
| **.execution_id** | 5 | |
| **.component** | ▶**Transform** at 0x7f28b8315350 | |
| **.component.inputs** | **['examples']** | ▶**Channel** of type **'Examples'** (1 artifact) at 0x7f28b780831( |
| | **['schema']** | ▶**Channel** of type **'Schema'** (1 artifact) at 0x7f28b7bf2410 |
| **.component.outputs** | **['transform_graph']** | ▶**Channel** of type **'TransformGraph'** (1 a 0x7f28b8320290 |
| | **['transformed_examples']** | ▶**Channel** of type **'Examples'** (1 artifact) 0x7f28b83204d0 |
| | **['updated_analyzer_cache']** | ▶**Channel** of type **'TransformCache'** (1 a 0x7f28b8320210 |
| | **['pre_transform_schema']** | ▶**Channel** of type **'Schema'** (1 artifact) at |
| | **['pre_transform_stats']** | ▶**Channel** of type **'ExampleStatistics'** (1 0x7f28b8320c10 |
| | **['post_transform_schema']** | ▶**Channel** of type **'Schema'** (1 artifact) at |

## 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.

```
# Declare name of module file
_tuner_module_file = 'tuner.py'
```

```
%%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])])

# Label key
LABEL_KEY = 'label_xf'

# Callback for the search strategy
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.Flatten(input_shape=(28, 28, 1)))

    # 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.

```
from tfx.proto import trainer_pb2

# Setup the Tuner component
tuner = 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)
    )


# 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)
```

```
tuner/bracket: 0
tuner/round: 0
Score: 0.8603125214576721
Trial summary
Hyperparameters:
units: 416
learning_rate: 0.0001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 6ca5abe7968c93a84e082fe4a2b27e09
Score: 0.8550000190734863
Trial summary
Hyperparameters:
units: 384
learning_rate: 0.0001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: a0fdc3b9fb08334ce255d571f2d05950
Score: 0.8537499904632568
Trial summary
Hyperparameters:
units: 384
learning_rate: 0.0001
tuner/epochs: 4
tuner/initial_epoch: 0
tuner/bracket: 1
tuner/round: 0
Score: 0.8500000238418579
Trial summary
Hyperparameters:
units: 64
learning_rate: 0.01
tuner/epochs: 10
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.8493750095367432
Trial summary
Hyperparameters:
units: 320
learning_rate: 0.001
tuner/epochs: 2
tuner/initial_epoch: 0
tuner/bracket: 2
tuner/round: 0
Score: 0.8465625047683716
Trial summary
Hyperparameters:
units: 448
learning_rate: 0.001
tuner/epochs: 4
tuner/initial_epoch: 2
tuner/bracket: 2
tuner/round: 1
tuner/trial_id: 5171612281f9db3f87581b0825f4fa02
Score: 0.8465625047683716
```

▼ **ExecutionResult** at 0x7f28b60741d0

| | | |
|---|---|---|
| **.execution_id** | 6 | |
| **.component** | ▶**Tuner** at 0x7f28b4726050 | |
| **.component.inputs** | **['examples']** | ▶**Channel** of type **'Examples'** (1 artifact) at 0x7f28b |
| | **['schema']** | ▶**Channel** of type **'Schema'** (1 artifact) at 0x7f28b7b |
| | **['transform_graph']** | ▶**Channel** of type **'TransformGraph'** (1 artifact) at 0 |
| **.component.outputs** | **['best_hyperparameters']** | ▶**Channel** of type **'HyperParameters'** (1 artifa 0x7f28b4726b90 |

## 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.

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

```
%%writefile {_trainer_module_file}

from tensorflow import keras
```