Chapter 10. Advanced TensorFlow Extended

With the previous two chapters on model deployments, we completed our overview of individual pipeline components. Before we take a deep dive into orchestrating these pipeline components, we want to pause and introduce advanced concepts of TFX in this chapter.

With the pipeline components we have introduced so far, we can create machine learning pipelines for most problems. However, sometimes we need to build our own TFX component or more complex pipeline graphs. Therefore, in this chapter, we will focus on how to build custom TFX components. We introduce the topic with a custom ingestion component that ingests images directly for computer vision ML pipelines. Furthermore, we will introduce advanced concepts of pipeline structures: generating two models simultaneously (e.g., for deployments with TensorFlow Serving and TFLite), as well as adding a human reviewer into the pipeline workflow.

ONGOING DEVELOPMENTS

At the time of this writing, some of the concepts we are introducing are still under development and, therefore, might be subject to future updates. We have done our best to update code examples with changes to the TFX functionality throughout the production of this publication, and all examples work with TFX 0.22. Updates to the TFX APIs can be found in the TFX documentation.

Advanced Pipeline Concepts

In this section, we will discuss three additional concepts to advance your pipeline setups. So far, all the pipeline concepts we've discussed comprised linear graphs with one entry and one exit point. In Chapter 1, we discussed the fundamentals of directed acyclic graphs. As long as our pipeline graph is directed and doesn't create any circular connections, we

can be creative with our setup. In the next sections, we will highlight a few concepts to increase the productivity of pipelines by:

- Training multiple models simultaneously
- Exporting models for mobile deployments
- Warm starting model training

Training Multiple Models Simultaneously

As we mentioned before, you can train multiple models simultaneously. A common use case for training multiple models from the same pipeline is when you want to train a different type of model (e.g., a more simplistic model), but you want to make sure that the trained model is getting fed with exactly the same transformed data and the exact same transform graph. Figure 10-1 shows how this setup would work.

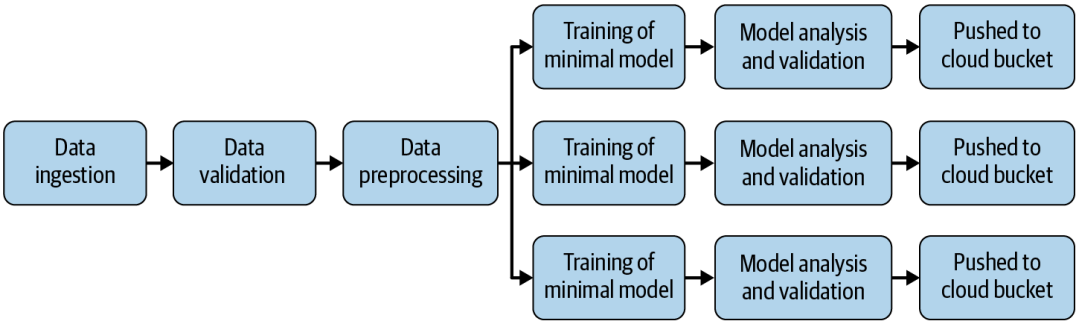

Figure 10-1. Training multiple models simultaneously

You can assemble such a graph with TFX by defining multiple `Trainer` components, as shown in the following code example:

```
def set_trainer(module_file, instance_name,
                train_steps=5000, eval_steps=100): ❶
    return Trainer(
        module_file=module_file,
        custom_executor_spec=executor_spec.ExecutorClassSpec(
            GenericExecutor),
        examples=transform.outputs['transformed_examples'],
        transform_graph=transform.outputs['transform_graph'],
        schema=schema_gen.outputs['schema'],
        train_args=trainer_pb2.TrainArgs(num_steps=train_steps),
        eval_args=trainer_pb2.EvalArgs(num_steps=eval_steps),
        instance_name=instance_name)
```

```
    prod_module_file = os.path.join(pipeline_dir, 'prod_module.py') ❷
    trial_module_file = os.path.join(pipeline_dir, 'trial_module.py')
    ...

    trainer_prod_model = set_trainer(module_file, 'production_model') ❸
    trainer_trial_model = set_trainer(trial_module_file, 'trial_model',
                                      train_steps=10000, eval_steps=500)
    ...
```

❶ Function to instantiate the `Trainer` efficiently.

❷ Load module for each `Trainer`.

❸ Instantiate a `Trainer` component for each graph branch.

At this step, we basically branch the graph into as many training branches as we want to run simultaneously. Each of the `Trainer` components consumes the same inputs from the ingestion, schema, and `Transform` components. The key difference between the components is that each component can run a different training setup, which is defined in the individual training module files. We have also added the arguments for the training and evaluation steps as a parameter to the function. This allows us to train two models with the same training setup (i.e., the same module file), but we can compare the models based on the different training runs.

Each instantiated training component needs to be consumed by its own `Evaluator`, as shown in the following code example. Afterward, the models can be pushed by its own `Pusher` components:

```
evaluator_prod_model = Evaluator(
    examples=example_gen.outputs['examples'],
    model=trainer_prod_model.outputs['model'],
    eval_config=eval_config_prod_model,
    instance_name='production_model')

evaluator_trial_model = Evaluator(
    examples=example_gen.outputs['examples'],
    model=trainer_trial_model.outputs['model'],
    eval_config=eval_config_trial_model,
```

```
        instance_name='trial_model')

    ...
```

As we have seen in this section, we can assemble fairly complex pipeline scenarios using TFX. In the following section, we will discuss how we can amend a training setup to export models for mobile deployments with TFLite.

Exporting TFLite Models

Mobile deployments have become an increasingly important platform for machine learning models. Machine learning pipelines can help with consistent exports for mobile deployments. Very few changes are required for mobile deployment compared to deployment to model servers (such as TensorFlow Serving, as discussed in Chapter 8). This helps keep the mobile and the server models updated consistently and helps the consumers of the model have a consistent experience across different devices.

TFLITE LIMITATIONS

Because of hardware limitations of mobile and edge devices, TFLite doesn't support all TensorFlow operations. Therefore, not every model can be converted to a TFLite-compatible model. For more information on which TensorFlow operations are supported, visit the TFLite website.

In the TensorFlow ecosystem, TFLite is the solution for mobile deployments. TFLite is a version of TensorFlow that can be run on edge or mobile devices. Figure 10-2 shows how the pipeline can include two training branches.

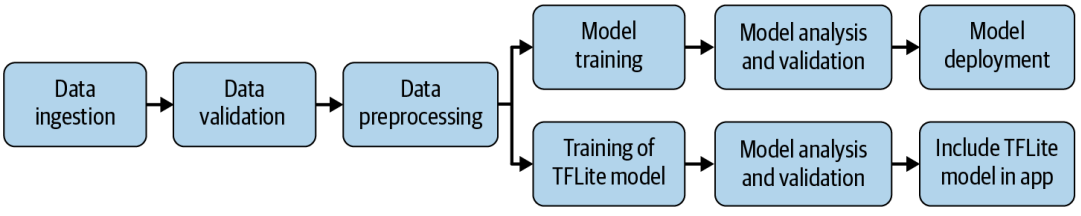

Figure 10-2. Exporting models for deployments in mobile apps

We can use the branch strategy we discussed in the previous section and amend our `run_fn` function of the module file to rewrite the saved models to a TFLite-compatible format.

Example 10-1 shows the additional functionality we need to add to our `run_fn` function.

Example 10-1. TFX Rewriter example

```
from tfx.components.trainer.executor import TrainerFnArgs
from tfx.components.trainer.rewriting import converters
from tfx.components.trainer.rewriting import rewriter
from tfx.components.trainer.rewriting import rewriter_factory

def run_fn(fn_args: TrainerFnArgs):
    ...
    temp_saving_model_dir = os.path.join(fn_args.serving_model_dir, 'temp')
    model.save(temp_saving_model_dir,
               save_format='tf',
               signatures=signatures) ❶

    tfrw = rewriter_factory.create_rewriter(
        rewriter_factory.TFLITE_REWRITER,
        name='tflite_rewriter',
        enable_experimental_new_converter=True
    ) ❷
    converters.rewrite_saved_model(temp_saving_model_dir, ❸
                                   fn_args.serving_model_dir,
                                   tfrw,
                                   rewriter.ModelType.TFLITE_MODEL)

    tf.io.gfile.rmtree(temp_saving_model_dir) ❹
```

❶ Export the model as a saved model.

❷ Instantiate the TFLite rewriter.

❸ Convert the model to TFLite format.

❹ Delete the saved model after conversion.

Instead of exporting a saved model after the training, we convert the saved model to a TFLite-compatible model and delete the saved model after exporting it. Our `Trainer` component then exports and registers the TFLite model with the metadata store. The downstream components like the `Evaluator` or the `Pusher` can then consume the TFLite-compliant model. The following example shows how we can evaluate the TFLite model, which is helpful in detecting whether the model optimizations (e.g., quantization) have led to a degradation of the model's performance:

```
eval_config = tfma.EvalConfig(
    model_specs=[tfma.ModelSpec(label_key='my_label', model_type=tfma.TF_LITE)],
    ...
)

evaluator = Evaluator(
    examples=example_gen.outputs['examples'],
    model=trainer_mobile_model.outputs['model'],
    eval_config=eval_config,
    instance_name='tflite_model')
```

With this presented pipeline setup, we can now produce models for mobile deployment automatically and push them in the artifact stores for model deployment in mobile apps. For example, a `Pusher` component could ship the produced TFLite model to a cloud bucket where a mobile developer could pick up the model and deploy it with Google's *ML Kit* in an iOS or Android mobile app.

CONVERTING MODELS TO TENSORFLOW.JS

Since TFX version 0.22, an additional feature of the `rewriter_factory` is available: the conversion of preexisting TensorFlow models to TensorFlow.js models. This conversion allows the deployment of models to web browsers and Node.js runtime environments. You can use this new functionality by replacing the `rewriter_factory` name with `rewriter_factory.TFJS_REWRITER` and set the `rewriter.ModelType` to `rewriter.ModelType.TFJS_MODEL` in Example 10-1.

Warm Starting Model Training

In some situations, we may not want to start training a model from scratch. *Warm starting* is the process of beginning our model training from a checkpoint of a previous training run, which is particularly useful if the model is large and training is time consuming. This may also be useful in situations under the General Data Protection Regulation (GDPR), the European privacy law that states that a user of a product can withdraw their consent for the use of their data at any time. By using warm start training, we can remove only the data belonging to this particular user and fine-tune the model rather than needing to begin training again from scratch.

In a TFX pipeline, warm start training requires the `Resolver` component that we introduced in Chapter 7. The `Resolver` picks up the details of the latest trained model and passes them on to the `Trainer` component:

```
latest_model_resolver = ResolverNode(
    instance_name='latest_model_resolver',
    resolver_class=latest_artifacts_resolver.LatestArtifactsResolver,
    latest_model=Channel(type=Model))
```

The latest model is then passed to the `Trainer` using the `base_model` argument:

```
trainer = Trainer(
    module_file=trainer_file,
    transformed_examples=transform.outputs['transformed_examples'],
    custom_executor_spec=executor_spec.ExecutorClassSpec(GenericExecutor),
    schema=schema_gen.outputs['schema'],
    base_model=latest_model_resolver.outputs['latest_model'],
    transform_graph=transform.outputs['transform_graph'],
    train_args=trainer_pb2.TrainArgs(num_steps=TRAINING_STEPS),
    eval_args=trainer_pb2.EvalArgs(num_steps=EVALUATION_STEPS))
```

The pipeline then continues as normal. Next, we want to introduce another useful feature we can add to our pipeline.

Human in the Loop

As part of the advanced TFX concepts, we want to highlight an experimental component that could elevate your pipeline setup. All the pipelines we have discussed so far run automatically from start to finish, and they might deploy your machine learning model automatically. Some TFX users have expressed their concerns about the fully automated setup because they wanted a human to review the trained model after the automatic model analysis. This could be to spot check your trained model or to gain confidence in the automated pipeline setup.

In this section, we will discuss the functionality of a *human in the loop* component. In Chapter 7, we discussed that once a model passes the validation step, it is "blessed." The downstream `Pusher` component listens to this blessing signal to know whether to push the model or not. But such a blessing can also be generated by a human, as Figure 10-3 shows.

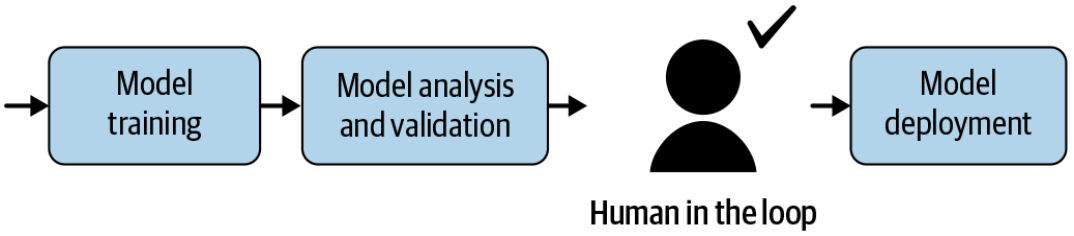

Figure 10-3. Human in the loop

Google's TFX team published a Slack notification component as an example of this custom component. The functionality we are discussing in this section could be extended and isn't limited to the Slack messenger.

The component's functionality is pretty straightforward. Once it is triggered by the orchestration tool, it submits a message to a given Slack channel with a link to the latest exported model and asks for a review by a data scientist (shown in Figure 10-4). A data scientist could now investigate the model manually with the WIT and review edge cases that aren't tested during the `Evaluator` step.

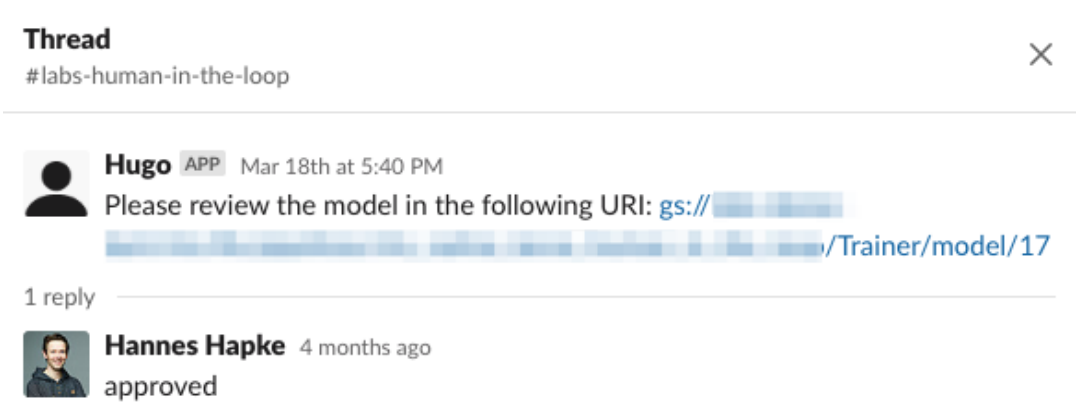

Figure 10-4. Slack message asking for review

Once the data scientist concludes their manual model analysis, they can respond in the Slack thread with their approval or rejection. The TFX component listens to the Slack responses and stores the decision in the metadata store. The decision can then be used by the downstream components. It is tracked in the model's audit trail. Figure 10-5 shows an example record from Kubeflow Pipeline's lineage browser. The metadata store tracks the "blessing" by the data scientist (i.e., the decision maker) and the time stamp (the Slack thread ID `1584638332.0001` identifies the timestamp as the time in Unix epoch format).

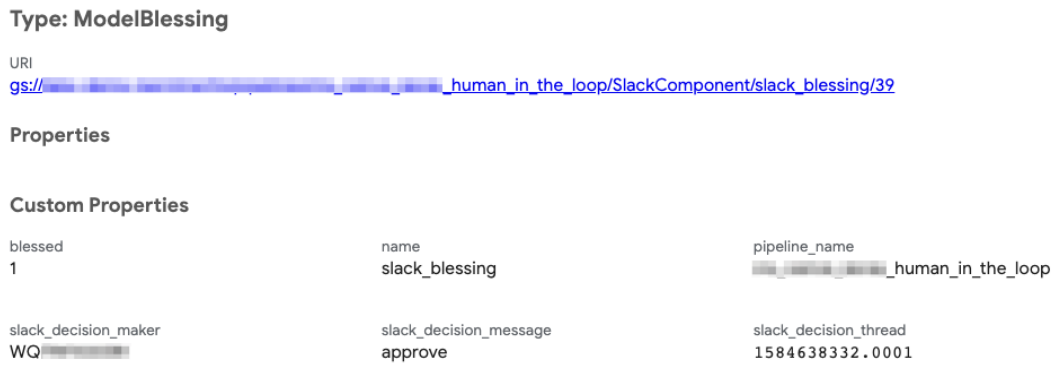

Figure 10-5. Audit trail in Kubeflow Pipelines

Slack Component Setup

For the Slack component to communicate with your Slack account, it requires a Slack *bot token*. You can request a bot token through the Slack API. Once you have a token, set an environment variable in your pipeline environment with the token string as shown in the following `bash` command:

```
$ export SLACK_BOT_TOKEN={your_slack_bot_token}
```

The Slack component is not a standard TFX component and therefore needs to be installed separately. You can install the component by cloning the TFX repository from GitHub and then installing the component individually:

```
$ git clone https://github.com/tensorflow/tfx.git

$ cd tfx/tfx/examples/custom_components/slack
$ pip install -e .
```

Once the component package is installed in your Python environment, the component can then be found on the Python path and loaded inside your TFX scripts. An example is shown in the following Python code. Please also remember to install the Slack component in the environment where you run your TFX pipelines. For example, if you run your pipelines with Kubeflow Pipelines, you will have to create a custom Docker image for your pipeline component, which contains the source code of the Slack component (since it isn't a standard TFX component).

How to Use the Slack Component

The installed Slack component can be loaded like any other TFX component:

```
from slack_component.component import SlackComponent

slack_validator = SlackComponent(
    model=trainer.outputs['model'],
    model_blessing=model_validator.outputs['blessing'],
    slack_token=os.environ['SLACK_BOT_TOKEN'], ❶
    slack_channel_id='my-channel-id', ❷
    timeout_sec=3600,
)
```

❶ Load the Slack token from your environment.

❷ Specify the channel where the message should appear.

When executed, the component will post a message and wait up to an hour (defined by the `timeout_sec` argument) for an answer. During this time, a data scientist can evaluate the model and respond with their approval or rejection. The downstream component (e.g., a `Pusher` component) can consume the result from the Slack component, as shown in the following code example:

```
pusher = Pusher(
    model=trainer.outputs['model'],
    model_blessing=slack_validator.outputs['slack_blessing'], ❶
    push_destination=pusher_pb2.PushDestination(
        filesystem=pusher_pb2.PushDestination.Filesystem(
            base_directory=serving_model_dir)))
```

❶ Model blessing provided by the Slack component.

With a few additional steps, you can enrich your pipelines with a human audit of the machine learning models that is triggered by the pipeline itself. This opens up many more workflows for pipeline applications (e.g., auditing dataset statistics or reviewing data drift metrics).

SLACK API STANDARDS

The implementation of the Slack component relies on the *Real Time Messaging* (RTM) protocol. This protocol is deprecated and might be replaced by a new protocol standard, which would affect the component's functionality.

Custom TFX Components

In Chapter 2, we discussed the architecture of TFX components and how each component consists of three parts: the driver, executor, and publisher. In this section, we want to go a little deeper and discuss how you can build your own components. First, we discuss how to write a component from scratch, and afterward, we'll discuss how to reuse existing components and customize them for your own use cases. In general, it is always easier to change an existing component's functionality than to write a component from scratch.

To demonstrate the implementation to you, as seen in Figure 10-6, we will develop a custom component for ingesting JPEG images and its labels in the pipeline. We will load all images from a provided folder and determine the label based on the filename. In our example, we want to train a machine learning model to classify cats and dogs. The filenames of our images carry the content of the image (e.g., *dog-1.jpeg*), so we can determine the label from the filename itself. We will load each image, convert it to `tf.Example` data structures, and save all samples together as TFRecord files for consumption by downstream components.

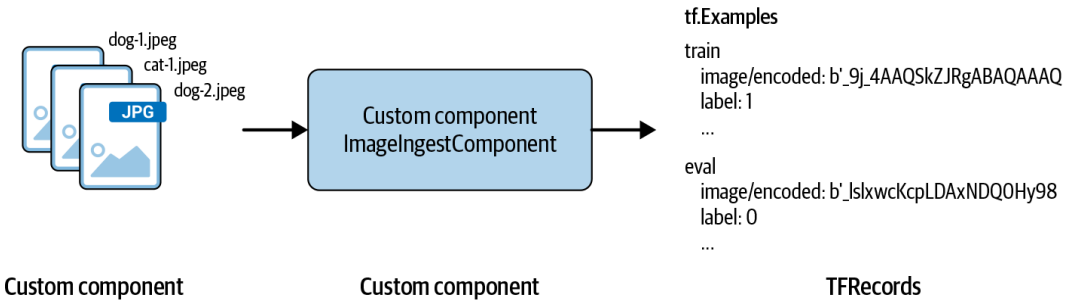

Figure 10-6. Functionality of our demo custom component

Use Cases of Custom Components

Even though we are discussing an ingestion component as an example for a custom component, you aren't limited by the architecture. Your custom component could be applied anywhere along your machine learning pipeline. The concepts discussed in the following sections provide you the highest flexibility to customize your machine learning pipelines to your needs. Some ideas for using custom components are:

- Ingesting data from your custom database
- Sending an email with the generated data statistics to the data science team
- Notifying the DevOps team if a new model was exported
- Kicking off a post-export build process for Docker containers
- Tracking additional information in your machine learning audit trail

We won't describe how to build each of these separately, but if one of these ideas is useful to you, the following sections will provide the knowledge to build your own component.

Writing a Custom Component from Scratch

If we want to write a custom component from scratch, we will need to implement a few component pieces. First, we must define the inputs and outputs of our component as a `ComponentSpec`. Then we can define our component executor, which defines how the input data should be processed and how the output data is generated. If the component requires inputs that aren't registered in the metadata store, we'll need to write a custom component driver. This is the case when, for example, we want to register an image path in the component and the artifact hasn't been registered in the metadata store previously.

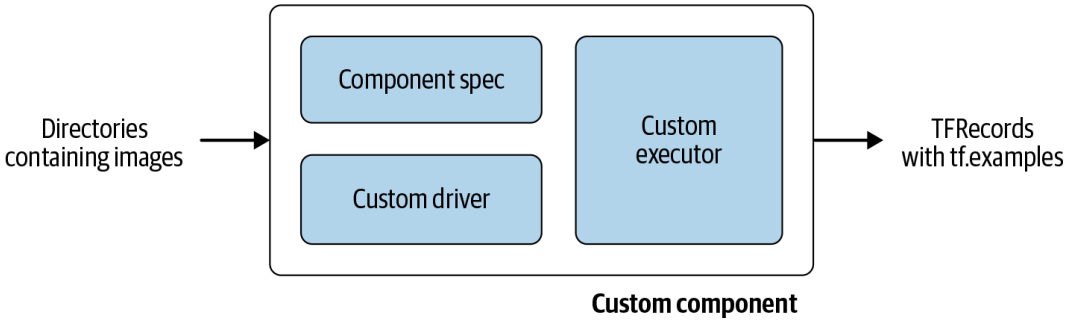

Figure 10-7. Parts of our custom component

The steps in Figure 10-7 might seem complicated, but we will discuss them each in turn in the following sections.

TRY TO REUSE COMPONENTS

If you are thinking about altering an existing TFX component in its functionality, consider reusing existing TFX components and changing the executor instead of starting from scratch, as we will discuss in the section "Reusing Existing Components".

Component specifications

The component specifications, or `ComponentSpec`, define how components communicate with each other. They describe three important details of each component: the component inputs, the component outputs, and potential component parameters that are required during the component execution. Components communicate through *channels*, which are inputs and outputs. These channels are types, as we will see in the following example. The component inputs define the artifacts that the component will receive from previously executed components or new artifacts

like file paths. The component outputs define which artifacts will be registered with the metadata store.

The component parameters define options that are required for execution but aren't available in the metadata store. This could be the `push_destination` in the case of the `Pusher` component or the `train_args` in the `Trainer` component. The following example shows a definition of our component specifications for our image ingestion component:

```
from tfx.types.component_spec import ChannelParameter
from tfx.types.component_spec import ExecutionParameter
from tfx.types import standard_artifacts

class ImageIngestComponentSpec(types.ComponentSpec):
    """ComponentSpec for a Custom TFX Image Ingestion Component."""
    PARAMETERS = {
        'name': ExecutionParameter(type=Text),
    }
    INPUTS = {
        'input': ChannelParameter(type=standard_artifacts.ExternalArtifact), ❶
    }
    OUTPUTS = {
        'examples': ChannelParameter(type=standard_artifacts.Examples), ❷
    }
```

❶ Using `ExternalArtifact` to allow new input paths

❷ Exporting `Examples`

In our example implementation of `ImageIngestComponentSpec`, we are ingesting an input path through the input argument `input`. The generated TFRecord files with the converted images will be stored in the path passed to the downstream components via the `examples` argument. In addition, we are defining a parameter for the component called `name`.

Component channels

In our example `ComponentSpec`, we introduced two types of component channels: `ExternalArtifact` and `Examples`. This is a particular pattern

used for ingestion components since they are usually the first component in a pipeline and no upstream component is available from which we could have received already-processed `Examples`. If you develop a component further down in the pipeline, you might want to ingest `Examples`. Therefore the channel type needs to be `standard_artifacts.Examples`. But we aren't limited to only two types. TFX provides a variety of types. The following shows a small list of available types:

- `ExampleStatistics`
- `Model`
- `ModelBlessing`
- `Bytes`
- `String`
- `Integer`
- `Float`
- `HyperParameters`

With our `ComponentSpec` now set up, let's take a look at the component executor.

Component executors

The component executor defines the processes inside the component, including how the inputs are used to generate the component outputs. Even though we will write this basic component from scratch, we can rely on TFX classes to inherit function patterns. As part of the `Executor` object, TFX will be looking for a function called `Do` for the execution details of our component. We will implement our component functionality in this function:

```
from tfx.components.base import base_executor

class Executor(base_executor.BaseExecutor):
    """Executor for Image Ingestion Component."""

    def Do(self, input_dict: Dict[Text, List[types.Artifact]],
           output_dict: Dict[Text, List[types.Artifact]],
           exec_properties: Dict[Text, Any]) -> None:

        ...
```

The code snippet shows that the `Do` function of our `Executor` expects three arguments: `input_dict`, `output_dict`, and `exec_properties`. These Python dictionaries contain the artifact references that we pass to and from the component as well as the execution properties.

ARTIFACTS CONTAIN REFERENCES

The information provided via the `input_dict` and `output_dict` contain the information stored in the metadata store. These are the references to the artifacts, not the underlying data itself. For example, our `input_dict` dictionary will contain a protocol buffer with the file location information instead of the data. This allows us to process the data efficiently with programs like Apache Beam.

To walk you through a basic implementation of a working `Do` method of the executor, we will reuse the implementation that we discussed in "Image Data for Computer Vision Problems" to convert images to TFRecord data structures. An explanation of the conversion process and details around the TFRecord data structures can be found there. This code should look familiar:

```
def convert_image_to_TFExample(image_filename, tf_writer, input_base_uri):

    image_path = os.path.join(input_base_uri, image_filename) ❶

    lowered_filename = image_path.lower() ❷
    if "dog" in lowered_filename:
        label = 0
    elif "cat" in lowered_filename:
        label = 1
    else:
        raise NotImplementedError("Found unknown image")

    raw_file = tf.io.read_file(image_path) ❸

    example = tf.train.Example(features=tf.train.Features(feature={ ❹
        'image_raw': _bytes_feature(raw_file.numpy()),
        'label': _int64_feature(label)
    }))
    writer.write(example.SerializeToString()) ❺
```

❶ Assemble the complete image path.

❷ Determine the label for each image based on the file path.

❸ Read the image from a disk.

❹ Create the `TensorFlow Example` data structure.

❺ Write the `tf.Example` to TFRecord files.

With the completed generic function of reading an image file and storing it in files containing the TFRecord data structures, we can now focus on custom component-specific code.

We want our very basic component to load our images, convert them to `tf.Examples`, and return two image sets for training and evaluation. For the simplicity of our example, we are hardcoding the number of evaluation examples. In a production-grade component, this parameter should be dynamically set through an execution parameter in the `ComponentSpecs`. The input to our component will be the path to the folder containing all the images. The output of our component will be the path where we'll store the training and evaluation datasets. The path will contain two subdirectories (`train` and `eval`) that contain the TFRecord files:

```
class ImageIngestExecutor(base_executor.BaseExecutor):

    def Do(self, input_dict: Dict[Text, List[types.Artifact]],
           output_dict: Dict[Text, List[types.Artifact]],
           exec_properties: Dict[Text, Any]) -> None:

        self._log_startup(input_dict, output_dict, exec_properties) ❶

        input_base_uri = artifact_utils.get_single_uri(input_dict['input']) ❷
        image_files = tf.io.gfile.listdir(input_base_uri) ❸
        random.shuffle(image_files)
        splits = get_splits(images)

        for split_name, images in splits:
            output_dir = artifact_utils.get_split_uri(
```

```
                output_dict['examples'], split_name) ❹

            tfrecord_filename = os.path.join(output_dir, 'images.tfrecord')
            options = tf.io.TFRecordOptions(compression_type=None)
            writer = tf.io.TFRecordWriter(tfrecord_filename, options=options) ❺
            for image in images:
                convert_image_to_TFExample(image, tf_writer, input_base_uri)
❻
```

❶ Log arguments.

❷ Get the folder path from the artifact.

❸ Obtain all the filenames.

❹ Set the split Uniform Resource Identifier (URI).

❺ Create a TFRecord writer instance with options.

❻ Write an image to a file containing the TFRecord data structures.

Our basic `Do` method receives `input_dict`, `output_dict`, and `exec_properties` as arguments to the method. The first argument contains the artifact references from the metadata store stored as a Python dictionary, the second argument receives the references we want to export from the component, and the last method argument contains additional execution parameters like, in our case, the component name. TFX provides the very useful `artifact_utils` function that lets us process our artifact information. For example, we can use the following code to extract the data input path:

```
artifact_utils.get_single_uri(input_dict['input'])
```

We can also set the name of the output path based on the split name:

```
artifact_utils.get_split_uri(output_dict['examples'], split_name)
```

The last mentioned function brings up a good point. For simplicity of the example, we have ignored the options to dynamically set data splits, as we discussed in Chapter 3. In fact, in our example, we are hardcoding the split names and quantity:

```
def get_splits(images: List, num_eval_samples=1000):
    """ Split the list of image filenames into train/eval lists """
    train_images = images[num_test_samples:]
    eval_images = images[:num_test_samples]
    splits = [('train', train_images), ('eval', eval_images)]
    return splits
```

Such functionality wouldn't be desirable for a component in production, but a full-blown implementation would go beyond the scope of this chapter. In the following section, we will discuss how you can reuse existing component functions and simplify your implementations. Our component in this section will have the same functionality as we discussed in Chapter 3.

Component drivers

If we would run the component with the executor that we have defined so far, we would encounter a TFX error that the input isn't registered with the metadata store and that we need to execute the previous component before running our custom component. But in our case, we don't have an upstream component since we are ingesting the data into our pipeline. The data ingestion step is the start of every pipeline. So what is going on?

As we discussed previously, components in TFX communicate with each other via the metadata store, and the components expect that the input artifacts are already registered in the metadata store. In our case, we want to ingest data from a disk, and we are reading the data for the first time in our pipeline; therefore, the data isn't passed down from a different component and we need to register the data sources in the metadata store.

CUSTOM DRIVERS ARE RARE

It is rare that you need to implement custom drivers. If you can reuse the input/output architecture of an existing TFX component or if the inputs are already registered with the metadata store, you won't need to write a custom driver and you can skip this step.

Similar to our custom executor, we can reuse a `BaseDriver` class provided by TFX to write a custom driver. We need to overwrite the standard behavior of the component, and we can do that by overwriting the `resolve_input_artifacts` method of the `BaseDriver`. A bare-bones driver will register our inputs, which is straightforward. We need to *unpack* the channel to obtain the `input_dict`. By looping over all the values of the `input_dict`, we can access each list of inputs. By looping again over each list, we can obtain each input and then register it at the metadata store by passing it to the function `publish_artifacts`. `publish_artifacts` will call the metadata store, publish the artifact, and set the state of the artifact as ready to be published:

```
class ImageIngestDriver(base_driver.BaseDriver):
  """Custom driver for ImageIngest."""

  def resolve_input_artifacts(
      self,
      input_channels: Dict[Text, types.Channel],
      exec_properties: Dict[Text, Any],
      driver_args: data_types.DriverArgs,
      pipeline_info: data_types.PipelineInfo) -> Dict[Text, List[types.Artifact]
    """Overrides BaseDriver.resolve_input_artifacts()."""
    del driver_args ❶
    del pipeline_info

    input_dict = channel_utils.unwrap_channel_dict(input_channels) ❷
    for input_list in input_dict.values():
        for single_input in input_list:
            self._metadata_handler.publish_artifacts([single_input]) ❸
            absl.logging.debug("Registered input: {}".format(single_input))
            absl.logging.debug("single_input.mlmd_artifact "
                               "{}".format(single_input.mlmd_artifact)) ❹
    return input_dict
```

❶ Delete unused arguments.

❷ Unwrap channel to obtain the input dictionary.

❸ Publish the artifact.

❹ Print artifact information.

While we loop over each input, we can print additional information:

```
print("Registered new input: {}".format(single_input))
print("Artifact URI: {}".format(single_input.uri))
print("MLMD Artifact Info: {}".format(single_input.mlmd_artifact))
```

With the custom driver now in place, we need to assemble our component.

Assembling the custom component

With our `ImageIngestComponentSpec` defined, the `ImageIngestExecutor` completed, and the `ImageIngestDriver` set up, let's tie it all together in our `ImageIngestComponent`. We could then, for example, load the component in a pipeline that trains image classification models.

To define the actual component, we need to define the specification, executor, and driver classes. We can do this by setting `SPEC_CLASS`, `EXECUTOR_SPEC`, and `DRIVER_CLASS`, as shown in the following example code. As the final step, we need to instantiate our `ComponentSpecs` with the component's arguments (e.g., input and output examples, and the provided name) and pass it to the instantiated `ImageIngestComponent`.

In the unlikely case that we don't provide an output artifact, we can set our default output artifact to be of type `tf.Example`, define our hard-coded split names, and set it up as a channel:

```
from tfx.components.base import base_component
from tfx import types
```

```
from tfx.types import channel_utils

class ImageIngestComponent(base_component.BaseComponent):
    """Custom ImageIngestWorld Component."""
    SPEC_CLASS = ImageIngestComponentSpec
    EXECUTOR_SPEC = executor_spec.ExecutorClassSpec(ImageIngestExecutor)
    DRIVER_CLASS = ImageIngestDriver

    def __init__(self, input, output_data=None, name=None):
        if not output_data:
            examples_artifact = standard_artifacts.Examples()
            examples_artifact.split_names = \
                artifact_utils.encode_split_names(['train', 'eval'])

            output_data = channel_utils.as_channel([examples_artifact])

        spec = ImageIngestComponentSpec(input=input,
                                        examples=output_data,
                                        name=name)
        super(ImageIngestComponent, self).__init__(spec=spec)
```

By assembling our `ImageIngestComponent`, we have tied together the individual pieces of our basic custom component. In the next section, we will take a look at how we can execute our basic component.

Using our basic custom component

After implementing the entire basic component to ingest images and turn them into TFRecord files, we can use the component like any other component in our pipeline. The following code example shows how. Notice that it looks exactly like the setup of other ingestion components that we discussed in Chapter 3. The only difference is that we need to import our newly created component and then run the initialized component:

```
import os

from tfx.utils.dsl_utils import external_input
from tfx.orchestration.experimental.interactive.interactive_context import \
    InteractiveContext

from image_ingestion_component.component import ImageIngestComponent
```

```
context = InteractiveContext()

image_file_path = "/path/to/files"
examples = external_input(dataimage_file_path_root)
example_gen = ImageIngestComponent(input=examples,
                                   name=u'ImageIngestComponent')
context.run(example_gen)
```

The output from the component can then be consumed by downstream components like `StatisticsGen`:

```
from tfx.components import StatisticsGen

statistics_gen = StatisticsGen(examples=example_gen.outputs['examples'])
context.run(statistics_gen)

context.show(statistics_gen.outputs['statistics'])
```

VERY BASIC IMPLEMENTATION

We want to caution you that the discussed implementation only provides basic functionality and is not production ready. For details of the missing functionality, please see the following section. For a product-ready implementation, please see our updated component implementation in the next sections.

Implementation review

In the previous sections, we walked through a basic component implementation. While the component is functioning, it is missing some key functionality that we discussed in Chapter 3 (e.g., dynamic split names or split ratios)—and we would expect such functionality from our ingestion component. The basic implementation also required a lot of boiler-plate code (e.g., the setup of the component driver). The ingestion of the images should be handled efficiently and in a scalable way. We can achieve such efficient data ingestion through the Apache Beam usage under the hood of TFX components.

In the next section, we will discuss how we could simplify the implementations and adopt the patterns we discussed in Chapter 3—for example,

ingesting data from Presto databases. By reusing common functionality, such as the component drivers, we can speed up implementation and reduce code bugs.

Reusing Existing Components

Instead of writing a component for TFX entirely from scratch, we can inherit an existing component and customize it by overwriting the executor functionality. As shown in Figure 10-8, this is generally the preferred approach when a component is reusing an existing component architecture. In the case of our demo component, the architecture is equivalent with a file base ingestion component (e.g., `CsvExampleGen`). Such components receive a directory path as a component input, load the data from the provided directory, turn the data into `tf.Examples`, and return the data structures in TFRecord files as output from the component.

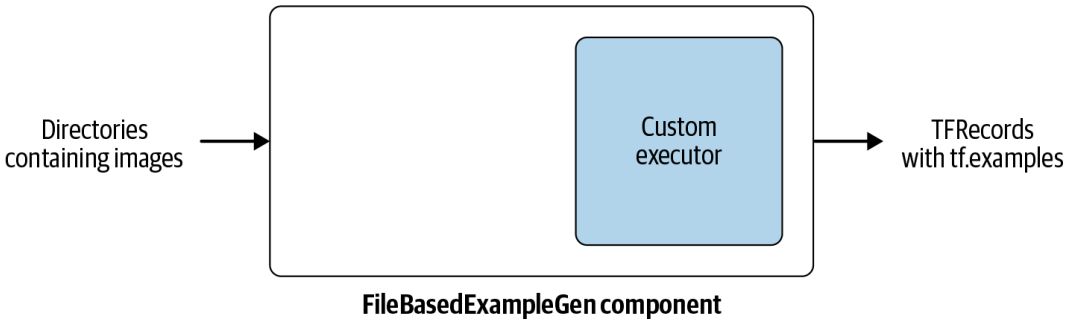

Figure 10-8. Extending existing components

As we discussed in Chapter 3, TFX provides the `FileBasedExampleGen` for this purpose. Since we are going to reuse an existing component, similar to our Avro and Parquet examples, we can simply focus on developing our custom executor and making it more flexible as our previous basic component. By reusing existing code infrastructure, we can also piggyback on existing Apache Beam implementations.

By reusing an existing component architecture for ingesting data into our pipelines, we can also reuse setups to ingest data efficiently with Apache Beam. TFX and Apache Beam provide classes (e.g., `GetInputSourceToExamplePTransform`) and function decorators (e.g., `@beam.ptransform_fn`) to ingest the data via Apache Beam pipelines. In our example, we use the function decorator `@beam.ptransform_fn`, which allows us to define Apache Beam transformation (`PTransform`).

The decorator accepts an Apache Beam pipeline, runs a given transformation (e.g., in our case, the loading of the images and their conversion to `tf.Examples`), and returns the Apache Beam `PCollection` with the transformation results.

The conversion functionality is handled by a function very similar to our previous implementation. The updated conversion implementation has one major difference: we don't need to instantiate and use a TFRecord writer; instead, we can fully focus on loading images and converting them to `tf.Examples`. We don't need to implement any functions to write the `tf.Examples` to TFRecord data structures because we did it in our previous implementation. Instead, we return the generated `tf.Examples` and let the underlying TFX/Apache Beam code handle the writing of the TFRecord files. The following code example shows the updated conversion function:

```
def convert_image_to_TFExample(image_path)): ❶

    # Determine the label for each image based on the file path.
    lowered_filename = image_path.lower()
    print(lowered_filename)
    if "dog" in lowered_filename:
        label = 0
    elif "cat" in lowered_filename:
        label = 1
    else:
        raise NotImplementedError("Found unknown image")

    # Read the image.
    raw_file = tf.io.read_file(image_path)

    # Create the TensorFlow Example data structure.
    example = tf.train.Example(features=tf.train.Features(feature={
        'image_raw': _bytes_feature(raw_file.numpy()),
        'label': _int64_feature(label)
    }))
    return example ❷
```

❶ Only the file path is needed.

❷ The function returns examples instead of writing them to a disk.

With the updated conversion function in place, we can now focus on implementing the core executor functionality. Since we are customizing an existing component architecture, we can use the same arguments as we discussed in Chapter 3, such as split patterns. Our `image_to_example` function in the following code example takes four input arguments: an Apache Beam pipeline object, an `input_dict` with artifact information, a dictionary with execution properties, and split patterns for ingestion. In the function, we generate a list of available files in the given directories and pass the list of images to an Apache Beam pipeline to convert each image found in the ingestion directories to `tf.Examples`:

```
@beam.ptransform_fn
def image_to_example(
    pipeline: beam.Pipeline,
    input_dict: Dict[Text, List[types.Artifact]],
    exec_properties: Dict[Text, Any],
    split_pattern: Text) -> beam.pvalue.PCollection:

    input_base_uri = artifact_utils.get_single_uri(input_dict['input'])
    image_pattern = os.path.join(input_base_uri, split_pattern)
    absl.logging.info(
        "Processing input image data {} "
        "to tf.Example.".format(image_pattern))

    image_files = tf.io.gfile.glob(image_pattern) ❶
    if not image_files:
        raise RuntimeError(
            "Split pattern {} did not match any valid path."
            "".format(image_pattern))

    p_collection = (
        pipeline
        | beam.Create(image_files) ❷
        | 'ConvertImagesToTFRecords' >> beam.Map(
            lambda image: convert_image_to_TFExample(image)) ❸
    )
    return p_collection
```

❶ Generate a list of files present in the ingestion paths.

❷ Convert the list to a Beam `PCollection`.

❸ Apply the conversion to every image.

The final step in our custom executor is to overwrite the `GetInputSourceToExamplePTransform` of the `BaseExampleGenExecutor` with our `image_to_example`:

```
class ImageExampleGenExecutor(BaseExampleGenExecutor):

    @beam.ptransform_fn
    def image_to_example(...):
        ...

    def GetInputSourceToExamplePTransform(self) -> beam.PTransform:
        return image_to_example
```

Our custom image ingestion component is now complete!

Using our custom executor

Since we are reusing an ingestion component and swapping out the processing executor, we can now follow the same patterns we discussed for the Avro ingestion in Chapter 3 and specify a `custom_executor_spec`. By reusing the `FileBasedExampleGen` component and overwriting the `executor`, we can use the entire functionality of ingestion components that we discussed in Chapter 3, like defining the input split patterns or the output train/eval splits. The following code snippet gives a complete example of using our custom component:

```
from tfx.components import FileBasedExampleGen
from tfx.utils.dsl_utils import external_input

from image_ingestion_component.executor import ImageExampleGenExecutor

input_config = example_gen_pb2.Input(splits=[
    example_gen_pb2.Input.Split(name='images',
                                pattern='sub-directory/if/needed/*.jpg'),
])

output = example_gen_pb2.Output(
    split_config=example_gen_pb2.SplitConfig(splits=[
        example_gen_pb2.SplitConfig.Split(
```

```
                name='train', hash_buckets=4),
            example_gen_pb2.SplitConfig.Split(
                name='eval', hash_buckets=1)
        ])
)

example_gen = FileBasedExampleGen(
    input=external_input("/path/to/images/"),
    input_config=input_config,
    output_config=output,
    custom_executor_spec=executor_spec.ExecutorClassSpec(
        ImageExampleGenExecutor)
)
```

As we have discussed in this section, extending the component executor will always be a simpler and faster implementation than writing a custom component from scratch. Therefore, we recommend this process if you are able to reuse existing component architectures.

Summary

In the chapter, we expanded on the TFX concepts from previous chapters. We discussed how to write custom components in detail. Writing custom components gives us the flexibility to extend existing TFX components and tailor them for our pipeline needs. Custom components allow us to integrate more steps into our machine learning pipelines. By adding more components to our pipeline, we can guarantee that all models produced by the pipeline have gone through the same steps. Since the implementation of custom components can be complex, we reviewed a basic implementation of a component from scratch and highlighted an implementation of a new component executor by inheriting existing component functionality.

We also discussed advanced settings for a training setup, such as branching pipeline graphs to produce multiple models from the same pipeline execution. This functionality can be used to produce TFLite models for deployments in mobile apps. We also discussed warm starting the training process to continuously train machine learning models. Warm starting model training is a great way of shortening the training steps for continuously trained models.

We introduced the concept of having a human in the loop in a machine learning pipeline setup and also discussed how the experimental component can be implemented. The human in the loop concept is a way of adding an expert review as a required pipeline step before deploying models. We believe that the combination of fully automated components and a few, critical reviews by data scientists will support the adoption of machine learning pipelines.

In the next two chapters, we will take a look at how to run our TFX pipeline in the orchestration environment of your choice.