Chapter 6. Model Training

Now that the data preprocessing step is complete and the data has been transformed into the format that our model requires, the next step in our pipeline is to train the model with the freshly transformed data.

As we discussed in Chapter 1, we won't cover the process of choosing your model architecture. We assume that you have a separate experimentation process that took place before you even picked up this book and that you already know the type of model you wish to train. We discuss how to track this experimentation process in Chapter 15 because it helps with creating a full audit trail for the model. However, we don't cover any of the theoretical background you'll need to understand the model training process. If you would like to learn more about this, we strongly recommend the O'Reilly publication *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd edition.

In this chapter, we cover the model training process as part of a machine learning pipeline, including how it is automated in a TFX pipeline. We also include some details of distribution strategies available in TensorFlow and how to tune hyperparameters in a pipeline. This chapter is more specific to TFX pipelines than most of the others because we don't cover training as a standalone process.

As shown in Figure 6-1, by this point data has been ingested, validated, and preprocessed. This ensures that all the data needed by the model is present and that it has been reproducibly transformed into the features that the model requires. All of this is necessary because we don't want the pipeline to fail at our next step. We want to ensure that the training proceeds smoothly because it is often the most time-consuming part of the entire pipeline.

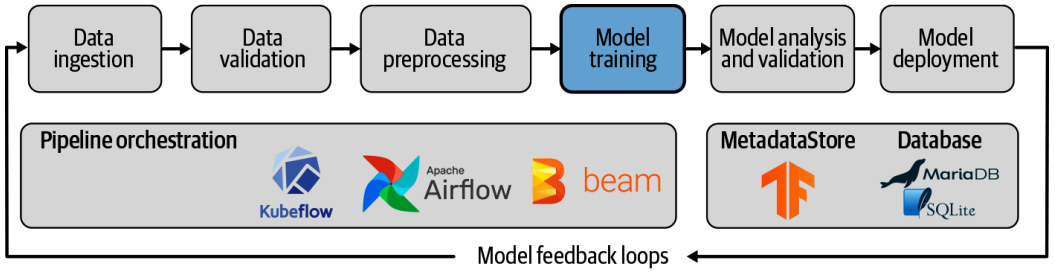

Figure 6-1. Model training as part of ML pipelines

One very important feature of training a model in a TFX pipeline is that the data preprocessing steps that we discussed in Chapter 5 are saved along with the trained model weights. This is incredibly useful once our model is deployed to production because it means that the preprocessing steps will always produce the features the model is expecting. Without this feature, it would be possible to update the data preprocessing steps without updating the model, and then the model would fail in production or the predictions would be based on the wrong data. Because we export the preprocessing steps and the model as one graph, we eliminate this potential source of error.

In the next two sections, we'll take a detailed look at the steps required to train a `tf.Keras` model as part of a TFX pipeline.[1]

Defining the Model for Our Example Project

Even though the model architecture is already defined, some extra code is necessary here. We need to make it possible to automate the model training part of the pipeline. In this section, we will briefly describe the model we use throughout this chapter.

The model for our example project is a hypothetical implementation, and we could probably optimize the model architecture. However, it showcases some common ingredients of many deep learning models:

- Transfer learning from a pretrained model
- Dense layers
- Concatenation layers

As we discussed in Chapter 1, the model in our example project uses data from the US Consumer Finance Protection Bureau to predict whether a consumer disputed a complaint about a financial product. The features in our model include the financial product, the company's response, the US state, and the consumer complaint narrative. Our model is inspired by the Wide and Deep model architecture, with the addition of the Universal Sentence Encoder from TensorFlow Hub to encode the free-text feature (the consumer complaint narrative).

You can see a visual representation of our model architecture in Figure 6-2, with the text feature (`narrative_xf`) taking the "deep" route and the other features taking the "wide" route.

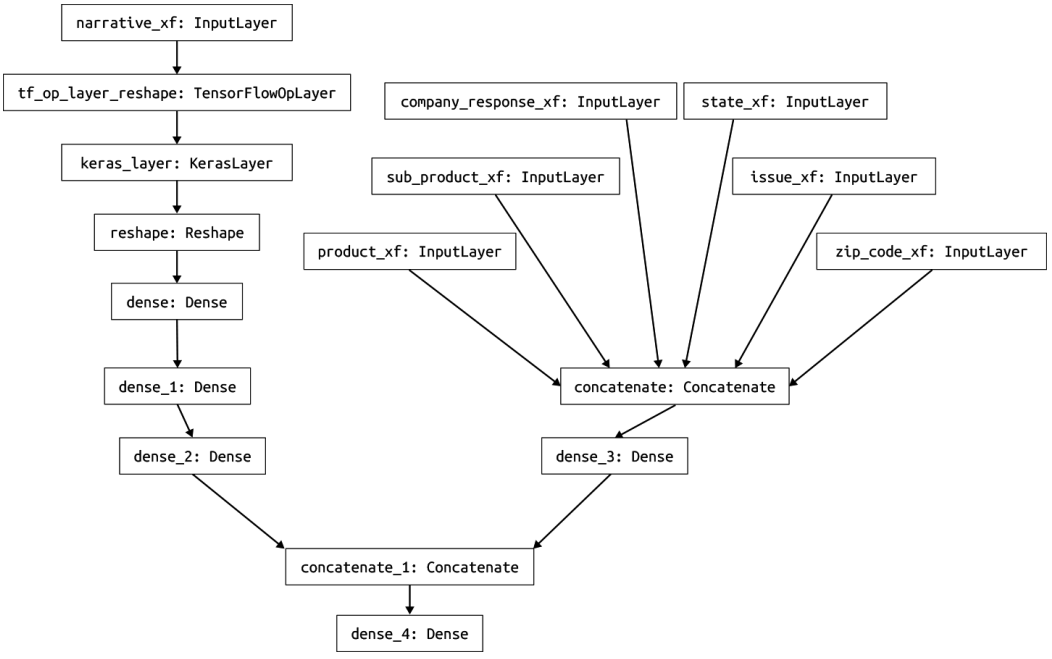

Figure 6-2. Model architecture for our example project

Example 6-1 shows the full model architecture definition. Because we want to export the model with our preprocessing steps, we need to guarantee that the model input names match the transformed feature names from `preprocessing_fn()` , which we discussed in Chapter 5. In our example model, we reuse the `transformed_name()` function described in Chapter 5 to add the suffix `_xf` to our features.

Example 6-1. Defining our model architecture

```
import tensorflow as tf
import tensorflow_hub as hub

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

def get_model():

    # One-hot categorical features
    input_features = []
    for key, dim in ONE_HOT_FEATURES.items(): ❶
        input_features.append(
            tf.keras.Input(shape=(dim + 1,),
                           name=transformed_name(key)))
```

```
    # Adding bucketized features
    for key, dim in BUCKET_FEATURES.items():
        input_features.append(
            tf.keras.Input(shape=(dim + 1,),
                           name=transformed_name(key)))

    # Adding text input features
    input_texts = []
    for key in TEXT_FEATURES.keys():
        input_texts.append(
            tf.keras.Input(shape=(1,),
                           name=transformed_name(key),
                           dtype=tf.string))

    inputs = input_features + input_texts

    # Embed text features
    MODULE_URL = "https://tfhub.dev/google/universal-sentence-encoder/4"
    embed = hub.KerasLayer(MODULE_URL) ❷
    reshaped_narrative = tf.reshape(input_texts[0], [-1]) ❸
    embed_narrative = embed(reshaped_narrative)
    deep_ff = tf.keras.layers.Reshape((512, ), input_shape=(1, 512))(embed_narrati

    deep = tf.keras.layers.Dense(256, activation='relu')(deep_ff)
    deep = tf.keras.layers.Dense(64, activation='relu')(deep)
    deep = tf.keras.layers.Dense(16, activation='relu')(deep)

    wide_ff = tf.keras.layers.concatenate(input_features)
    wide = tf.keras.layers.Dense(16, activation='relu')(wide_ff)

    both = tf.keras.layers.concatenate([deep, wide])

    output = tf.keras.layers.Dense(1, activation='sigmoid')(both)
    keras_model = tf.keras.models.Model(inputs, output) ❹

    keras_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                        loss='binary_crossentropy',
                        metrics=[
                            tf.keras.metrics.BinaryAccuracy(),
                            tf.keras.metrics.TruePositives()
                        ])
    return keras_model
```

❶ Loop over the features and create an input for each feature.

❷ Load the `tf.hub` module of the Universal Sentence Encoder model.

❸ Keras inputs are two-dimensional, but the encoder expects one-dimensional inputs.

❹ Assemble the model graph with the functional API.

Now that we have defined our model, let's move on to describe the process to integrate it into a TFX pipeline.

The TFX Trainer Component

The TFX `Trainer` component handles the training step in our pipeline. In this section, we will first describe how to train the Keras model from the example project in a one-off training run. At the end of the section, we will add some considerations for other training situations and for `Estimator` models.

All the steps we will describe may seem lengthy and unnecessary compared to the normal Keras training code. But the key point here is that the `Trainer` component will produce a model that will be put into production, where it will transform new data and use the model to make predictions. Because the `Transform` steps are included in this model, the data preprocessing steps will always match what the model is expecting. This removes a huge potential source of errors when our model is deployed.

In our example project, the `Trainer` component requires the following inputs:

- The previously generated data schema, generated by the data validation step discussed in Chapter 4
- The transformed data and its preprocessing graph, as discussed in Chapter 5
- Training parameters (e.g., the number of training steps)
- A module file containing a `run_fn()` function, which defines the training process

In the next section, we will discuss the setup of the `run_fn` function. We also will cover how to train a machine learning model in our pipeline and

export it to the next pipeline step that we will discuss in Chapter 7.

run_fn() Function

The `Trainer` component will look for a `run_fn()` function in our module file and use the function as an entry point to execute the training process. The module file needs to be accessible to the `Trainer` component. If you run the component in an interactive context, you can simply define the absolute path to the module file and pass it to the component. If you run your pipelines in production, please check Chapter 11 or Chapter 12 for details on how to provide the module file.

The `run_fn()` function is a generic entry point to the training steps and not `tf.Keras` specific. It carries out the following steps:

- Loading the training and validation data (or the data generator)
- Defining the model architecture and compiling the model
- Training the model
- Exporting the model to be evaluated in the next pipeline step

The `run_fn` for our example project performs these four steps as shown in Example 6-2.

Example 6-2. `run_fn()` function of our example pipeline

```
def run_fn(fn_args):

    tf_transform_output = tft.TFTransformOutput(fn_args.transform_output)
    train_dataset = input_fn(fn_args.train_files, tf_transform_output) ❶
    eval_dataset = input_fn(fn_args.eval_files, tf_transform_output)

    model = get_model() ❷
    model.fit(
        train_dataset,
        steps_per_epoch=fn_args.train_steps,
        validation_data=eval_dataset,
        validation_steps=fn_args.eval_steps) ❸

    signatures = {
        'serving_default':
            _get_serve_tf_examples_fn(
                model,
                tf_transform_output).get_concrete_function(
```

```
                    tf.TensorSpec(
                        shape=[None],
                        dtype=tf.string,
                        name='examples')
                )
    } ❹
    model.save(fn_args.serving_model_dir,
               save_format='tf', signatures=signatures)
```

❶ Call the `input_fn` to get data generators.

❷ Call the `get_model` function to get the compiled Keras model.

❸ Train the model using the number of training and evaluation steps passed by the `Trainer` component.

❹ Define the model signature, which includes the serving function we will describe later.

This function is fairly generic and could be reused with any other `tf.Keras` model. The project-specific details are defined in helper functions like `get_model()` or `input_fn()`.

In the following sections, we want to take a closer look into how we load the data, train, and export our machine learning model inside the `run_fn()` function.

Load the data

The following lines in the `run_fn` load our training and evaluation data:

```
def run_fn(fn_args):
    tf_transform_output = tft.TFTransformOutput(fn_args.transform_output)
    train_dataset = input_fn(fn_args.train_files, tf_transform_output)
    eval_dataset = input_fn(fn_args.eval_files, tf_transform_output)
```

In the first line, the `run_fn` function receives a set of arguments, including the transform graph, example datasets, and training parameters through the `fn_args` object.

Data loading for model training and validation is performed in batches, and the loading is handled by the `input_fn()` function as shown in Example 6-3.

Example 6-3. `Input_fn` function of our example pipeline

```
LABEL_KEY = 'labels'

def _gzip_reader_fn(filenames):
    return tf.data.TFRecordDataset(filenames,
        compression_type='GZIP')

def input_fn(file_pattern,
             tf_transform_output, batch_size=32):

    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,
        label_key=transformed_name(LABEL_KEY)) ❶

    return dataset
```

❶ The dataset will be batched into the correct batch size.

The `input_fn` function lets us load the compressed, preprocessed datasets that were generated by the previous Transform step.[2] To do this, we need to pass the `tf_transform_output` to the function. This gives us the data schema to load the dataset from the TFRecord data structures generated by the Transform component. By using the preprocessed datasets, we can avoid data preprocessing during training and speed up the training process.

The `input_fn` returns a generator (a `batched_features_dataset`) that will supply data to the model one batch at a time.

Compile and train the model

Now that we have defined our data-loading steps, the next step is defining our model architecture and compiling our model. In our `run_fn`, this will require a call to `get_model()`, which we have described, so it just needs a single line of code:

```
model = get_model()
```

Next, we train our compiled `tf.Keras` model with the Keras method *fit()*:

```
model.fit(
    train_dataset,
    steps_per_epoch=fn_args.train_steps,
    validation_data=eval_dataset,
    validation_steps=fn_args.eval_steps)
```

TRAINING STEPS VERSUS EPOCHS

The TFX `Trainer` component defines the training process by the number of training steps rather than by epochs. A *training step* is when the model is trained on a single batch of data. The benefit of using steps rather than epochs is that we can train or validate models with large datasets and only use a fraction of the data. At the same time, if you want to loop over the training dataset multiple times during training, you can increase the step size to a multiple of the available samples.

Once the model training is complete, the next step is to export the trained model. We will have a detailed discussion about exporting models for deployment in Chapter 8. In the following section, we want to highlight how the preprocessing steps can be exported with the model.

Model export

Finally, we export the model. We combine the preprocessing steps from the previous pipeline component with the trained model and save the model in TensorFlow's *SavedModel* format. We define a *model signature* based on the graph generated by the Example 6-4 function. We will describe model signatures in much more detail in "Model Signatures" in Chapter 8.

In the `run_fn` function, we define the model signature and save the model with the following code:

```
signatures = {
    'serving_default':
        _get_serve_tf_examples_fn(
            model,
            tf_transform_output).get_concrete_function(
                tf.TensorSpec(
                    shape=[None],
                    dtype=tf.string,
                    name='examples')
            )
}
model.save(fn_args.serving_model_dir,
            save_format='tf', signatures=signatures)
```

The `run_fn` exports the `get_serve_tf_examples_fn` as part of the model signature. When a model has been exported and deployed, every prediction request will pass through the `serve_tf_examples_fn()` shown in Example 6-4. With every request, we parse the serialized `tf.Example` records and apply the preprocessing steps to the raw request data. The model then makes a prediction on the preprocessed data.

Example 6-4. Applying the preprocessing graph to model inputs

```
def get_serve_tf_examples_fn(model, tf_transform_output):

    model.tft_layer = tf_transform_output.transform_features_layer() ❶

    @tf.function
    def serve_tf_examples_fn(serialized_tf_examples):
        feature_spec = tf_transform_output.raw_feature_spec()
        feature_spec.pop(LABEL_KEY)
        parsed_features = tf.io.parse_example(
            serialized_tf_examples, feature_spec) ❷

        transformed_features = model.tft_layer(parsed_features) ❸
        outputs = model(transformed_features) ❹
        return {'outputs': outputs}

    return serve_tf_examples_fn
```

❶ Load the preprocessing graph.

❷ Parse the raw `tf.Example` records from the request.

❸ Apply the preprocessing transformation to raw data.

❹ Perform prediction with preprocessed data.

With the definition of our `run_fn()` function in place, let's discuss how we can run the `Trainer` component.

Running the Trainer Component

As shown in Example 6-5, the `Trainer` component takes the following as input:

- The Python module file, here saved as *module.py*, containing the `run_fn()`, `input_fn()`, `get_serve_tf_examples_fn()`, and other associated functions we discussed earlier
- The transformed examples generated by the Transform component
- The transform graph generated by the Transform component
- The schema generated by the data validation component
- The number of training and evaluation steps

Example 6-5. `Trainer` component

```
from tfx.components import Trainer
from tfx.components.base import executor_spec
from tfx.components.trainer.executor import GenericExecutor ❶
from tfx.proto import trainer_pb2

TRAINING_STEPS = 1000
EVALUATION_STEPS = 100

trainer = Trainer(
    module_file=os.path.abspath("module.py"),
    custom_executor_spec=executor_spec.ExecutorClassSpec(GenericExecutor), ❷
    transformed_examples=transform.outputs['transformed_examples'],
    transform_graph=transform.outputs['transform_graph'],
    schema=schema_gen.outputs['schema'],
```

```
        train_args=trainer_pb2.TrainArgs(num_steps=TRAINING_STEPS),
        eval_args=trainer_pb2.EvalArgs(num_steps=EVALUATION_STEPS))
```

❶ Load the `GenericExecutor` to override the training executor.

❷ Override the executor to load the `run_fn()` function.

In a notebook environment (an interactive context), we can run the `Trainer` component, like any previous component, with the following command:

```
context.run(trainer)
```

After the model training and exporting is completed, the component will register the path of the exported model with the metadata store. Downstream components can pick up the model for the model validation.

The `Trainer` component is generic and not limited to running TensorFlow models. However, the components later in the pipeline expect that the model is saved in the TensorFlow SavedModel format. The SavedModel graph includes the Transform graph, so the data preprocessing steps are part of the model.

OVERRIDING THE TRAINER COMPONENT'S EXECUTOR

In our example project, we override the `Trainer` component's executor to enable the generic training entry point `run_fn()` function instead of the default `trainer_fn()` function, which only supports `tf.Estimator` models. In Chapter 12, we will introduce another Trainer executor, the `ai_platform_trainer_executor.GenericExecutor`. This executor allows you to train models on Google Cloud's AI Platform instead of inside your pipeline. This is an alternative if your model requires specific training hardware (e.g., GPUs or tensor processing units [TPUs]), which aren't available in your pipeline environment.

Other Trainer Component Considerations

In our examples so far in this chapter, we have only considered a single training run of a Keras model. But we can also use the `Trainer` compo-

nent to fine-tune a model from a previous run or to train multiple models simultaneously, and we will describe these in "Advanced Pipeline Concepts". We can also use it to optimize a model through hyperparameter search, and we will discuss this more in "Model Tuning".

In this section, we will also discuss how to use the `Trainer` component with an `Estimator` model and how to load your SavedModel exported by the `Trainer` component outside a TFX pipeline.

Using the Trainer component with an Estimator model

Until recently, TFX supported only `tf.Estimator` models and the `Trainer` component was solely designed for `Estimators`. The default implementation of the `Trainer` component used the `trainer_fn()` function as an entry point to the training process, but this entry point is very `tf.Estimator` specific. The `Trainer` component expects the Estimator inputs to be defined by functions like `train_input_fn()`, `eval_input_fn()`, and `serving_receiver_fn` ().[3]

As we discussed in "Running the Trainer Component", the core functionality of the component can be swapped out with the generic training executor `GenericExecutor`, which uses the `run_fn()` function as its entry point to the training process.[4] As the name of the executor implies, the training process becomes generic and not tied to `tf.Estimator` or `tf.Keras` models.

Using the SavedModel outside a pipeline

If we would like to inspect the exported SavedModel outside a TFX pipeline, we can load the model as a *concrete function*,[5] which represents the graph of a single signature:

```
model_path = trainer.outputs.model.get()[0].uri
model = tf.saved_model.load(export_dir=model_path)
predict_fn = model.signatures["serving_default"]
```

With the model loaded as a concrete function, we can now perform predictions. The exported model expects the input data to be provided in the `tf.Example` data structure as shown in the following example. More details around the `tf.Example` data structure, and how other features (like

integers and floats) can be converted can be found in Example 3-1. The following code shows how to create the serialized data structure and perform a model prediction by calling the `prediction_fn()` function:

```
example = tf.train.Example(features=tf.train.Features(feature={
    'feature_A': _bytes_feature(feature_A_value),
    ...
    })) ❶

serialized_example = example.SerializeToString()
print(predict_fn(tf.constant([serialized_example])))
```

❶ The `_bytes_feature` helper function is defined in Example 3-1.

If you would like to inspect the progress of the model in detail during training, you can do this using TensorBoard. We will describe how to use TensorBoard in our pipeline in the next section.

Using TensorBoard in an Interactive Pipeline

TensorBoard is another wonderful tool that is part of the TensorFlow ecosystem. It has many helpful functions that we can use in our pipelines, for example, monitoring metrics while training, visualizing word embeddings in NLP problems, or viewing activations for layers in the model. A new Profiler feature lets us profile the model to understand performance bottlenecks.

An example of TensorBoard's basic visualization is shown in Figure 6-3.

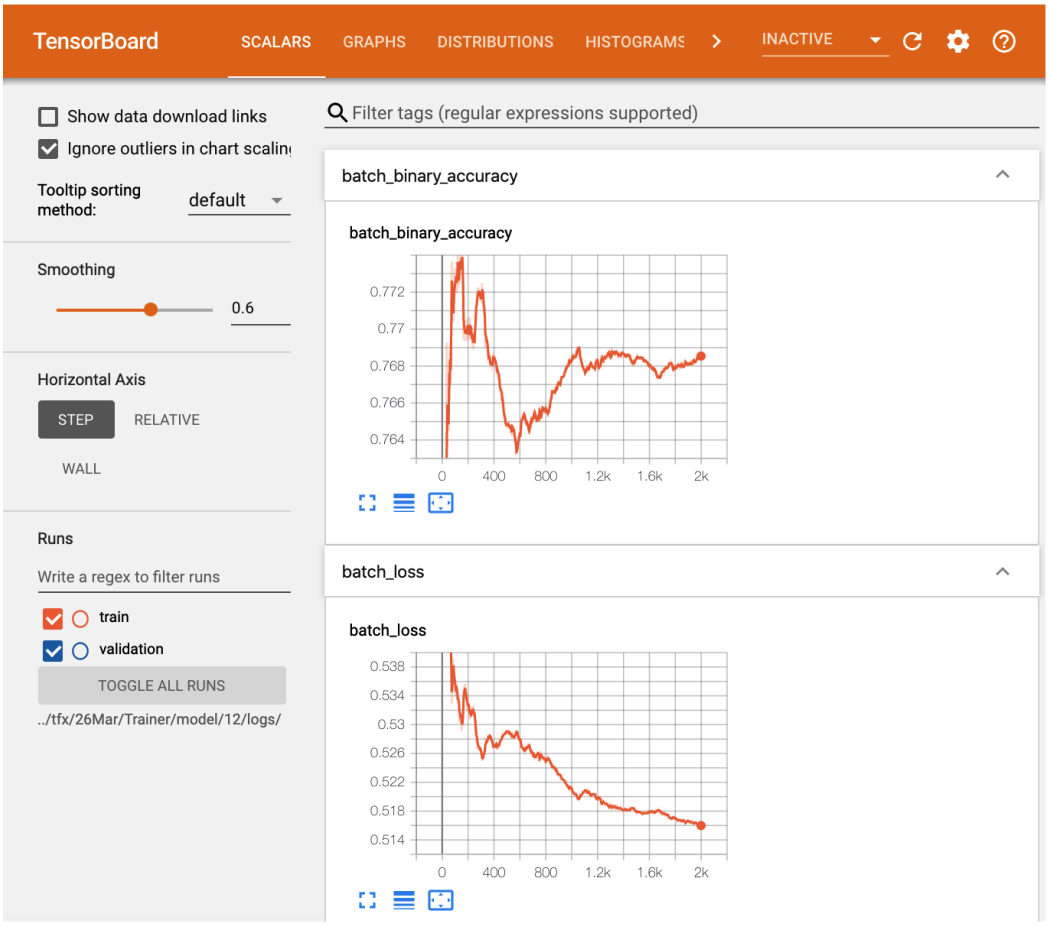

Figure 6-3. Viewing metrics while training in TensorBoard

To be able to use TensorBoard in our pipeline, we need to add callbacks in the `run_fn` function and log the training to a folder we specify:

```
log_dir = os.path.join(os.path.dirname(fn_args.serving_model_dir), 'logs')
tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir=log_dir, update_freq='batch')
```

We also need to add the callback to our model training:

```
model.fit(
    train_dataset,
    steps_per_epoch=fn_args.train_steps,
    validation_data=eval_dataset,
    validation_steps=fn_args.eval_steps,
    callbacks=[tensorboard_callback])
```

Then, to view TensorBoard in a notebook, we get the location of the model training logs and pass it to TensorBoard:

```
model_dir = trainer.outputs['output'].get()[0].uri

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

We can also use TensorBoard outside a notebook by running:

```
tensorboard --logdir path/to/logs
```

Then connect to *http://localhost:6006/* to view TensorBoard. This gives us a larger window to view the details.

Next, we will introduce some useful strategies for training large models on multiple GPUs.

Distribution Strategies

TensorFlow provides distribution strategies for machine learning models that can't be adequately trained on a single GPU. You might want to consider distribution strategies when you want to accelerate your training or you can't fit the entire model into a single GPU.

The strategies we describe here are abstractions to distribute the model parameters across multiple GPUs or even multiple servers. In general, there are two groups of strategies: *synchronous* and *asynchronous* training. Under the synchronous strategies, all training workers train with different slices of the training data synchronously and then aggregate the gradients from all workers before updating the model. The asynchronous strategies train models independently with the entire dataset on different workers. Each worker updates the gradients of the model asynchronously, without waiting for the other workers to finish. Typically, synchronous strategies are coordinated via all-reduce operations[6] and asynchronous strategies through a parameter server architecture.

A few synchronous and asynchronous strategies exist, and they have their benefits and drawbacks. At the time of writing this section, Keras supports the following strategies:

MirroredStrategy

This strategy is relevant for multiple GPUs on a single instance, and it follows the synchronous training pattern. The strategy *mirrors* the model and the parameters across the workers, but each worker receives a different batch of data. The MirroredStrategy is a good default strategy if you train a machine learning model on a single node with multiple GPUs and your machine learning model fits in the GPU memory.

CentralStorageStrategy

In contrast to the MirroredStrategy, the variables in this strategy aren't mirrored across all GPUs. Instead, they are stored in the CPU's memory and then copied into the assigned GPU to execute the relevant operations. In case of a single GPU operation, the CentralStorageStrategy will store the variables on the GPU, not in the CPU. CentralStorageStrategy is a good strategy for distributing your training when you train on a single node with multiple GPUs and your complete model doesn't fit in the memory of single GPU, or when the communication bandwidth between the GPUs is too limited.

MultiWorkerMirroredStrategy

This follows the design patterns of the MirroredStrategy, but it copies the variables across multiple workers (e.g., compute instances). The MultiWorkerMirroredStrategy is an option if one node isn't enough for your model training.

TPUStrategy

This strategy lets you use Google Cloud's TPUs. It follows the synchronous training pattern and basically works like MirroredStrategy except it uses TPUs instead of GPUs. It requires its own strategy since the MirroredStrategy uses GPU-specific all-reduce functions. TPUs have a huge amount of RAM available, and the cross-TPU communication is highly optimized, which is why the TPU strategy uses the mirrored approach.

ParameterServerStrategy

The ParameterServerStrategy uses multiple nodes as the central variable repository. This strategy is useful for models exceeding the available resources (e.g., RAM or I/O bandwidth) of a single node.

The ParameterServerStrategy is your only option if you can't train on a single node and the model is exceeding the RAM or I/O limitations of a node.

OneDeviceStrategy

The whole point of the OneDeviceStrategy is to test the entire model setup before engaging in real distributed training. This strategy forces the model training to only use one device (e.g., one GPU). Once it is confirmed that the training setup is working, the strategy can be swapped.

NOT ALL STRATEGIES ARE AVAILABLE VIA THE TFX TRAINER COMPONENT

At the time of writing this section, the TFX `Trainer` component only supports the MirroredStrategy. While the different strategies can currently be used with `tf.keras`, they will be made accessible via the `Trainer` component in the second half of 2020, according to the TFX roadmap.

Because the MirroredStrategy is supported by the TFX Trainer, we'll show an example of it here. We can apply the MirroredStrategy easily by adding a few lines before invoking our model creation and the subsequent `model.compile()` call:

```
mirrored_strategy = tf.distribute.MirroredStrategy() ❶
with mirrored_strategy.scope(): ❷
    model = get_model()
```

❶ Instance of distribution strategy.

❷ Wrap model creation and compilation with Python manager.

In this example setup, we create an instance of the MirroredStrategy. In order to apply the distribution strategy to our model, we wrap the model creation and compilation with the Python manager (in our case, it all happens inside of the `get_model()` function). This will create and compile our model under the distribution scope of our choice. The MirroredStrategy will use all available GPUs of the instance. If you want to reduce the number of GPU instances being used (e.g., in case you share

instances), you can specify the GPUs to be used with the MirroredStrategy by changing the creation of the distribution strategy:

```
mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

In this example, we specify two GPUs to be used for our training runs.

BATCH SIZE REQUIREMENT WHEN USING THE MIRROREDSTRATEGY

The MirroredStrategy expects that the batch size is proportional to the number of devices. For example, if you train with five GPUs, the batch size needs to be a multiple of the number of GPUs. Please keep this in mind when you set up your `input_fn()` function as described in Example 6-3.

These distribution strategies are useful for large training jobs that won't fit on the memory of a single GPU. Model tuning, which we will discuss in the next section, is a common reason for us to need these strategies.

Model Tuning

Hyperparameter tuning is an important part of achieving an accurate machine learning model. Depending on the use case, it may be something that we do during our initial experiments or it may be something we want to include in our pipeline. This is not a comprehensive introduction to model tuning, but we will give a brief overview of it here and describe how it may be included in a pipeline.

Strategies for Hyperparameter Tuning

Depending on the type of model in your pipeline, the choice of hyperparameters will be different. If your model is a deep neural network, hyperparameter tuning is especially critical to achieving good performance with neural networks. Two of the most important sets of hyperparameters to tune are those controlling the optimization and the network architecture.

For optimization, we recommend using Adam or NAdam by default. The learning rate is a very important parameter to experiment with, and

there are many possible options for learning rate schedulers. We recommend using the largest batch size that fits in your GPU memory.

For very large models, we suggest the following steps:

- Tune the initial learning rate, starting with 0.1.
- Pick a number of steps to train for (as many as patience allows).
- Linearly decay the learning rate to 0 over the specified number of steps.

For smaller models, we recommend using early stopping to avoid overfitting. With this technique, model training is stopped when the validation loss does not improve after a user-defined number of epochs.

For the network architecture, two of the most important parameters to tune are the size and number of layers. Increasing these will improve training performance, but it may cause overfitting and will mean the model takes longer to train. You can also consider adding residual connections between the layers, particularly for deep architectures.

The most popular hyperparameter search approaches are *grid search* and *random search*. In grid search, every combination of parameters is tried exhaustively, whereas in random search, parameters are sampled from the available options and may not try every combination. Grid search can get extremely time consuming if the number of possible hyperparameters is large. After trying a range of values, you can fine-tune by taking the best-performing hyperparameters and starting a new search centered on them.

In the TensorFlow ecosystem, hyperparameter tuning is implemented using the Keras Tuner and also Katib, which provides hyperparameter tuning in Kubeflow. In addition to grid and random search, both of these packages support Bayesian search and the Hyperband algorithm.

Hyperparameter Tuning in TFX Pipelines

In a TFX pipeline, hyperparameter tuning takes in the data from the Transform component and trains a variety of models to establish the best hyperparameters. The hyperparameters are then passed to the `Trainer` component, which then trains a final model using them.

In this case, the model definition function (the `get_model` function in our example) needs to accept the hyperparameters as an input and build the model according to the specified hyperparameters. So, for example, the number of layers needs to be defined as an input argument.

THE TFX TUNER COMPONENT

The TFX Tuner component was released as we were finalizing this book. You can view the source code in the project's GitHub repo.

Summary

In this chapter, we described how to move our model training from a standalone script to an integrated part of our pipeline. This means that the process can be automated and triggered whenever we would like—as soon as new data arrives in the pipeline or when the previous model accuracy dips beneath a predefined level. We also described how the model and the data preprocessing steps are saved together to avoid any errors from a mismatch between the preprocessing and the training. We additionally covered strategies for distributing the model training and for tuning the hyperparameters.

Now that we have a saved model, the next step is to dive into the details of what it can do.

1 We use a Keras model in our example project, but TFX also works perfectly with an `Estimator` model. Examples can be found in the TFX documentation.

2 The `Trainer` component could be used without the previous `Transform` component, and we could load the raw datasets. However, in this case, we would miss out on an excellent feature of TFX, which is exporting the preprocessing and the model graphs as one SavedModel graph.

3 `tf.Keras` models can be converted to `tf.Estimator` models through the `tf.model_to_estimator()` conversion. However, with the recent updates to TFX, this is no longer the recommended best practice.

4 If you are interested in the steps of how component executors can be developed and exchanged, we recommend the section "Reusing Existing Components" in

Chapter 10.

5 For more details on concrete functions, check out the TensorFlow documentation.

6 The all-reduce operation reduces information from all the workers to a single information; in other words, it enables synchronization between all training workers.