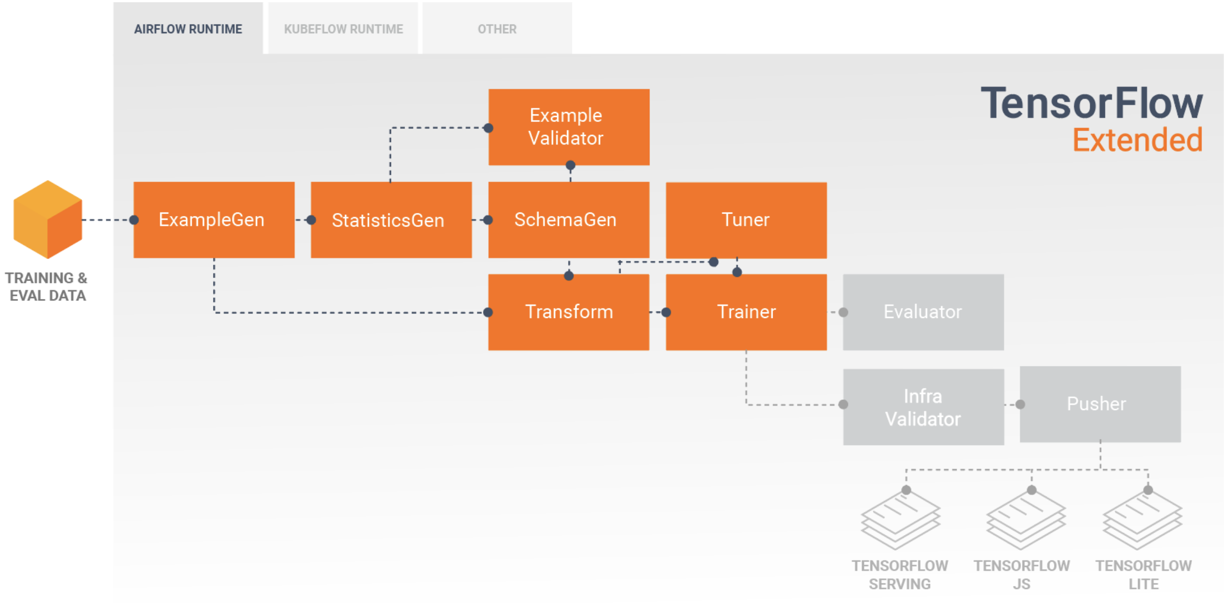
Hyperparameter tuning and model training with TFX

Doing hyperparameter tuning within a Tensorflow Extended (TFX) pipeline.



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

The *Tuner* utilizes the Keras Tuner API under the hood to tune a model's hyperparameters.

Setup

Install TFX

```
!pip install -U pip
!pip install tfx==1.12.0
```

In Google Colab, need to restart the runtime at this point to finalize updating the packages just installed. Click the `Restart Runtime` at the end of the output cell above (after installation), or by selecting `Runtime > Restart Runtime` in the Menu bar.

Imports

```
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)
```

Download and prepare the dataset

```
# 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}
```

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

```
Downloading and preparing dataset Unknown size (download: Unknown size, generated: Unknown siz
Dl Completed...: 100%    4/4 [00:05<00:00, 1.42s/ url]

Dl Size...: 100%    29/29 [00:05<00:00, 9.58 MiB/s]

Extraction completed...: 100%    4/4 [00:05<00:00, 1.60s/ file]
```

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

```
# 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)
```

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

Copy the train split so it can be consumed by the ExampleGen component

```
# 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

Initialize the Interactive Context

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

ExampleGen

ExampleGen is the initial input component of a pipeline that ingests and optionally splits the input dataset. ImportExampleGen consumes TFRecords.

```
# 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 Inte
WARNING:apache_beam.io.tfrecordio:Couldn't find python-snappy so the implementation of _TFReco
```

```
# 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

StatisticsGen calculates statistics for the dataset.

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

context.run(statistics_gen)
```

▼ExecutionResult at 0x7f75129b1820

.execution_id	2
.component	▶**StatisticsGen** at 0x7f75129b1670
.component.inputs	**['examples']** ▶**Channel** of type **'Examples'** (1 artifact) at 0x7f750dc7dac0
.component.outputs	**['statistics']** ▶**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7f75129b1b20

SchemaGen

SchemaGen infers a data schema, and validates incoming data to ensure that it is formatted correctly.

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

▼ExecutionResult at 0x7f750cf02700

.execution_id	3
.component	▶**SchemaGen** at 0x7f75129b19d0
.component.inputs	**['statistics']** ▶**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7f75129b1b20
.component.outputs	**['schema']** ▶**Channel** of type **'Schema'** (1 artifact) at 0x7f750cf025b0

```
# 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

ExampleValidator looks for anomalies and missing values in the dataset.

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

▼ExecutionResult at 0x7f75129b1e80

.execution_id	4
.component	▶**ExampleValidator** at 0x7f7512bf8e80
.component.inputs	**['statistics']** ▶**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7f75129b1b20 **['schema']** ▶**Channel** of type **'Schema'** (1 artifact) at 0x7f750cf025b0
.component.outputs	**['anomalies']** ▶**Channel** of type **'ExampleAnomalies'** (1 artifact) at

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

Artifact at ./pipeline/ExampleValidator/anomalies/4

'train' split:

No anomalies found.

'eval' split:

No anomalies found.

Transform

Transform performs feature engineering on the dataset.

```
_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(_IMAGE_KEY)])

    return outputs
```

```
Writing fmnist_transform.py
```

Pass in the examples, schema, and transform module file.

Ignore the warnings and `udf_utils` related errors.

```
# 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. Ty
WARNING:root:This output type hint will be ignored and not used for type-checking purposes. Ty
WARNING:root:This input type hint will be ignored and not used for type-checking purposes. Typ
WARNING:root:This output type hint will be ignored and not used for type-checking purposes. Ty
WARNING:root:This input type hint will be ignored and not used for type-checking purposes. Typ
WARNING:root:This output type hint will be ignored and not used for type-checking purposes. Ty
```

▼**ExecutionResult** at 0x7f750d78df10

.execution_id	5	
.component	▶**Transform** at 0x7f750d78dd30	
.component.inputs	**['examples']**	▶**Channel** of type **'Examples'** (1 artifact) at 0x7f750dc7dac0
	['schema']	▶**Channel** of type **'Schema'** (1 artifact) at 0x7f750cf025b0
.component.outputs	**['transform_graph']**	▶**Channel** of type **'TransformGraph'** (1 artifact) at 0x7f750cf0d580
	['transformed_examples']	▶**Channel** of type **'Examples'** (1 artifact) at 0x7f750cf0df70
	['updated_analyzer_cache']	▶**Channel** of type **'TransformCache'** (1 artifact) at 0x7f750cf0d3a0
	['pre_transform_schema']	▶**Channel** of type **'Schema'** (1 artifact) at 0x7f750cf0d670
	['pre_transform_stats']	▶**Channel** of type **'ExampleStatistics'** (1

Tuner

Prepare a *tuner module file* which contains a `tuner_fn()` function.

In `_input_fn()`, the transformed examples as TFRecords compressed in `.gz` format are loaded into the memory. Once loaded, create batches of features and labels for hypertuning.

`tuner_fn()` returns a `TunerFnResult` tuple containing the `tuner` object and a set of arguments to pass to `tuner.search()` method.

```
# 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])]
)

# Input key
_IMAGE_KEY = 'image_xf'

# 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.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
```

Pass a `num_steps` argument to the train and eval args and it is used in the `steps_per_epoch` and `validation_steps` arguments in the tuner module above. This can be useful for avoiding going through the entire dataset when tuning. For example, training data is very large, it would be incredibly time consuming to iterate through it entirely just for one epoch and one set of hyperparameters. Set the number of steps so to only go through a fraction of the dataset.

Total number of steps in one epoch = `number of examples / batch size`. In this example, `48000 examples / 32 (default size)` = `1500` steps per epoch for the train set (compute val steps from 12000 examples). Since `500` is passed to 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.

```
# 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)
)
```

```
# 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 01m 23s]
val_accuracy: 0.8637499809265137

Best val_accuracy So Far: 0.8853124976158142
Total elapsed time: 00h 12m 29s
Results summary
Results in ./pipeline/.temp/6/kt_hyperband
Showing 10 best trials
<keras_tuner.engine.objective.Objective object at 0x7f750b447550>
Trial summary
Hyperparameters:
units: 256
learning_rate: 0.001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0018
Score: 0.8853124976158142
Trial summary
Hyperparameters:
units: 224
learning_rate: 0.001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 2
tuner/round: 2
tuner/trial_id: 0015
Score: 0.8812500238418579
Trial summary
Hyperparameters:
units: 96
learning_rate: 0.001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 2
tuner/round: 2
tuner/trial_id: 0013
Score: 0.870312511920929
Trial summary
Hyperparameters:
units: 448
learning_rate: 0.0001
tuner/epochs: 10
tuner/initial_epoch: 4
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0023
Score: 0.8675000071525574
Trial summary
Hyperparameters:
units: 224
learning_rate: 0.001
tuner/epochs: 4
tuner/initial_epoch: 2
tuner/bracket: 2
tuner/round: 1
tuner/trial_id: 0007
Score: 0.8668749928474426
Trial summary
Hyperparameters:
units: 352
learning_rate: 0.0001
tuner/epochs: 10
tuner/initial_epoch: 0
tuner/bracket: 0
tuner/round: 0
Score: 0.8637499809265137
Trial summary
Hyperparameters:
units: 352
learning_rate: 0.001
tuner/epochs: 2
tuner/initial_epoch: 0
tuner/bracket: 2
tuner/round: 0
Score: 0.8587499856948853
Trial summary
Hyperparameters:
units: 96
learning_rate: 0.001
tuner/epochs: 4
tuner/initial_epoch: 2
tuner/bracket: 2
tuner/round: 1
tuner/trial_id: 0008
Score: 0.856249988079071
Trial summary
```

Trainer

The Trainer component looks for a `run_fn()` function that defines and trains the model.

Get the best result of the Tuner component through `fn_args.hyperparameters`, and pass it into `model_builder()`. Alternatively, just explicitly define the number of hidden units and learning rate.

Trial summary

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

tuner/epochs: 4

```
%%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
    """

    # 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
```

```
# 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'])
)
```

When re-training your model, don't always have to re-tune hyperparameters. Can import it with the ImporterNode.

```
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 (kerastuner.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)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 flatten_1 (Flatten)         (None, 784)               0         
                                                                 
 dense_1 (Dense)             (None, 256)               200960    
                                                                 
 dropout_1 (Dropout)         (None, 256)               0         
                                                                 
 dense_2 (Dense)             (None, 10)                2570      
                                                                 
=================================================================
Total params: 203,530
Trainable params: 203,530
Non-trainable params: 0
_________________________________________________________________
14993/14993 [==============================] - 147s 10ms/step - loss: 0.3424 - accuracy: 0.874
```

▼**ExecutionResult** at 0x7f7533520040

.execution_id	7		
.component	▶**Trainer** at 0x7f75227a2be0		
.component.inputs		**['examples']**	▶**Channel** of type **'Examples'** (1 artifact) at 0x7f750cf0df70
		['transform_graph']	▶**Channel** of type **'TransformGraph'** (1 artifact) at 0x7f750cf0d580
		['schema']	▶**Channel** of type **'Schema'** (1 artifact) at 0x7f750cf025b0
		['hyperparameters']	▶**Channel** of type **'HyperParameters'** (1 artifact) at

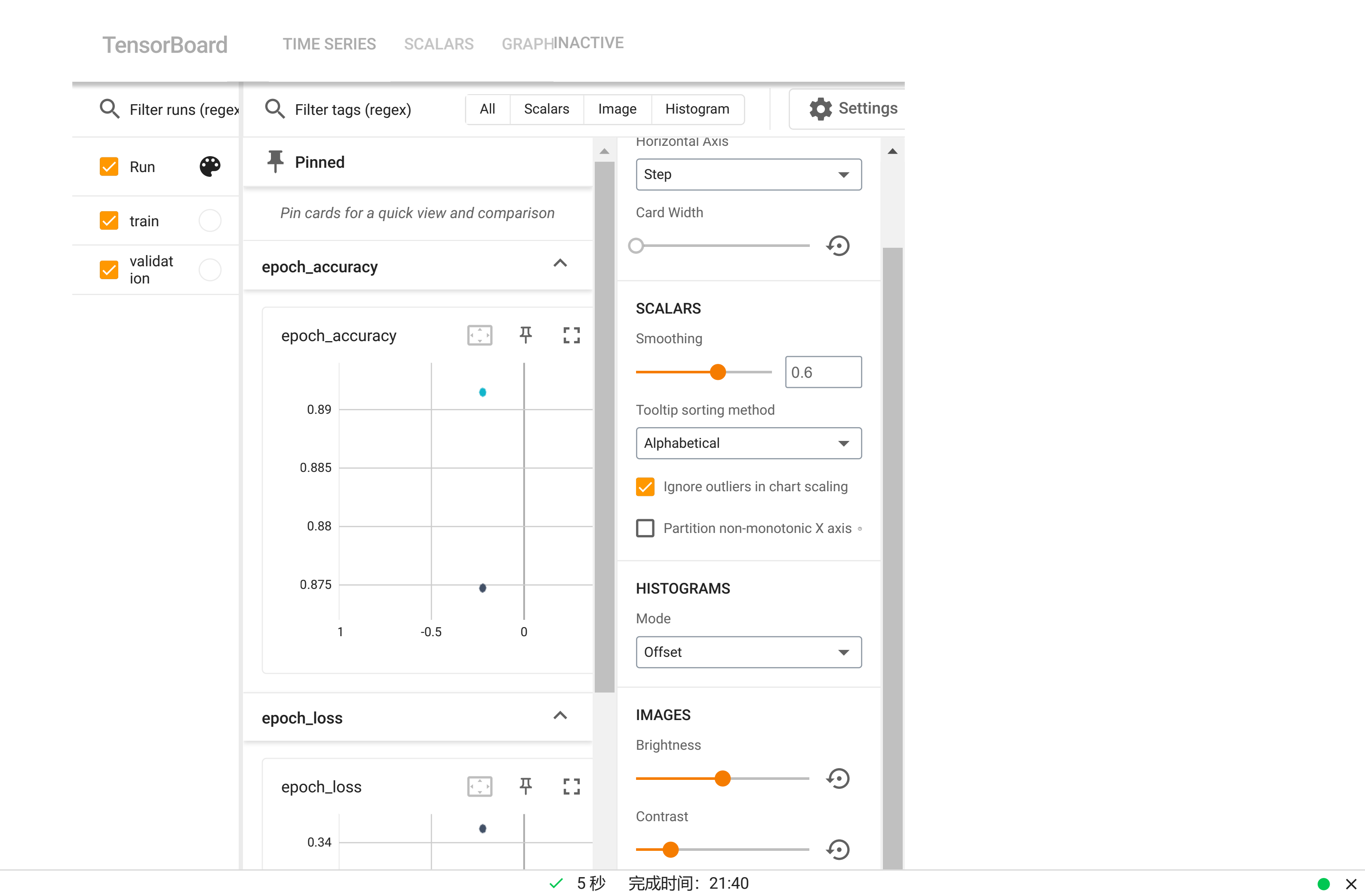
The file is saved as `saved_model.pb`

```
# 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', 'variables', 'assets', 'fingerprint.pb', 'keras_metadata.pb']
```

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}
```


TensorBoard
TIME SERIES
SCALARS
GRAPHINACTIVE
Filter runs (regex
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
train
validat
ion
Pinned
Pin cards for a quick view and comparison
epoch_accuracy
epoch_accuracy
0.89
0.885
0.88
0.875
1
-0.5
0
epoch_loss
epoch_loss
0.34
Horizontal Axis
Step
Card Width
SCALARS
Smoothing
0.6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset
IMAGES
Brightness
Contrast
5 秒
完成时间：21:40