O'REILLY®

Building Machine Learning Pipelines

Automating Model Life Cycles with TensorFlow

Early Release

RAW & UNEDITED

Hannes Hapke

Building Machine Learning Pipelines

Automating Model Life Cycles with TensorFlow

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Hannes Hapke and Catherine Nelson

Building Machine Learning Pipelines

by Hannes Hapke & Catherine Nelson

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Development Editor: Nicole Taché

Acquisitions Editor: Jonathan Hassell

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

January 2020: First Edition

Revision History for the Early Release

- 2020-10-01: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492045441 for release details.

Chapter 1. Introduction

Machine learning and in particular deep learning has emerged as a technology to tackle complex problems such as understanding video feeds in self-driving cars or personalizing medications. Researchers and machine learning engineers have paid a great deal of attention to the model architectures and concepts. With machine learning models being used in more applications and software tools, machine learning requires the same standardization of processes the software industry went through in the last decade. This book will introduce pipelines for machine learning projects and demonstrate them on an end-to-end project.

In this chapter, we will outline the steps that go into building machine learning pipelines. We will focus on why proper pipelines are critical for successful data science projects and back up the reasons with business examples. We will lay the groundwork for the full book by introducing the individual chapters. Throughout the book we'll use an example project to demonstrate the principles we describe. At the end of this

chapter, we will introduce our example project, its underlying dataset and its implementation.

What are Machine Learning Pipelines?

During the last few years, the developments in the field of machine learning have been astonishing. With the broad availability of Graphical Processing Units (GPUs) and the developments of new deep learning concepts like Transformers (e.g., Bert), or Generative Adversarial Networks (e.g., DCGANs), the number of AI projects has skyrocketed. The number of AI startups is endless and corporations are applying the latest machine learning concepts to their business problems. In this rush for the most performant machine learning solution, we observed that data scientists and machine learning engineers are lacking good sources of information for concepts and tools to accelerate, reuse, manage and deploy their developments. What is needed is the standardization of machine learning pipelines.

Our intention with this book is to contribute to the standardization of machine learning projects by walking the readers through an entire machine learning pipelines, end-to-end.

Machine learning pipelines are processes to accelerate, reuse, manage and deploy machine learning models. Software engineering went through the same changes a decade or so ago with the introduction of Continuous Integration (CI) and Continuous Deployment (CD). Back in the day, it was a lengthy process to test and deploy a web app. These days, these processes have been greatly simplified by a few tools and concepts. While the deployment of web apps required the collaboration between a DevOps engineer and the software developer, today, the app can be tested and deployed reliably in a matter of minutes. In terms of workflows, data scientists and machine learning engineers can learn a lot from software engineering.

From our personal experience, most data science projects do not have the luxury of a large team including multiple data scientists and machine learning engineers to deploy models. This makes it difficult to build an entire pipeline in-house from scratch. It may mean that machine learning projects turn into one-off efforts where performance degrades after time, the data scientist spends much of their time fixing errors when the underlying data changes, or the model is not used widely. Therefore we are outlining processes to:

- Version your data effectively and kick off a new model training run

- Efficiently pre-process data for your model training and validation
- Version control your model checkpoints during training
- Track your model training experiments
- Analyze and validate the trained and tuned models
- Deploy the validated model
- Scale the deployed model
- Capture new training data and model performance metrics with feedback loops

The list left out one important point: the training and tuning of the model. We assume that you already have a good working knowledge of that step. If you are getting started with machine or deep learning, these O'Reilly publications are a great starting point to familiarize yourself with machine learning:

- Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms 1st Edition by Nikhil Buduma, Nicholas Locascio
- Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems 1st Edition by Aurélien Géron

Overview of Machine Learning Pipelines

A machine learning pipeline starts with the collection of new training data and ends with receiving some kind of feedback on how your newly trained model is performing. This feedback can be a production performance metric, or feedback from users of your product. The pipeline includes a variety of steps including data pre-processing, model training and model analysis as well as the deployment of the model. You can imagine that stepping through these steps manually is cumbersome and very error-prone. In the course of this book, we will introduce tools and solutions to automate your model life cycle.

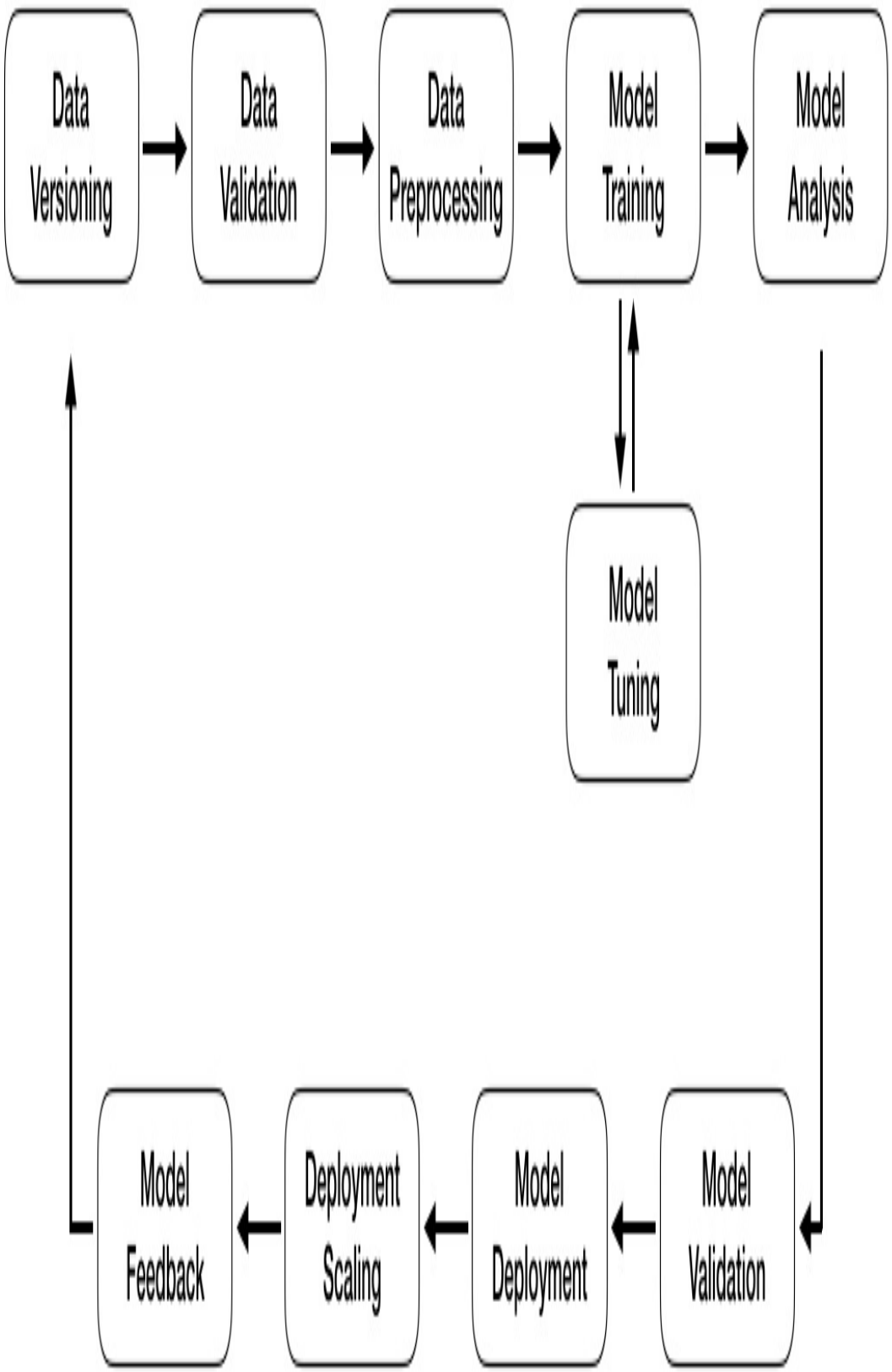

Data Versioning
Data Validation
Data Preprocessing
Model Training
Model Analysis
Model Tuning
Model Validation
Model Deployment
Deployment Scaling
Model Feedback

Figure 1-1. Model Life Cycle

As you can see in Figure 1-1, the pipeline is actually a recurring cycle. Data can be continuously collected and therefore machine learning models can be updated. More data generally means improved models [1] Automation is then key. In real-world applications, you want to re-train your models frequently. If this is a manual process, where it is necessary to manually validate the new training data or analyze the updated models, a data scientist or machine learning engineer would have no time to develop new models for entirely different business problems.

A model life cycle commonly includes:

Experiment Tracking

All the operations in the model life cycle need to be tracked to allow automation. Experiment tracking is often overlooked in machine learning pipelines, but this step offers great returns for very little investment. When data scientists optimize machine learning models, they evaluate various model types, model architectures, hyperparameters and data sets. We have seen data science teams store their training results in physical scrapbooks. Now imagine a data scientist wants to build onto

previous work of a colleague. With the learning experience captured in physical notebooks, transferring knowledge within teams will be cumbersome.

Whether you optimize your models manually or you tune the models automatically, capturing and sharing the results of your optimization process is essential. Team members can quickly evaluate the progress of the model updates. At the same time, the author of the models receives automated records of the performed experiments. The tools we will introduce will automate the tracking process, so there is no need for manual result tracking.

In the machine learning world, experiment tracking will be a safeguard against potential litigations. If a data science team is facing the question of whether an edge case was considered while training the model, the experiment tracking can assist in tracing the model parameters and iterations.

Data Versioning

Data versioning is the beginning of the model life cycle. When a new cycle is kicked off, for example when new training data is available, a snapshot of the data will be version controlled and it can kick off a new cycle. This step is comparable to

version control in software engineering, except we don’t check in the software code, but the model training and validation data.

Data Validation

Before training a new model version, we need to validate the new data. Data validation focusses on checking the statistics of the new data and alerting the data scientist if any abnormalities are detected. For example, if you are training a binary classification model, your training data could contain 50% of class A samples and 50% of class B samples. Data validation tools provide alerts if the split between those classes changes, where perhaps the newly collected data is split 70/30 between the two classes. If a model is being trained with such a biased training set and the data scientist hasn’t adjusted the model’s loss function, or over/under sampled category A or B, the model would be biased towards the dominant category.

Common data validation tools will also allow you to compare different datasets. Let’s say you have a data set with a dominant label and you split the data set into a training and validation set, you need to make sure that the label split is roughly the same between the two data sets. Data validation tools will allow you to compare data sets and highlight

abnormalities.

If the validation highlights anything out of the ordinary to light, the life cycle can be stopped here and the data scientist can be alerted. If a shift in the data is detected, the data scientist or the machine learning engineer can either change the sampling of the individual label samples (e.g. only pick the same number of label samples) or change the model's loss function and kick off a new model build pipeline and restart the life cycle.

Data Preprocessing

It is highly likely that you can not use your freshly collected data and train your machine learning model directly. In almost all cases, you will need to preprocess the data to use it for your training runs. Labels often need to be converted to one or multi-hot vectors. The same applies to the model inputs. If you train a model from text data, you want to convert the characters of the text to indices or the text tokens to word vectors. Since the preprocessing is only required prior to the model training and not with every training epoch, it makes the most sense to run the preprocessing in its own life cycle step before training the model.

A variety of tools have been developed to process data efficiently and fast. The list of possible solutions is endless and can range from a simple Python script to elaborate graph models. While most data scientists focus on the processing capabilities of the preferred tools, it is also important that modifications of the preprocessing steps can be linked to the processed data and vice versa. That means if someone modifies a processing step (e.g. allowing an additional label in a one-hot vector conversion), the previous training data should become invalid and force an update of the entire pipeline.

Model Training and Tuning

The model training step is the core of the machine learning pipeline. In this step, we train a model to take inputs and predict an output with the lowest error possible. With larger models and especially with large training sets, this step can quickly become difficult to manage. Since memory is generally a finite resource for our computations, the efficient distribution of the model training is crucial.

Model tuning has seen a great deal of attention lately because it can yield significant performance improvements and can provide a competitive edge. In the earlier step of the model training, we assumed that we would do one training run. But

how could we pick the most optimal model architecture or hyperparameters in only one run? Impossible! That is where model tuning comes in. With today's DevOps tools, we can replicate machine models and their training setups effortless. This gives us the opportunity to spin up a large number of models in parallel or in a sequence (depending on the optimization method) and train the model in all the different configurations we would like to tune the model for.

During the model tuning step, the training of the machine learning models with different hyperparameters, e.g. the model's learning rate, or the number of network layers, can be automated. The tuning tool will pick a set of parameters from a list of parameter suggestions. The choice of the parameter values can either be based on a grid search where we would sweep over all combinations of parameters or based on more probabilistic approaches where we can try to estimate the best, next set of parameters to train the model with. Tuning tools will set up the model training runs, similar to the training runs we have performed in the earlier training step. The tool will just allow us to perform the training at a larger scale and fully automated. At the same time, every training run will report all training parameters and its evaluation metrics back to the experiment tracking tool, so that we can review the model performance holistically.

Model Analysis

Once we have determined the most optimal set of model parameters, which grants the highest accuracy or the lowest loss, we need to analysis its performance before deploying the model to our production environment.

Model analysis has gained a lot of attention in the past year or two, and rightfully tough. It is critically important to validate models for production use against biases. During these steps, we are validating the model against an unseen *analysis dataset*, which shouldn't be a subset of the previously used training and validation set. During the model analysis, we'll expose the model to small variations of the analysis dataset and measure how sensitive the model's predictions are against the small variations. At the same time, analysis tools measure if the model predicts dominantly for one label for a subset section of the dataset. A key reason in favor of a proper model analysis is that bias against a subsection of the data might get lost in the validation process while training the model. During the training, the model accuracy against the validation set is often calculated as an average over the entire data set which will make it hard to discover a bias.

Similar to the model tuning step and the final selection of the best performing model, this workflow step requires a review

by a data scientist. However, we will demonstrate how the entire analysis can be automated and only the final review is done by a human. The automation will keep the analysis of the models consistent and comparable against other analyzes.

Model Versioning

The purpose of the model versioning and validation step is to keep track of which model, set of hyperparameters and data sets have been selected as the next version of the model.

Semantic versioning in software engineering requires you to increase the major version number when you make an incompatible change in your API, otherwise, you would increase the minor version number. Model release management has another degree of freedom: the dataset. There are situations, where you can achieve a significantly different model performance without changing a single model parameter or architecture description, but by providing significantly more data for the training process. Well, does that performance increase warrant a major version upgrade?

While the answer to this question might be different for every data science team, it is essential to document all inputs to a new model version (hyperparameters, data sets, architecture)

and track them as part of this release step.

Model Deployment

Once you have trained, tuned and analyzed your model, it is ready for prime time. Unfortunately, too many models are being deployed with one-off implementations, which makes updating models a brittle process.

Some model servers have been open-sourced in the last few years which allow efficient deployments. Modern model servers allow you to deploy your models without writing web app code. Often they provide you with multiple API interfaces like Representational State Transfer (REST) or Remote Procedure Call (RPC) protocols and allow you to host multiple versions of the same model simultaneously. Hosting multiple versions are the same time will allow you to run A/B tests on your models and provide valuable feedback about your model improvements.

Model servers also allow you to update the model version without redeploying your application, which will reduce your application’s downtime and reduce the communication between the application development and the machine learning teams.

Feedback Loops

The last step of the machine learning life cycle is often forgotten, but it is crucial to the success of data science projects. We need to close the loop and measure the effectiveness and performance of the newly deployed model.

During this step, we can capture valuable information about the performance of the model and capture new training data to increase our data sets to update our model and create a new version.

With the capturing of the data, we can close the loop of the life cycle. Besides the two manual review steps, we can automate the entire life cycle. Data scientists should be able to focus on the development of new models, not on updating and maintaining existing models.

Data Privacy

At the time of writing, data privacy considerations sit outside the standard model life cycle. We expect this to change in the future as consumer concerns grow over the use of their data and new laws are brought in to restrict the usage of personal data. This will lead to the integration of privacy-preserving methods for machine learning into tools for building pipelines.

Two current options for increasing privacy in machine learning models are discussed in this book: differential privacy, which provides a mathematical guarantee that model predictions do not expose a user's data, and federated learning, where the raw data does not leave a user's device.

Why Machine Learning Pipelines?

The key benefit of machine learning pipelines lies in the automation of the model life cycle steps. When new training data becomes available, a workflow which includes the data validation, preprocessing, model tuning, analysis and deployment should be triggered. We observed too many data science teams manually going through these steps, which is costly and also a source for errors.

FOCUS ON NEW MODELS, NOT MAINTAINING EXISTING MODELS

Machine learning pipelines will free up data scientists from maintaining existing models. We have observed too many data scientists spending their days on keeping previously developed models up-to-date. They run scripts manually to preprocess their training data, they write one-off deployment scripts or manually tune their models. The time spent on these activities

could be used to develop completely new models. The data scientists could solve or automate more business problems instead of keeping existing solutions up-to-date.

MANUAL PIPELINES DON'T SCALE

Imagine a data scientist takes six weeks to develop a model and is then asked to update the newly developed model every two weeks. If it takes the data scientist one or two days to update the model, soon she/he will be completely busy just maintaining previous work rather than engaging in new challenging work. Automated pipelines allow the data scientists to develop new models, the fun part of their job. Ultimately, this will lead to higher job satisfaction and retention in a competitive job market.

AUTOMATED PIPELINES PREVENT BUGS

Automated pipelines can prevent bugs. As we will see in the later chapters, newly created models will be tied to a set of versioned data and the preprocessing steps will be tied to the developed model. That means that if new data is collected, a new model will be generated. If the preprocessing steps are updated, the training data will become invalid and a new model will be generated. In manual machine learning workflows, a common source for bugs is a change in the

preprocessing step after a model was trained. In that case, we would deploy a model with different processing instructions than we trained the model with. These bugs might be really difficult to debug since an inference of the model is still possible, but simply incorrect. With automated workflows, these errors can be prevented.

USEFUL PAPER TRAIL

The experiment tracking and the model release management generates a paper trail of the model changes. The experiment tracking will keep track of the changes to the model's hyperparameters, the used data sets and the resulting model metrics (e.g. loss or accuracy). The model release management will keep track of which model was ultimately selected and deployed. This paper trail is especially valuable if the data science team needs to recreate a model or track the model performance.

The Business Case for Automated Machine Learning Pipelines

The implementation of automated machine learning pipelines will lead to two impacts for a data science team:

- Faster development time for new models
- Simpler processes to update existing models

Both aspects will reduce the costs of data science projects. But furthermore, automated machine learning pipelines will:

- Help detect potential biases in the data sets or in the trained models. Spotting biases can prevent harm to people who interact with the model, for example Amazon's machine learning powered resume screener that was found to be biased against women [2].
- The paper trail created by the experiment tracking and the model release management will assist if questions around the General Data Protection Regulation (GDPR) compliance arise.
- The automation of the model updates will free up development time for data scientists and increase their job satisfaction.

Overview of the Chapters

In the subsequent chapters, we will introduce specific steps for building machine learning pipelines and demonstrate how these work with an example project.

Chapter 2: Pipeline Orchestration provides an overview of how to orchestrate your machine learkning pipelines. We are introducing three common tools for the orchestration of the pipeline tasks: Apache Beam, Apache Airflow and Kubeflow Pipelines.

Chapter 3: TensorFlow Extended introduces the Tensorflow Extended (TFX) ecosystem, explains how tasks communicate with each other, and how TFX components works. We are also addressing how to write your own custom components.

Chapter 4: Data Validation explains how the data that flows into your pipeline can be validated efficiently, using the TensorFlow Data Validation. This will alert you if the new data changes substantially from previous data in a way that may affect your model's performance.

Chapter 5: Data Preprocessing focusses on the preprocessing of the data (the feature engineering), using TensorFlow Transform to convert the data from raw data to features suitable for training a machine learning model.

Chapter 6: Model Analysis will introduce useful metrics for understanding your model in production, including those that may allow you to uncover biases in the model's predictions. We will also give an overview of tools for interpreting and

explaining the predictions.

Chapter 7: Model Validation shows how to control the versioning of your model when a new version improves on one of the metrics from the previous chapter, using the Model Validator component of TensorFlow Extended. The model in the pipeline can be automatically updated to the new version.

Chapter 8: Model Deployment focusses on how to deploy your machine learning model efficiently. Starting off with a simple *Flask* implementation, we will highlight the limitations of such custom model applications. We will introduce *TensorFlow Serving* and highlight its batching functionality. As an alternative to TensorFlow Serving, we will introdue *Seldon*, an open-source solution, which allows the deployment of any machine learning framework.

Chapter 9: Model Deployment to Web Broswers and Edge Devices discusses how to deploy your trained model in situations away from a traditional server-based setup. We will give examples of deploying to browsers using TensorFlow.js and to mobile and other devices using TFLite.

Chapter 10: Feedback Loops discusses how to turn your model pipeline into a cycle that can be improved by feedback from users of the final product. We'll discuss what type of data

to capture to improve the model for future versions, and how to feed that data back into the pipeline.

Chapter 11: Example Pipeline with Apache AirFlow brings together the entire pipline for our example project and shows the end-to-end pipeline.

Chapter 12: Example Pipeline with KubeFlow Pipelines discusses how to use KubeFlow to orchestrate and scale our example project.

Chapter 13: Data Privacy for Machine Learning introduces the rapidly changing field of privacy-preserving machine learning and discusses two important methods for this: differential privacy and federated learning.

In the final Chapter 14, we will outline some new and upcoming machine learning technologies that we expect to improve the process of building machine learning pipelines in the future.

Our Example Project

To follow along with the chapters, we have created an example project using open source data. The dataset is a collection of

consumer complaints about financial products in the US, and it contains a mixture of structured data (categorical/numeric data) and unstructured data (text).

The machine learning problem is: given data about the complaint, predict whether the complaint was disputed by the consumer. In this dataset, 16% of complaints are disputed, so the dataset is not balanced.

The source code of our demo project can be found in our GitHub repository at link:https://github.com/inc0/ml-projects-book

Downloading the Dataset

The dataset can be found here:https://catalog.data.gov/dataset/consumer-complaint-database or here:https://www.kaggle.com/sebastienverpile/consumercomplaintsdata

For easier access to the data sets, we have provided a download script. Once you have cloned the git repository, you can retrieve the first data set by executing

Example 1-1.

```
$ sh src/ch01_introduction/download_data.sh
```

Our Machine Learning Model

The core of our example deep learning example is the model generated by the function `get_model`. The model features are the US state, company, type of financial product, whether there was a timely response to the complaint and how the complaint was submitted.

Our demo model consists then of five layers. The first layer is our embedding layer which converts the categorical data into a vector representation. The second layer is a one-dimensional convolution layer. After the convolutional layer, we apply a global max pooling layer to the convolutional filters from the previous layer. The information then is passed to three fully connected layers which provide a softmax activation as the output layer for the categorical classification.

Example 1-2.

```
def get_model(show_summary=True, max_len=64,
vocab_size=10000, embedding_dim=100):
    input_state = tf.keras.layers.Input(shape=(1,),
name="state_xf")
    input_company = tf.keras.layers.Input(shape=
(1,), name="company_xf")
    input_product = tf.keras.layers.Input(shape=
```

```
(1,), name="product_xf")
    input_timely_response = tf.keras.layers.Input(
        shape=(1,), name="timely_response_xf")
    input_submitted_via = tf.keras.layers.Input(
        shape=(1,), name="submitted_via_xf")

    x_state = tf.keras.layers.Embedding(60, 5)
(input_state)
    x_state = tf.keras.layers.Reshape((5, ),
input_shape=(1, 5))(x_state)

    x_company = tf.keras.layers.Embedding(2500, 20)
(input_company)
    x_company = tf.keras.layers.Reshape((20, ),
input_shape=(1, 20))(x_company)

    x_company = tf.keras.layers.Embedding(2, 2)
(input_product)
    x_company = tf.keras.layers.Reshape((2, ),
input_shape=(1, 2))(x_company)

    x_timely_response = tf.keras.layers.Embedding(2,
2)(input_timely_response)
    x_timely_response = tf.keras.layers.Reshape((2,
), input_shape=(1, 2))(x_timely_response)

    x_submitted_via = tf.keras.layers.Embedding(10,
3)(input_submitted_via)
    x_submitted_via = tf.keras.layers.Reshape((3, ),
input_shape=(1, 3))(x_submitted_via)

    conv_input = tf.keras.layers.Input(shape=
(max_len, ), name="Issue_xf")
    conv_x = tf.keras.layers.Embedding(vocab_size,
```

```
embedding_dim)(conv_input)
    conv_x = tf.keras.layers.Conv1D(128, 5,
activation='relu')(conv_x)
    conv_x = tf.keras.layers.GlobalMaxPooling1D()
(conv_x)
    conv_x = tf.keras.layers.Dense(10,
activation='relu')(conv_x)

    x_feed_forward = tf.keras.layers.concatenate(
        [x_state, x_company, x_company,
x_timely_response, x_submitted_via, conv_x])
    x = tf.keras.layers.Dense(100,
activation='relu')(x_feed_forward)
    x = tf.keras.layers.Dense(50, activation='relu')
(x)
    x = tf.keras.layers.Dense(10, activation='relu')
(x)
    output = tf.keras.layers.Dense(
        1, activation='sigmoid',
name='Consumer_disputed_xf')(x)
    inputs = [
        input_state, input_company, input_product,
        input_timely_response, input_submitted_via,
conv_input]
    tf_model = tf.keras.models.Model(inputs, output)
    tf_model.compile(optimizer='rmsprop',
                     loss='binary_crossentropy',
                     metrics=['accuracy'])
    if show_summary:
        tf_model.summary()
    return tf_model
```

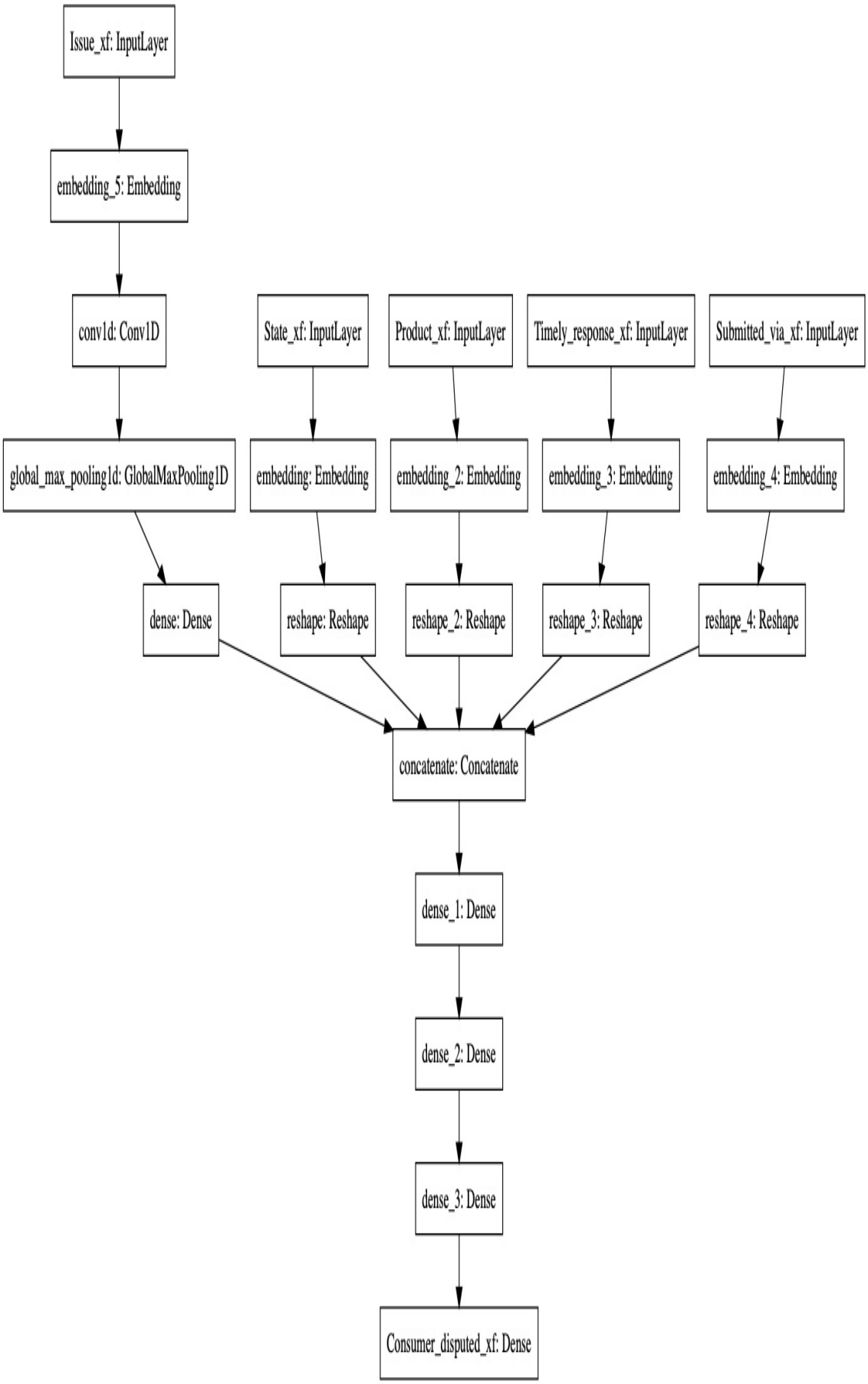

Issue_xf: InputLayer
embedding_5: Embedding
conv1d: Conv1D
global_max_pooling1d: GlobalMaxPooling1D
dense: Dense
State_xf: InputLayer
embedding: Embedding
reshape: Reshape
Product_xf: InputLayer
embedding_2: Embedding
reshape_2: Reshape
Timely_response_xf: InputLayer
embedding_3: Embedding
reshape_3: Reshape
Submitted_via_xf: InputLayer
embedding_4: Embedding
reshape_4: Reshape
concatenate: Concatenate
dense_1: Dense
dense_2: Dense
dense_3: Dense
Consumer_disputed_xf: Dense

Figure 1-2. Keras model

Over the course of the chapters, we won’t modify the core model. What we’ll update are the preprocessing steps, the model export steps, etc. but never the model itself.

If you have any question regarding the model itself, feel free to connect with us via Github.

Who is this book for?

The primary audience for the book is data scientists and machine learning engineers who want to go beyond training a one-off machine learning model and who want to successfully productize their data science projects. You should be comfortable with the basic machine learning concepts and familiar with at least one machine learning framework (e.g. PyTorch, TensorFlow, Keras). The examples in this book are based on TF and Keras, but the core concepts can be applied to any framework.

A secondary audience for this book is managers of data science projects, software developers or DevOps engineers who want to enable their organization to accelerate their data science projects. If you are interested in better understanding automated machine learning life cycles and how it can benefit your organization, the next chapters will introduce a toolchain to achieve exactly that.

Summary

In this chapter, we have introduced the concept of machine learning pipelines and explained the individual steps. We have explained the benefits of automating this process. In addition, we have set the stage of the following chapters with a brief outline of every chapter and an introduction of our example project. In the next chapter we will start building our pipeline!

---

1 As long as the training and validation data is balanced.

2 Reuters article, October 9th, 2018 [Link to Come]

Chapter 2. Pipeline Orchestration

We discussed in Chapter 1 how machine learning pipelines consist of multiple steps (data validation, data preprocessing, model training and so on). The correct execution sequence of the tasks is essential for successful completion. Also, the output of each step, e.g. the data preprocessing, needs to be captured and used as inputs for the following tasks.

While data pipeline tools coordinate the machine learning pipeline steps, model tracking repositories like the *TensorFlow Extended MetaStore* capture the outputs of the individual processes. In this chapter we will introduce pipeline tools to coordinate the machine learning pipelines. In the following chapter, we will provide an overview of TensorFlow's MetaStore and look behind the scenes of the TensorFlow pipeline components.

This chapter serves as an introduction to the pipeline tools. We highly recommend a deep dive into the setup of the workflow

management tool of your choice if you plan to run the tool in a production environment.

Why Pipeline Orchestration

In 2014, a group of machine learning engineers at Google concluded that one of the reasons why machine learning projects are failing is that most projects come with custom code to bridge the gap between the machine learning pipeline steps. Project-specific scripts, often written in bash or Python, get the job of data science pipelines done. However, the scripts don’t transfer easily from one project to the next. The researchers summarized ...

Chapter 3. Data Validation with TensorFlow

In the last two chapters, we introduced how we can orchestrate machine learning workflows and manage the workflow's metadata. With this in mind, we can start building our workflows.

Data is the basis for every machine learning model, and the model's usefulness and performance depend on the data used to train, validate and analyze the model. As you can image, without robust data, we can't build robust models. Moreover, in colloquial terms, you might have heard the phrase: "Garbage in, garbage out" - meaning that our models won't perform if the underlying data isn't curated and validated. This is the exact purpose of our first workflow step in our machine learning pipeline: data validation.

In this chapter, we introduce you to a Python package from the TensorFlow ecosystem called TensorFlow Data Validation. We show you how you can set up the package in your data

science projects, walk you through the common use cases and highlight some very useful workflows.

TensorFlow Data Validation assists you in comparing multiple data sets with each other, and it highlights if your data schema changes over time (data drift) or if your training data is significantly different from your data to validate your models or data which is used to infer your model (data skew).

At the end of the chapter, we integrate our first workflow step into our Airflow pipelines.

Why Data Validation?

In machine learning, we are trying to learn from patterns in data sets and to generalize these learnings. This puts our data sets in the front and center of our machine learning workflows, and the data quality becomes fundamental to the success of our machine learning projects.

Every workflow step in our machine learning life cycle determines if the workflow can proceed to the next step or if the entire workflow needs to be abandoned and restarted, for example with more training data.

If our goal is the automation of our machine learning model updates, validating our data is essential. In particular, when we say validating, we mean three distinct checks on our data:

- Check for data anomalies
- Check that the data schema hasn’t changed
- Check that the statistics of our new data sets still align with statistics from our previous training data sets

Data validation performs these checks and highlights any failures. If a failure is detected, we can stop the workflow and address the data issue by hand, e.g., by curating a new data set or select different features.

We demonstrate in this chapter how data validation can assist you in your model feature engineering. It produces statistics around your data features and highlights if a feature contains a high percentage of missing values or if features are highly correlated.

Data validation lets you compare the statistics of different data sets. This simple step can assist you in debugging your model issues. For example, data validation can compare the statistics of your training against your validation data. With a few lines of code, it brings any difference to your attention. You might

train a binary classification model with a perfect label split of 50% positive labels and 50% negative labels, but the label split isn't 50/50 in your validation set. This difference in the label distribution ultimately skews your validation metrics.

In a world where data sets continuously grow, data validation is crucial to make sure that our machine learning models are still up to the task. Because we can compare schemas, we can quickly detect if the data structure in newly obtained data sets had changed, e.g. when a feature was deprecated. It can also detect if your data starts to *drift*. That means that your newly collected data has different underlying statistics than the initial data set used to train your model. This drift could mean that new features need to be selected or that the data preprocessing steps need to be updated, e.g. if the minimum or maximum of a numerical column changes.

In the following sections, we walk you through the different use cases we have just touched on. However, before that, let's take a look at the required installation steps to get TensorFlow Data Validation up and running.

TensorFlow Data Validation

The TensorFlow ecosystem offers a tool which can assist you in the data validation, called *TensorFlow Data Validation*, or *TFDV* for short. It is part of the TensorFlow Extended project.

TensorFlow Data Validation allows you to perform the kind of analyses we discussed previously, e.g. generating schemas and validating new data against an existing schema. However, it also offers visualizations based on the Google PAIR project *Facets* [Link to Come].

Sort by

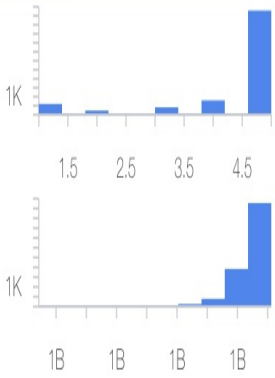
Feature order ▾ ☐ Reverse order Feature search (regex enabled)

Features: ☑ int(1) ☑ float(1) ☑ string(6)

Numeric Features (2)

Chart to show: Standard ▾ ☐ log ☐ expand

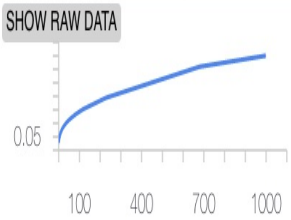
	count	missing	mean	std dev	zeros	min	median	max
overall	7,840	0%	4.4	1.2	0%	1	5	5
unixReviewTime	7,840	0%	1.37B	33.6M	0%	1.02B	1.38B	1.41B

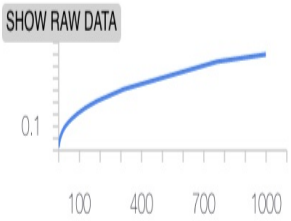


Categorical Features (6)

Chart to show: Standard ▾ ☐ log ☐ expand

	count	missing	unique	top	freq top	avg str len
reviewerID	7,840	0%	6,093	A1Y09Q...	213	13.76
asin	7,840	0%	5,705	B000SJ...	114	10
reviewerName	7,840	0%	5,715	"T=...	213	13.33





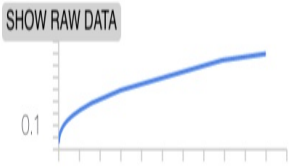


Figure 3-1. Screenshot of a TensorFlow Data Validation Visualization based on Facets

TensorFlow Data Validation accepts two input formats to start the data validation: TensorFlow's TFRecords and Command-separated values (CSV) files.

Behind the scenes, TensorFlow Data Validation distributes the analysis on Apache Beam, an open source tool to define and run data pipelines. However, this doesn't mean you need to learn a new tool; TensorFlow Data Validation is abstracting all of that for you.

Installation

The setup of TensorFlow Data Validation is straight-forward. You can install the PyPI package with

Example 3-1. PyPI installation of TensorFlow Data Validation

```
$ pip install tensorflow-data-validation
```

At the time of writing this chapter, TensorFlow Data Validation is installing the Apache Beam dependency version 2.11.0 automatically.

With this is brief installation, you can now integrate your data

validation into your machine learning workflows or analyze your data visually in a Jupyter notebook. Let’s walk through a couple of use-cases in the following sections.

Generating Statistics from your Data

As mentioned earlier, TensorFlow Data Validation lets you load TFRecord and CSV data sources without any hassle.

As an example, we can load our converted Amazon review data CSV data directly with TFDV and generate statistics for each feature.

Example 3-2. Generating Validation Statistics from CSV Data

```
import tensorflow_data_validation as tfdv
stats = tfdv.generate_statistics_from_csv(

data_location='reviews_Collectibles_and_Fine_Art.csv'
,
    delimiter='|')
```

You can generate feature statistics from TFRecords in a very similar way with

Example 3-3. Generating Validation Statistics from TFRecords Data

```
stats = tfdv.generate_statistics_from_tfrecord(

data_location='reviews_Collectibles_and_Fine_Art.tfr
```

```
ecords')
```

Both TensorFlow Data Validation methods generate a data structure which stores the metrics like minimum, maximum average values.

The data structure looks like this

Example 3-4. TensorFlow Data Validation Statistics Data Structure

```
datasets {
  num_examples: 7840
  features {
    name: "reviewerID"
    type: STRING
    string_stats {
      common_stats {
        num_non_missing: 7840
        min_num_values: 1
        max_num_values: 1
        avg_num_values: 1.0
        num_values_histogram {
          buckets {
            low_value: 1.0
            high_value: 1.0
            sample_count: 784.0
...
```

STATISTICS PROVIDED BY TENSORFLOW DATA VALIDATION

At the time of writing this chapter, TensorFlow Data Validation can generate the following statistics:

For numerical features, TensorFlow Data Validation computes for every feature:

- The overall count of data records
- The percentage of missing data records
- The mean and standard deviation across the feature
- The minimum and maximum value across the feature
- The percentage of zero values across the feature

In addition, it generates a histogram of the values for each feature.

For categorical features, TensorFlow Data Validation provides:

- The overall count of data records
- The percentage of missing data records
- The number of unique records
- The average string length of all records of a feature

- For each category, TFDV determines the sample count for each label and its rank

In a moment, you'll see how we can turn these statistics into something actionable.

CONVERTING YOUR DATA TO TFRECORDS

Before we discuss how we can use the data statistics, let's briefly talk about the input data. We mentioned that TensorFlow Data Validation supports two input formats: Comma-separate value and TFRecords files. However, what should you do if your data isn't available in either format? We recommend converting your data to TFRecords, and we'll show you how you can do that with a few lines of code.

TFRECORDS

TFRecords is a data structure to store data sets efficiently. TFRecord files contain a `tf.Example` record for every data record. Each record contains 1 or more features which would represent the columns in our data. If you are interested in the internals of TFRecords, we recommend the TensorFlow documentation [Link to Come].

TFRecords can be helpful to handle your data efficiently, e.g. when you want to load your data as an iterator or if you want to shuffle your data. TFRecords supports these actions on

your data out of the box.

The following code example highlights how you can convert any data source (in our example JSON data) to TFRecords which you can then use in your machine learning pipeline. As we'll discuss in our following chapter on data preprocessing, TFRecords are very useful to handle your model training data efficiently.

To convert your data to TFRecords, you need to create `tf.Example` structures for every data record in your data set. `tf.Example` represents your data records in Google's ProtoBuffer protocol to serialize the structured data. You don't need to worry about all the internals since TensorFlow is providing you with functions to convert your data into the ProtoBuf structure.

Example 3-5. TFRecord Data Structure

```
Record 1:
tf.Example
    tf.Features
        'column A': tf.train.Feature
        'column B': tf.train.Feature
        'column C': tf.train.Feature
```

TensorFlow Data Features can contain three different data

types:

- bytes
- float
- int64

To reduce the redundancy of code, we'll define helper functions (shown in Example 3-6) to assist with converting the data records into the correct data structure used by `tf.Example`.

Example 3-6. Convert your data to tf.train.Feature

```
import tensorflow as tf

def _bytes_feature(value):
    return
tf.train.Feature(bytes_list=tf.train.BytesList(value=
[value]))

def _float_feature(value):
    return
tf.train.Feature(float_list=tf.train.FloatList(value=
[value]))

def _int64_feature(value):
    return
tf.train.Feature(int64_list=tf.train.Int64List(value=
[value]))
```

With the helper functions in place, we can now read our original data file and convert every data record into a `tf.Example` data structure and save all records in a TFRecord file. The code snippet Example 3-7 was inspired by Julian McAuley [1]. It lets us load the Amazon review data directly from the compressed file, loop over every data record and write it to a TFRecord data file.

Example 3-7. Write your data to a TFRecord file

```
import gzip
import tensorflow as tf

tfrecords_filename =
'reviews_Collectibles_and_Fine_Art.tfrecords'
tf_record_writer =
tf.python_io.TFRecordWriter(tfrecords_filename) ❶

gdata =
gzip.open('reviews_Collectibles_and_Fine_Art.json.gz'
, 'r')
for data in gdata:
    record = eval(data) ❷
    example = tf.train.Example
(features=tf.train.Features(feature={ ❸
        'reviewerID':
_bytes_feature(record['reviewerID']),
        'asin': _bytes_feature(record['asin']),
        'reviewerName':
_bytes_feature(record['reviewerName']),
        'reviewText':
_bytes_feature(record['reviewText']),
```

```
        'overall':
_float_feature(record['overall']),
        'summary':
_bytes_feature(record['summary']),
        'unixReviewTime':
_int64_feature(record['unixReviewTime']),
        'reviewTime':
_bytes_feature(record['reviewTime'])}))

tf_record_writer.write(example.SerializeToString())
❹

writer.close()
```

❶ Creates a `TFRecordWriter` object which saaves to the path specified in `tfrecords_filename`

❷ `eval` converts each record to a Python dict data structure tf.train.Example for every data record

❸ We save the serialized data structure to the TFRecord file

❹

With the ability to generate statistics for your data from CSV files, TFRecods or any arbitrary data source, let's take a look now at how we use the statistics to infer a data schema from the statistics.

Generating Schema from your Data

Data schemata are a form of describing the representation of your data sets. A schema defines which features are expected in your data set and which type each feature is based on.

Besides, your schema should define the boundaries of your data, e.g. outlining minimums, maximums, thresholds of allowed missing records for a feature.

The schema definition of your data set can then be used to validate future data set to determine if they are “in line” with your previous training sets. The schemas generated by TensorFlow Data Validation can also be used in the following workflow step when you are preprocessing your data sets to convert them to data which can be used to train machine learning models.

As shown in Example 3-8, you can generate the schema information from your generated statistics with a single function call.

Example 3-8. Generating data schema from the previously generated statistics

```
schema = tfdv.infer_schema(stats)
```

`tfdv.infer_schema` generates a schema protocol defined by TensorFlow [2]

Example 3-9. Generated schema protocol after inferring the data schema

```
feature {
  name: "reviewerID"
```

```
  type: BYTES
  presence {
    min_fraction: 1.0
    min_count: 1
  }
  shape {
    dim {
      size: 1
    }
  }
}
feature {
  name: "asin"
  type: BYTES
  ...
```

As shown in Example 3-10, you can display the schema with a single function call in any Jupyter notebook.

Example 3-10. Visualize the generated schema

```
tfdv.display_schema(schema)
```

Feature name	Type	Presence	Valency	Domain
'reviewerID'	BYTES	required		-
'asin'	BYTES	required		-
'reviewerName'	BYTES	required		-
'reviewText'	BYTES	required		-
'overall'	FLOAT	required		-
'summary'	BYTES	required		-
'unixReviewTime'	INT	required		-
'reviewTime'	BYTES	required		-

Figure 3-2. Screenshot of a schema visualization

With the schema now defined, you can go ahead and compare your training or validation data sets or check your data sets against any data drift or skew.

Comparing Data Sets

Let's say you have two data sets, training and validation data set. Before training your machine learning model, you would like to determine how representative the validation set is in regards to the training set. Does the validation data follow your training data schema? Are any feature columns or a significant number of feature values missing? With TensorFlow Data Validation, you can quickly determine the answer.

As shown in Example 3-11, you can load both data sets and then visualize both data sets together. If you are executing the code below in a Jupyter notebook, you can compare the data set statistics easily.

Example 3-11. Visualize the generated schema

```
train_stats =
tfdv.generate_statistics_from_tfrecord(
    data_location=train_tfrecords_filename)
val_stats = tfdv.generate_statistics_from_tfrecord(
    data_location=val_tfrecords_filename)
```

```
tfdv.visualize_statistics(lhs_statistics=val_stats,
rhs_statistics=train_stats,
                          lhs_name='VAL_DATASET',
rhs_name='TRAIN_DATASET')
```

Sort by

Feature order ▾ Reverse order Feature search (regex enabled)

Features: int(1) float(1) string(6)

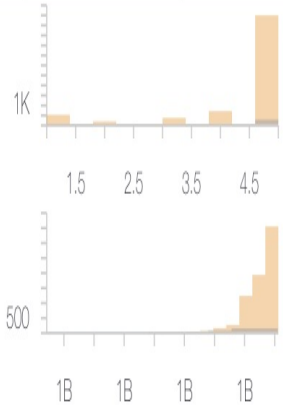
VAL_DATASET TRAIN_DATASET

Numeric Features (2)

Chart to show: Standard ▾

log expand percentages

	count	missing	mean	std dev	zeros	min	median	max
overall								
	499	0%	3.99	1.45	0%	1	5	5
	7,341	0%	4.42	1.18	0%	1	5	5
unixReviewTime								
	499	0%	1.33B	68.4M	0%	1.02B	1.36B	1.41B
	7,341	0%	1.38B	27.4M	0%	1.19B	1.38B	1.41B



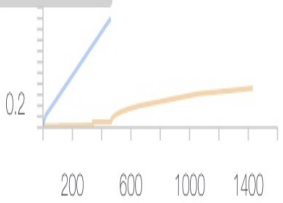
Categorical Features (6)

Chart to show: Standard ▾

log expand percentages

	count	missing	unique	top	freq top	avg str len
reviewerID						
	499	0%	463	A3B9XT...	11	13.76
	7,341	0%	5,707	A1Y09Q...	212	13.76
asin						
	233	53.31%	98	B000I03...	23	10
	7,341	0%	5,546	B000SJ...	114	10

SHOW RAW DATA



SHOW RAW DATA

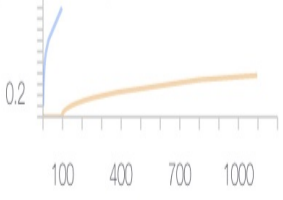


Figure 3-3. Comparison between a training and validation data set

Figure 3-3 shows the difference between the two data sets. For example, the validation data set (containing 499 records) has a lower average of `overall` values. This could mean that the feature is drifting away from the training data. More importantly, the visualization highlighted that more than half of all validation records don't contain `asin` information [3]. If the `asin` is an important feature for your model training, you would have to fix your data capturing methods to "harvest" the data with the correct product identifiers.

The schema of the training data we generated earlier now becomes very handy. TensorFlow Data Validation lets you validate any data statistics against the schema, and it reports any anomalies.

The Example 3-12 lets you detect the anomalies easily. The generated anomaly protocol can then be analyzed programmatically or visualized as shown in Example 3-13 and Figure 3-4.

Example 3-12. Check the validation set against the data schema from the training set

```
anomalies =
tfdv.validate_statistics(statistics=val_stats,
schema=schema)
```

Example 3-13. Visualize the anomalies in a Jupyter notebook

```
tfdv.display_anomalies(anomalies)
```

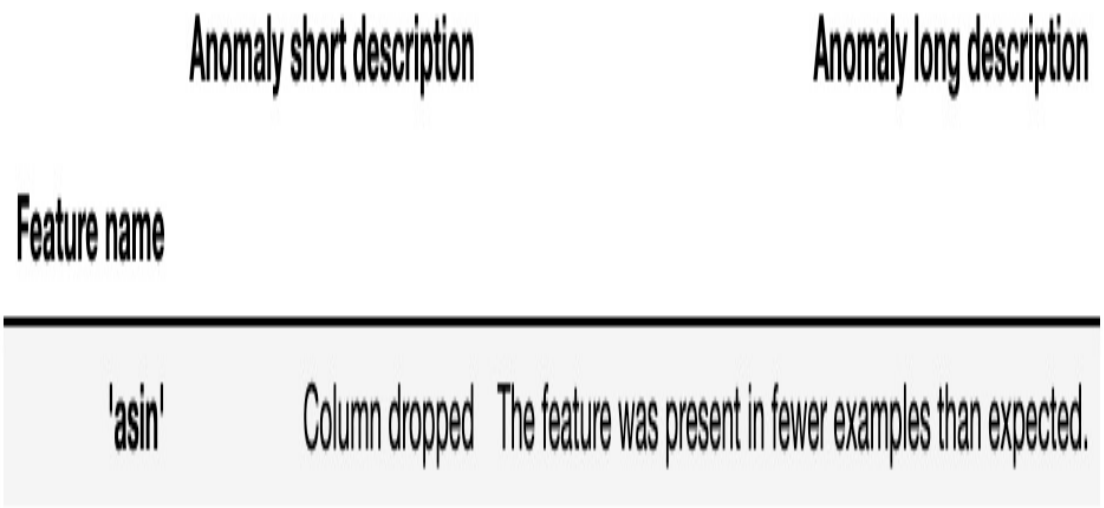


Figure 3-4. Visualize the anomalies in a Jupyter notebook

Example 3-14 shows the underlying anomaly protocol contains useful information which you can use to automate your machine learning workflow.

Example 3-14. Snippet from the anomaly protocol

```
anomaly_info {
  key: "asin"
  value {
    description: "The feature was present in fewer
examples than expected."
    severity: ERROR
    short_description: "Column dropped"
    reason {
```

```
      type: FEATURE_TYPE_LOW_FRACTION_PRESENT
      short_description: "Column dropped"
      description: "The feature was present in fewer
examples than expected."
    }
    path {
      step: "asin"
    }
  }
}
```

In the next two sections, we highlight two common use cases for data validation: Data skew and data drift detection.

Data Skew Detection

We speak of data skew if there is a difference between two data sets of different types. For example, when we split our initial data set into a training and validation data set, they can be skewed. At the same time, we speak of data skew when newly collected data doesn’t conform with the expected schema, feature details or distribution anymore. It is up to the data scientist to decide which skew is acceptable and at what level the workflow needs to be stopped and feature engineering needs to be performed again.

We can detect skew in the data schema with the anomaly

detection we discussed earlier in this chapter. Feature skew, for example, when a feature in newly collected data contains more missing values can be detected with the earlier discussed anomaly detection.

The last type of skew detection we haven't discussed is the change in data distribution between the data sets. TFDV provides you the option to compare `serving_statistics` against your training data set and its schema. Example 3-15 showcases how you can compare the data sets. If the difference between the two data sets exceeds your threshold of the L-infinity norm for a given feature, TensorFlow Data Validation highlights it as an anomaly.

Example 3-15. Snippet from the anomaly protocol

```
tfdv.get_feature(schema,
'asin').skew_comparator.infinity_norm.threshold =
0.01
skew_anomalies = tfdv.validate_statistics(
        statistics=train_stats, schema=schema,
serving_statistics=serving_stats)
```

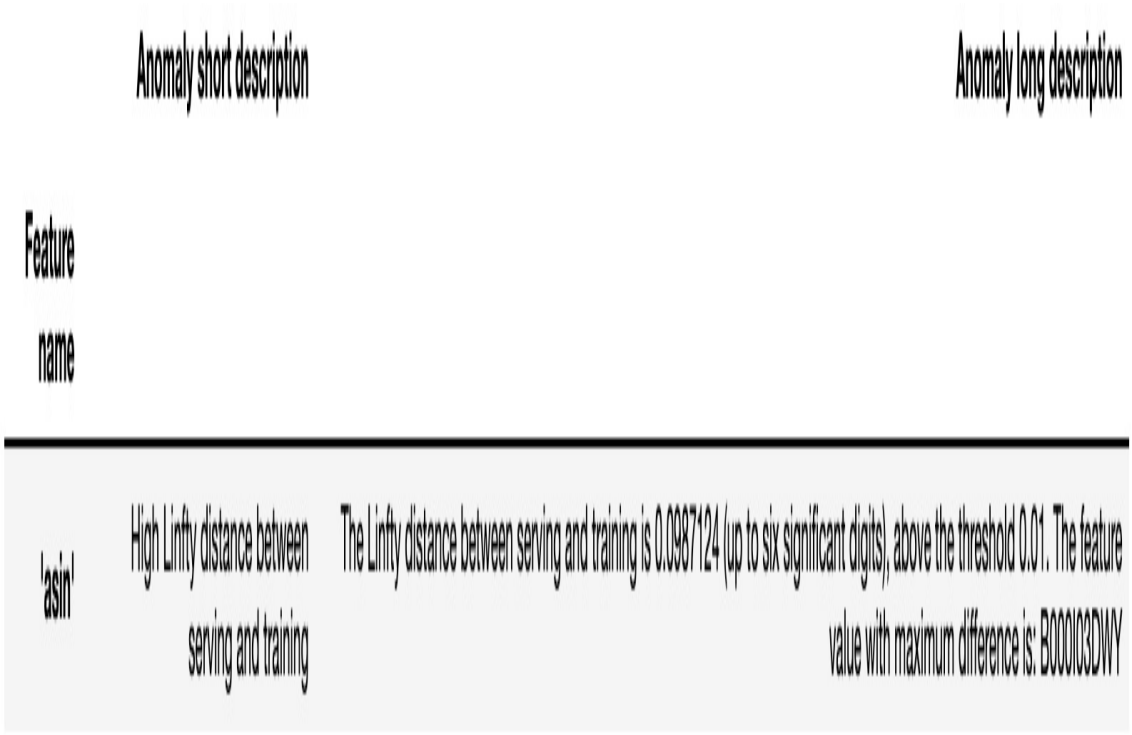


Figure 3-5. Visualization of the data skew between our training and serving data set

L-INFINITY NORM

The *L-infinity norm* is an expression to define the difference between two vectors (in our case, features). The L-infinity norm is defined as the maximum absolute value of the vector's entries.

Data Drift Detection

In contrast to data skew, data drift compares two data sets of

the same type, e.g. two training sets collected on two different days. There is often the use case to determine if training sets "drift" from the original data set. If drift is detected, the data scientist should check either the model architecture or determine if feature engineering needs to be performed again.

Similar to our skew example, you define your `drift_comparator` for the features you would like to watch and compare. You can then call `validate_statistics` with the two data set statistics as arguments, one for your basis (e.g., yesterday's data set) and the data set statistics you want to compare them with (e.g., today's data set).

Example 3-16. Snippet from the anomaly protocol

```
tfdv.get_feature(schema,
'asin').drift_comparator.infinity_norm.threshold =
0.01
skew_anomalies = tfdv.validate_statistics(
        statistics=train_stats_today, schema=schema,
previous_statistics=train_stats_yesterday)
```

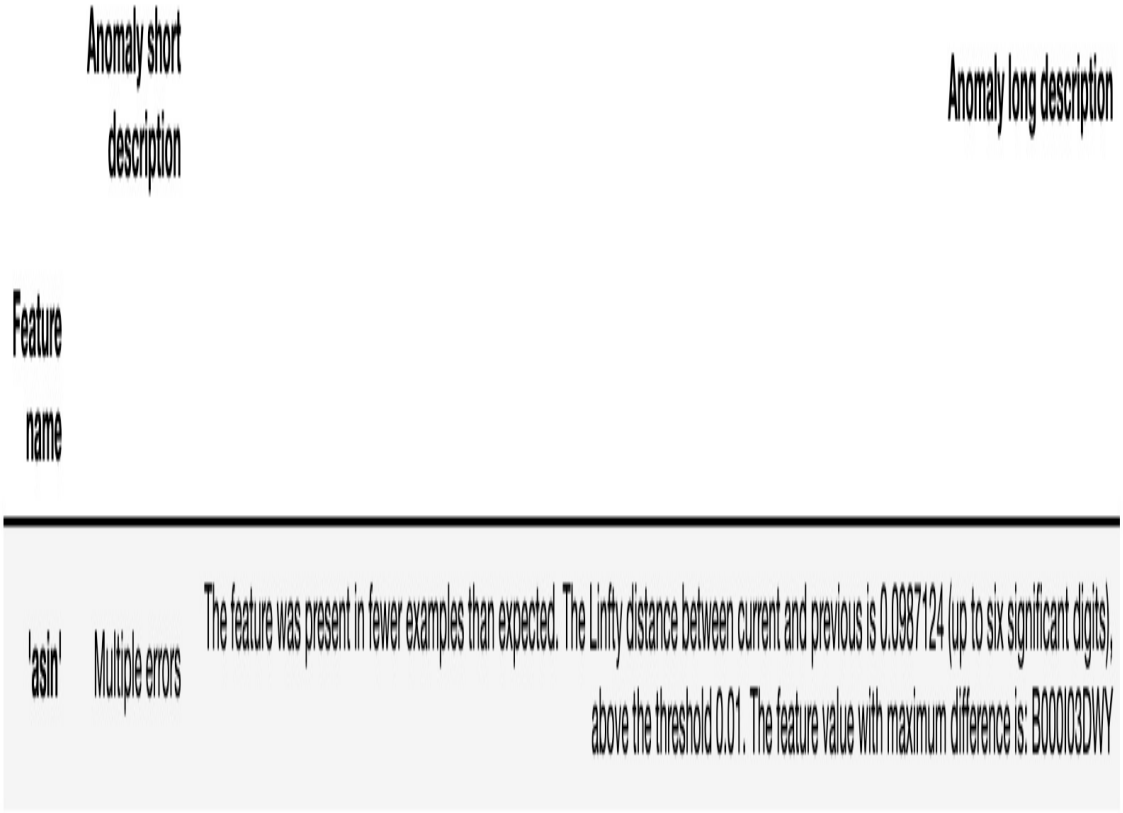


Figure 3-6. Visualization of the data drift between two training sets

[[Google Cloud Dataflow]] === Processing large Data Sets with Google Cloud Platform

As you collect more data, the data validation becomes a more time-consuming step in your machine learning workflow. One way of reducing the time to perform the validation is by taking advantage of available cloud solutions. By using a cloud provider, you aren’t limited to the computation power limited by your laptop or your on-premise data center.

As an example of usage of a cloud provider, we'll be introducing how to run TensorFlow's data validation on Google Cloud's product *DataFlow*. TensorFlow Data Validation is running on Apache Beam, which makes a switch to GCP DataFlow very easy.

Google Cloud's DataFlow lets you accelerate your data validation tasks by parallelizing and distributing them across the allocated nodes for your data processing task. While *DataFlow* charges for the number of CPUs and Gigabytes of memory allocated, it can speed up your pipeline step.

We'll demonstrate a minimal setup to distribute your data validation tasks. For more information, we highly recommend the extended Google Cloud Platform documentation [4]. We assume that you have a Google Cloud account created, the billing details set up and the `GOOGLE_APPLICATION_CREDENTIALS` environment variable set in your terminal shell. If you need help to get started, please refer to Appendix A.

You can use the same method we discussed previously, e.g. `tfdv.generate_statistics_from_tfrecord`, but the methods require additional arguments, `pipeline_options` and `output_path`. While

`output_path` points at the Google Cloud bucket where the data validation results should be written to, `pipeline_options` is an object which contains all the Google Cloud details to run your data validation on Google Cloud. Example 3-17 and Example 3-18 show you how you can set up such a pipeline object.

We recommend creating a storage bucket for your DataFlow tasks. The storage bucket holds all the data sets and temporary files.

Example 3-17. Configure the Google Cloud options to run TensorFlow Data Validation with DataFlow

```
from apache_beam.options.pipeline_options import (
    PipelineOptions, GoogleCloudOptions,
StandardOptions)

options = PipelineOptions()
google_cloud_options =
options.view_as(GoogleCloudOptions)
google_cloud_options.project =
'<YOUR_GCP_PROJECT_ID>' ❶
google_cloud_options.job_name = '<YOUR_JOB_NAME>' ❷
google_cloud_options.staging_location =
'gs://<YOUR_GCP_BUCKET>/staging' ❸
google_cloud_options.temp_location =
'gs://<YOUR_GCP_BUCKET>/tmp'
options.view_as(StandardOptions).runner =
'DataflowRunner'
```

❶ Set your project id

Give your job a name
❷ Point towards a storage bucket for staging and tmp files
❸

Once you have configured the Google Cloud options, you need to configure the setup for the DataFlow workers. All tasks are executed on workers which need to be provisioned with the necessary packages to run the tasks. In our case, we need to install TensorFlow Data Validation by specifying it as an additional package. To do that, download the latest TensorFlow Data Validation package (the binary whl file) [5] to your local system. Choose a version which can be executed on a Linux system, e.g. `tensorflow_data_validation-VERSION_NUMBER-cp27-cp27mu-manylinux1_x86_64.whl`. To configure the worker setup options, specify the path to the downloaded package in the `setup_options.extra_packages` list as shown in Example 3-18.

Example 3-18. Configure the setup options to instantiate your DataFlow workers

```
from apache_beam.options.pipeline_options import
SetupOptions

setup_options = options.view_as(SetupOptions)
setup_options.extra_packages = [
    '/path/to/tensorflow_data_validation-0.13.1-
cp27-cp27mu-manylinux1_x86_64.whl']
```

With all the option configurations in place, you can kick off the data validation task from your local machine, and they are executed on the Google Cloud DataFlow instances.

Example 3-19. Configure the Google Cloud options to run TensorFlow Data Validation with DataFlow

```
data_set_path =
'gs://<YOUR_GCP_BUCKET>/train_reviews.tfrecords'
output_path = 'gs://<YOUR_GCP_BUCKET>/'
tfdv.generate_statistics_from_tfrecord(data_set_path,


output_path=output_path,

pipeline_options=options)
```

After you have started the data validation with DataFlow, you can switch back to the Google Cloud console. Your newly kicked off job should be listed similar to Figure 3-7.

Name	Type	End time	Elapsed time	Start time	Status	SDK version	ID	Region
dataflow-example	Batch	-	36 sec	Mar 30, 2019, 12:33:06 PM	Running	2.10.0	2019-03-30_12_33_05-2316197501266235547	us-central1

Figure 3-7. Google Cloud DataFlow Job Console

You can then check the details of the running job, its status and its autoscaling details.


Google Cloud Platform
OReilly Book
dataflow-example
LOGS
Job
ReadData
Running
DecodeData
Running
GenerateStatistics
Part running
WriteStatsOutput
Part running
Job summary
Job name
dataflow-example
Job ID
2019-03-30_12_33_05-2316197501266235547
Region
us-central1
Job status
Running
Stop job
SDK version
Apache Beam SDK for Python 2.10.0
A newer version of the SDK family exists, and updating is recommended. Learn more
Job type
Batch
Start time
Mar 30, 2019, 12:33:06 PM
Elapsed time
53 sec
Autoscaling
Resource metrics
Current vCPUs
0
Total vCPU time
0 vCPU hr
Current memory
0 B
Total memory time
0 GB hr
Current PD
0 B
Total PD time
0 GB hr
Current SSD PD
0 B

Figure 3-8. Google Cloud DataFlow Job Details

You can see with a few steps you can parallelize and distribute your data validation tasks in a cloud environment. In the next section, we'll discuss the integration of the data validation tasks into our automated machine learning pipelines.

Integrate TensorFlow Data Validation into your Machine Learning Pipeline

In the two previous chapters, we laid the groundwork for our automated machine learning pipelines. We discussed how you can orchestrate pipelines and how the required metadata is stored so that the individual pipeline tasks can access the results from the previous steps. Now, it's time to put all these things together and start integrating our machine learning pipelines.

As we discuss elsewhere, the TensorFlow ecosystem provides a framework called TensorFlow Extended. The framework is performing the orchestration between the pipeline tasks, the central data repository for the pipeline tasks, the `meta store` and our pipeline orchestration tool. The components provided by the `tfx` framework act as a wrapper around the

various TensorFlow Extended packages and they communicate with the TensorFlow MetaStore to read the output from the previous workflow step and to save the results from each component.

In our first machine learning workflow step, we want to:

- Read a data set and convert it to `tf.Example` data structure
- Generate statistics of the data set
- Generate a schema of the data set
- Validate the statistics and the schema against our previous data sets

For every of our desired workflow steps, `tfx` provides a component which is performing the task for us. Let's steps through the steps and add them to our Airflow DAG we had set up in Chapter 2.

Example 3-20 shows how you can "prep" the data for the workflow. You can see, you don't need to interact with the TensorFlow MetaStore. It all happens behind the scene.

Example 3-20. Converting our sample data set to tf.Example records for our data pipeline

```
from tfx.utils.dsl_utils import csv_input
from tfx import components

examples = csv_input('/path/to/the/csv/data')
examples_gen =
components.ExamplesGen(input_data=examples)
```

Once the data is converted to `tf.Example` records, we can generate our statistics. This can be done with a single Python command as shown in Example 3-21.

Example 3-21. Generating our Data Set Statistics

```
compute_eval_stats = components.StatisticsGen(
    input_data=examples_gen.outputs.eval_examples,
    name='compute-eval-stats')
```

As easily as we can generate our statistics, we can generate our data schema. Example 3-22 shows you how to do it.

Example 3-22. Generating our Data Set Schema

```
infer_schema = components.SchemaGen(
    stats=compute_training_stats.outputs.output)
```

With the statistics and schema in place, we can now validate our new data set as shown in Example 3-23.

Example 3-23. Validating our newly generated Data Schema and Statistics

```
validate_stats = components.ExampleValidator(
```

```
    stats=compute_eval_stats.outputs.output,
    schema=infer_schema.outputs.output)
```

The component objects we just set up can now be added to our component list which is passed to the pipeline orchestration tool.

```
components = [ ❶
    examples_gen,
    compute_eval_stats,
    validate_stats
]

_pipeline = pipeline.Pipeline(
    pipeline_name='your_ml_pipeline',
    pipeline_root=pipeline_root,
    components=components,
)

AirflowDAGRunner(_airflow_config).run(_pipeline)
```

Add data validation components to the pipeline

❶

By adding the newly created components to the pipeline component list, the pipeline orchestration tool will detect the new task dependencies.

As you saw, we implemented our first machine learning

workflow step efficiently. Should there be a misalignment between the data set statistics or schema between the new and the previous data set, the components throw an error, and the data pipeline fails. Otherwise, the data pipeline moves on to the next step, the data preprocessing.

Summary

In this chapter, we discussed the importance of data validation and how you can efficiently perform and automate it. We discussed how to generate data statistics and schemas and how to compare two different data sets based on the statistics and schemas. We stepped through an example of how you could run your data validation on Google Cloud with DataFlow, and ultimately we integrated the machine learning step in our automated pipeline.

In the following chapters, we extend our pipeline setup starting with the data preprocessing in the next chapter.

---

1 Amazon Review data sets and code snippets on how to load the data sets can be found at *http://snap.stanford.edu/data/amazon/productGraph/*

2 You can find the protobuf definitions for the schema protocol in the TensorFlow repository: *https://github.com/tensorflow/metadata/blob/master/tensorflow_metada*

ta/proto/v0/schema.proto

3 `asin` is Amazon's unique identifier for products.

4 *https://cloud.google.com/dataflow/docs*

5 Download TFDV packages from *https://pypi.org/project/tensorflow-data-validation/#files*

Chapter 4. Model Deployment with TensorFlow Serving

The deployment of your machine learning model is the last step before others can use your model and make predictions with the model. Unfortunately, the deployment of machine learning models falls into a grey zone in today's thinking of the division of labor in the digital world. It isn't just a DevOps task since it requires some knowledge of the model architecture and its hardware requirements. At the same time, deploying deep learning models is a bit outside of the comfort zone of machine learning engineers and data scientists. They know their models inside out but tend to struggle with the DevOps side of the deployment part. In this and the following chapters, we want to bridge the gap between the worlds and guide data scientists and DevOps engineers through the steps to deploy machine learning models.

Machine learning models can be deployed in mainly three

ways. The most common way today is the deployment of a machine learning model to a model server. The client which requests a prediction submits the input data to the model server and in return will receive a prediction. This requires that the client can connect with the model server. In this chapter, we are focusing on this type of model deployment.

There are situations where you don’t want to submit the input data to a model server, e.g., when the input data is sensitive, or there are privacy concerns. In this situation, you can deploy the machine learning model to the user’s browser. For example, if you want to determine if an image contains sensitive information, you could classify the sensitivity level of the image before it is uploaded to a cloud server.

However, then there is a third type of model deployment: to edge devices. There are situations which don’t allow you to connect to a model server to make predictions, e.g. remote sensors or IoT devices. The number of applications being deployed to edge devices is increasing and it is now a valid option for model deployments. In the following chapter, we will describe the last two mentioned methods.

In this chapter, we highlight TensorFlow’s Serving module and introduce its setup and usage. This is not the only way of deploying deep learning models; there are a few options

existing at the moment. At the time of writing this chapter, we feel that Tensorflow Serving offers the most simple server setup and the best performance.

Let’s start the chapter with how you shouldn’t set up a model server, before we deep dive into TensorFlow Serving.

A Simple Model Server

Most introductions to deploying machine learning models follow roughly the same script:

- Create a web app with Python (Flask or Django)
- Create an API endpoint in the web app
- Load the model structure and its weights
- Call the predict method on the loaded model
- Return the prediction results as an HTTP request

Example 4-1 shows an example implementation of such an endpoint for our basic model.

Example 4-1. Example Setup of a Flask Endpoint to Infer Model Predictions

```
import json
from flask import Flask
from keras.models import load_model
from utils import preprocess ❶

model = load_model('model.h5') ❷
app = Flask(__name__)

@app.route('/classify', methods=['POST'])
def classify():
    review = request.form["review"]
    preprocessed_review = preprocess(review)
    prediction =
model.predict_classes([preprocessed_review])[0] ❸
    return json.dumps({"score": int(prediction)})
```

Preprocessing to convert characters to indices

❶ Load your trained model

❷ Perform the prediction and return the prediction in the http

❸ response

This setup is a quick and easy implementation, perfect for demonstration projects. We do not recommend using Example 4-1 to deploy machine learning models to production endpoints.

In the next section, let’s discuss why we don’t recommend deploying deep learning models with such a setup. The reasons are our benchmark for our proposed deployment solution.

Why it isn’t Recommended

While the Example 4-1 implementation can be sufficient for demonstration purposes, it has some significant drawbacks for scalable machine learning deployments.

CODE SEPARATION

In our demonstration example Example 4-1, we assumed that the trained model is deployed with the API code base and also lives in the code base. That means that there is no separation between the API code and the machine learning model. This can be problematic when the data scientists want to update a model, and such an update requires coordination with the API team. Such coordination also requires that the API and data science teams work in sync to avoid unnecessary delays on the model deployments.

With the intertwined API and data science code base, it also creates ambiguity around the API ownership.

The missing code separation also requires that the model has to be loaded in the same programming language as the API code is written. This mixing of backend and data science code can ultimately prevent your API team from upgrading your API backend.

In this chapter, we highlight how you can separate your models from your API code effectively and simplify your deployment workflows.

LACK OF MODEL VERSION CONTROL

Example 4-1 doesn't provide any provision for different model versions. If you wanted to add a new version, you would have to create a new endpoint (or add some branching logic to the existing endpoint). This requires extra attention to keep all endpoints structurally the same, and it requires much boilerplate code.

The lack of model version control also requires the API and the data science team to coordinate which version is the default version and how to phase in new models.

INEFFICIENT MODEL INFERENCE

For any request to your prediction endpoint based written in the Flask setup as shown in Example 4-1, a full round trip is performed. That means each request is preprocessed and inferred individually. The key reason why we argue that such a setup is only for demonstration purposes is that it is highly inefficient. During the training of your model, you probably use a batching technique which allows you to compute

multiple samples at the same time and then apply the gradient change for your batch to your network's weights. You can apply the same technique when you want the model to make predictions. A model server can gather all requests during an expectable timeframe or until the batch is full and ask the model for its predictions. This is an especially effective method when the inference runs on GPUs.

In this chapter, we introduce how you can easily set up such a batching behavior for your model server.

TensorFlow Serving

As you have seen along with the chapters of this book, TensorFlow comes with a fantastic ecosystem of extensions and tools. One of the earlier open-sourced extensions was TensorFlow Serving. It allows you to deploy any TensorFlow graph and you can make predictions from the graph through the standardized endpoints. As we discuss in a moment, TensorFlow Serving handles the model and version management for you, lets you serve models based on policies and lets you load your models from various sources. At the same time, it is focused on high-performance throughput for low-latency predictions. TensorFlow Serving is being used

internally at Google and is adopted by a good number of corporations and startups [1].

TensorFlow Architecture Overview

TensorFlow Serving provides you the functionality to load models from a given source (e.g. AWS S3 buckets) and notifies the Loader if the source has changed. As [Link to Come] shows, everything behind the scenes of TensorFlow Serving is controlled by a Model Manager which manages when to update the models and which model is used for the inferences. The rules for the inference determination are set by the policy which is managed by the model manager. Depending on your configuration, you can, for example, load one model at a time and have the model update automatically once the source module detects a newer version.

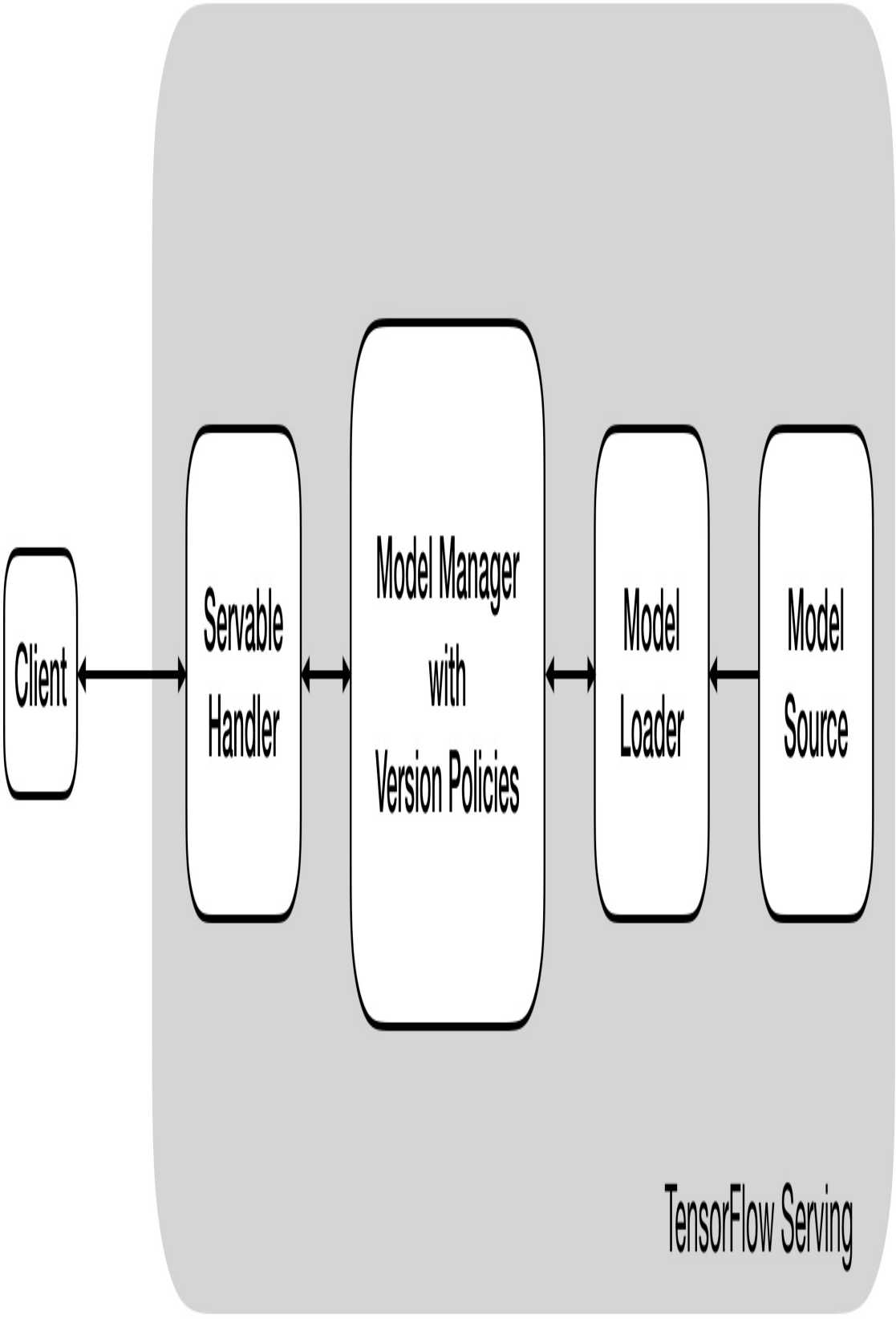

Client
Servable Handler
Model Manager with Version Policies
Model Loader
Model Source
TensorFlow Serving

Figure 4-1. Overview of the TensorFlow Serving Architecture

Before the deep dive into the server configuration to show you how you can set up these policies, let's talk about how we need to export the models so that they can be loaded into TensorFlow Serving.

Exporting Models for TensorFlow Serving

Before we dive into the TensorFlow Serving configurations, let's discuss how you can export your machine learning models so that they can be used by TensorFlow Serving.

Depending on your type of TensorFlow model, the export steps are slightly different. The exported models have the same file structure as we see in a moment.

For Keras models, you can use

```
saved_model_path = tf.saved_model.save(model,
"./saved_models")
```

while for TensorFlow Estimator models, you need to first

declare a `receiver function`

```
    def serving_input_receiver_fn():
        sentence = tf.placeholder(dtype=tf.string,
    shape=[None, 1], name='sentence')
        fn =
    tf.estimator.export.build_raw_serving_input_rece
    iver_fn(
            features={'sentence': sentence})
        return fn
```

and then export the `Estimator` model with the `export_saved_model` method of the estimator.

```
    estimator = tf.estimator.Estimator(model_fn,
    'model', params={})
    estimator.export_saved_model('saved_models/',
    serving_input_receiver_fn)
```

Both export methods produce output which looks similar to this following output.

```
    ...
    WARNING:tensorflow:Model was compiled with an
    optimizer, but the optimizer is not from
    `tf.train` (e.g. `tf.train.AdagradOptimizer`).
    Only the serving graph was exported. The train
    and evaluate graphs were not added to the
```

```
SavedModel.
WARNING:tensorflow:From
/usr/local/lib/python3.6/dist-
packages/tensorflow/python/saved_model/signature
_def_utils_impl.py:205: build_tensor_info (from
tensorflow.python.saved_model.utils_impl) is
deprecated and will be removed in a future
version.
Instructions for updating:
This function will only be available through the
v1 compatibility library as
tf.compat.v1.saved_model.utils.build_tensor_info
or tf.compat.v1.saved_model.build_tensor_info.
INFO:tensorflow:Signatures INCLUDED in export
for Classify: None
INFO:tensorflow:Signatures INCLUDED in export
for Regress: None
INFO:tensorflow:Signatures INCLUDED in export
for Predict: ['serving_default']
INFO:tensorflow:Signatures INCLUDED in export
for Train: None
INFO:tensorflow:Signatures INCLUDED in export
for Eval: None
INFO:tensorflow:No assets to save.
INFO:tensorflow:No assets to write.
INFO:tensorflow:SavedModel written to:
saved_models/1555875926/saved_model.pb
Model exported to:  b'saved_models/1555875926'
```

In your specified output directory, in our example we used `saved_models/`, we find the exported model. For every exported model, TensorFlow creates a directory with the timestamp of the export as its folder name.

```
$ tree saved_models/
saved_models/
└── 1555875926
    ├── assets
    │   └── saved_model.json
    ├── saved_model.pb
```

```
    └── variables
        ├── checkpoint
        ├── variables.data-00000-of-00001
        └── variables.index

3 directories, 5 files
```

The folder contains the following files and subdirectories:

saved_model.pb

The binary protobuf file contains the exported model graph structure as a `MetaGraphDef` object.

variables

The folder contains the binary files with the exported variable values and checkpoints corresponding to the exported model graph.

assets

This folder is created when additional files are needed to load the exported model. The additional file can vocabularies which we have seen in the data preprocessing chapter.

MODEL VERSIONING

When data scientists and machine learning engineers export models, the question of model versioning often comes up. In Software Engineering developers have the common understanding that minor changes and bug fixes lead to an increase of the minor version number and large code changes, especially breaking changes, require an update of the major version number.

In machine learning, we have at least 2 more degrees of freedom: While your model code describes the model's architecture and the support functionality (e.g. preprocessing), you can encounter different model performance with a single code change.

Your model can produce very different prediction results just by tweaking model hyperparameters like the learning rate, dropout rate and so on.

At the same time, no changes to the model architecture and parameters can produce a model with different performance characteristics.

No clear consensus has been established around the versioning of models. From our experience, this worked very well:

- Changes in the model architecture (e.g., new layers) require a new model name. The name should be descriptive of the model's architecture.

- Changes to the model's hyperparameters

require a new model version number.

- Training the same model with a more extensive data set also requires a new model version number to differentiate that the model performance will be different than from previous model exports.

Instead of increasing the model version number, it has been beneficial to use the Unix epoch timestamp of the export time as the version number. The newer model versions will then always have a higher model version number, and the machine learning engineer doesn't need to worry about which version to increase.

Model Signatures

Models are exported with a together with a signature which specifies the graph inputs and outputs. Inputs to your model graph are determined by your Keras `InputLayer` definitions or, in the case of a TensorFlow Estimator, they are defined by your `serving_input_receiver_fn` function. The model outputs are determined by the model graph.

For example, a classification model takes an input `sentence`

and outputs the predicted `classes` together with the corresponding `scores`.

```
signature_def: {
  key  : "classification_signature"
  value: {
    inputs: {
      key  : "inputs"
      value: {
        name: "sentence:0"
        dtype: DT_STRING
        tensor_shape: ...
      }
    }
    outputs: {
      key  : "classes"
      value: {
        name: "index_to_string:0"
        dtype: DT_STRING
        tensor_shape: ...
      }
    }
    outputs: {
      key  : "scores"
      value: {
        name: "TopKV2:0"
        dtype: DT_FLOAT
        tensor_shape: ...
      }
    }
```

```
    method_name: "tensorflow/serving/classify"
  }
}
```

TensorFlow Serving provides high-level APIs for three common use cases:

- Classification
- Prediction
- Regression

Each inference use case corresponds to a different model signature.

For example, linear regression models based on `tf.estimator.LinearRegressor` are exported with a regression signature like

```
signature_def: {
  key  : "regression_signature"
  value: {
    inputs: {
      key  : "inputs"
      value: {
        name: "input_tensor_0"
```

```
        dtype: ...
        tensor_shape: ...
      }
    }
    outputs: {
      key  : "outputs"
      value: {
        name: "y_outputs_0"
        dtype: DT_FLOAT
        tensor_shape: ...
      }
    }
    method_name: "tensorflow/serving/regress"
  }
}
```

At the same time, prediction models based on TensorFlow's `tf.estimator.LinearEstimator` are exported with prediction signature

```
signature_def: {
  key  : "prediction_signature"
  value: {
    inputs: {
      key  : "inputs"
      value: {
        name: "sentence:0"
        dtype: ...
        tensor_shape: ...
```

```
      }
    }
    outputs: {
      key  : "scores"
      value: {
        name: "y:0"
        dtype: ...
        tensor_shape: ...
      }
    }
    method_name: "tensorflow/serving/predict"
  }
}
```

Inspecting Exported Models

After all the talk about exporting your model and the corresponding model signatures, let's discuss how you can inspect the exported models before deploying them with TensorFlow Serving.

When you install the TensorFlow Serving Python API with

```
$pip install tensorflow-serving-api
```

you have access to a handy command line tool called

SavedModel CLI. `saved_model_cli` lets you

- Inspect the signatures of exported models: This is very useful primarily when you didn't export the model yourself, and you want to learn about the inputs and outputs of the model graph.
- Test the exported models: The CLI tools let you infer the model without deploying it with TensorFlow Serving. This is extremely useful when you want to test your model input data.

Inspecting the Model

`saved_model_cli` helps you understand the model dependencies without inspecting the original graph code.

If you don't know the available `tag-sets`, you can inspect the model with

```
$ saved_model_cli show --dir saved_models/
The given SavedModel contains the following tag-
sets:
serve
```

If your model contains different graphs for different environments, e.g., a graph for a CPU or GPU inference, you

will see multiple tags. If your model contains multiple tags, you need to specify a tag to inspect the details of the model.

Once you know the `tag_set` you want to inspect, add it as an argument, and `saved_model_cli` will provide you the available model signatures. Our example model has only one signature which is called `serving_default`.

```
$ saved_model_cli show --dir saved_models/ --
tag_set serve
The given SavedModel `MetaGraphDef` contains
`SignatureDefs` with the
following keys:
SignatureDef key: "serving_default"
```

With the `tag_set` and `signature_def` information, you can now inspect the model's inputs and outputs. To obtain the detailed information, add the `signature_def` to the CLI arguments.

```
$ saved_model_cli show --dir saved_models/ \
        --tag_set serve --signature_def
serving_default
The given SavedModel SignatureDef contains the
following input(s):
  inputs['sentence'] tensor_info:
      dtype: DT_STRING
      shape: (-1, 1)
      name: x:0
The given SavedModel SignatureDef contains the
following output(s):
```

```
  outputs['rating'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 1)
      name: rating:0
Method name is: tensorflow/serving/predict
```

If you want to see all signatures regardless of the `tag_set` and `signature_def`, you can use the `--all` argument

```
$ saved_model_cli show --dir saved_models/ --all
...
```

Testing the Model

`saved_model_cli` also lets you test the export model with sample input data.

You have three different ways to submit the sample input data for the model test inference.

`--inputs`

The argument points at a NumPy file containing the input data formatted as NumPy `ndarray`.

`--input_exprs`

The argument allows you to define a Python expression to

specify the input data. You can use NumPy functionality in your expressions.

--input_examples

The argument is expecting the input data formatted a `tf.Example` data structure (check the chapter on data validation where we introduced the `tf.Example` data structure)

For testing the model, you can specify exactly one of the input arguments. Furthermore, `saved_model_cli` provides three optional arguments:

--outdir

`saved_model_cli` will write any graph output to `stdout`. If you rather would like to write the output to a file, you can specify the target directory with `--outdir`.

--overwrite

If you opt for writing the output to a file, you can specify with `--overwrite` that the files can be overwritten.

--tf_debug

If you further want to inspect the model, you can step through the model graph with the TensorFlow Debugger (tfdbg).

Here is an example inspection of our demonstration model:

```
$ saved_model_cli run --dir saved_models/ --
tag_set serve \
--signature_def x1_x2_to_y --input_examples
`examples=[{"state_xf":"CA"}]`
```

After all the introduction of how to export models and how to inspect them, let’s dive into the TensorFlow Serving installation, setup and operation.

Setting up TensorFlow Serving

There are two easy ways to get TensorFlow Serving installed on your serving instances. You can either run TensorFlow Serving on Docker or, if you run an Ubuntu OS on your serving instances, you can install the Ubuntu package.

Docker Installation

The easiest way of installing TensorFlow Serving is downloading the pre-build docker image. As you have seen in Chapter 2, you can obtain the image by running

```
$ docker pull tensorflow/serving
```

> **NOTE**
>
> If you haven't installed or used `Docker` before, check out our brief introduction in Appendix [Link to Come].

If you are running the Docker container on an instance with GPU's available you will need to download the latest build with GPU-support.

```
$ docker pull tensorflow/serving:latest-gpu
```

The Docker image with GPU support requires NVIDIA's Docker support for GPUs. The installation steps can be found on the company's website [2].

Native Ubuntu Installation

If you want to run TensorFlow Serving without the overhead

of running Docker, you can install Linux binary packages available for Ubuntu distributions.

The installation steps are similar to other non-standard Ubuntu packages. First, you need to add a new package source the distribution's source list or add a new list file to the `sources.list.d` directory by executing

```
$ echo "deb [arch=amd64]
http://storage.googleapis.com/tensorflow-
serving-apt \
  stable tensorflow-model-server tensorflow-
model-server-universal" \
  | sudo tee /etc/apt/sources.list.d/tensorflow-
serving.list
```

in your Linux terminal. Before updating your package registry, you should add the packages' public key to your distribution's key chain.

```
$ curl
https://storage.googleapis.com/tensorflow-
serving-apt/\
tensorflow-serving.release.pub.gpg | sudo apt-
key add -
```

After updating your package registry, you can install TensorFlow Serving on your Ubuntu operating system.

```
$ apt-get update
$ apt-get install tensorflow-model-server
```

WARNING

Google provides two Ubuntu packages for TensorFlow Serving! The earlier referenced **tensorflow-model-serve** package is the preferred package, and it comes with specific CPU optimizations pre-compiled (e.g., AVX instructions).

At the time of writing this chapter, a second package with the name **tensorflow-model-server-universal** is also provided. It doesn't contain the pre-compiled optimizations and can, therefore, be run on old hardware (e.g., CPUs without the AVX instruction set).

Building TensorFlow Serving from Source

Should you be in the situation that you can run TensorFlow Serving on a pre-built Docker image or take advantage of the Ubuntu packages, for example, if you are running on a different Linux distribution, you can build TensorFlow

Serving from the source.

At the moment, you can only build TensorFlow Serving for Linux operating systems and the build tool `bazel` is required. You can find detailed instructions in the TensorFlow Serving documentation [3].

If you build TensorFlow Serving from scratch, it is highly recommended to compile the Serving version for the specific TensorFlow version of your models and the available hardware of your serving instances.

Configure a TensorFlow Server

Out of the box, TensorFlow Serving can run in two different modes. You can specify a model, and TensorFlow Serving will always provide the latest model. Alternatively, you can specify a configuration file with all models and versions to be loaded, and TensorFlow Serving will load all named models.

Single Model Configuration

If you want to run TensorFlow Serving loading a single model and switch to newer model versions when they are available,

the single model configuration is preferred.

If you run TensorFlow Serving in a Docker environment, you can run the `tensorflow\serving` image with the following command

```
$ docker run -p 8500:8500 \ ❶
             -p 8501:8501 \
             --mount
type=bind,source=/tmp/models,target=/models/my_m
odel \ ❷

             -e MODEL_NAME=my_model ❸
             -t tensorflow/serving ❹
```

❶ Specify the default ports
❷ Create a bin mound to load the models
❸ Specify your model
❹ Specify the docker image

By default, TensorFlow Serving is configured to create a REST and gRPC endpoint. By specifying both ports, 8500 and 8501, we expose the REST and gRPC capabilities. The docker `run` command creates a mount between a folder on the host (source) and within the container (target) filesystem. In Chapter 2, we discussed how to pass environmental variables to the docker container. To run the server in a single model configuration, you need to specify the model name

`MODEL_NAME`.

If you want to run the docker image pre-built for GPU images, you need to swap out the name of the docker image to latest GPU-build with

```
$ docker run ...
             -t tensorflow/serving:latest-gpu
```

If you have decided to run TensorFlow Serving without the Docker container, you can run it with the command

```
$ tensorflow_model_server --port=8500 \
                          --rest_api_port=8501 \
                          --model_name=my_model
\
                          --
model_base_path=/models/my_model
```

In both scenarios, you should see output on your terminal which is similar to the following

```
2019-04-26 03:51:20.304826: I
tensorflow_serving/model_servers/server.cc:82]
  Building single TensorFlow model file config:
  model_name: my_model model_base_path:
/models/my_model
2019-04-26 03:51:20.307396: I
tensorflow_serving/model_servers/server_core.cc:
```

```
461]
  Adding/updating models.
2019-04-26 03:51:20.307473: I
tensorflow_serving/model_servers/server_core.cc:
558]
  (Re-)adding model: my_model
...
2019-04-26 03:51:34.507436: I
tensorflow_serving/core/loader_harness.cc:86]
  Successfully loaded servable version {name:
my_model version: 1556250435}
2019-04-26 03:51:34.516601: I
tensorflow_serving/model_servers/server.cc:313]
  Running gRPC ModelServer at 0.0.0.0:8500 ...
[warn] getaddrinfo: address family for nodename
not supported
[evhttp_server.cc : 237] RAW: Entering the event
loop ...
2019-04-26 03:51:34.520287: I
tensorflow_serving/model_servers/server.cc:333]
  Exporting HTTP/REST API at:localhost:8501 ...
```

From the server output, you can see that the server loaded our model `my_model` successfully and that created two endpoints: One REST and one gRPC endpoint.

TensorFlow Serving makes the deployment of machine learning models extremely easy. One great advantage of serving models that way is the “hot swap” capability. If a new model is uploaded, the server’s model manager will detect the new version, unload the existing model and load the newer model for inferencing.

Let’s say you update the model and export the new model

version to the mounted folder on the host machine (if you are running with the docker setup), no configuration change is required. The model manager will detect the newer model and reload the endpoints. It will notify you about the unloading of the older model and the loading of the newer model. In your terminal, you should find messages like

```
2019-04-30 00:21:56.486988: I
tensorflow_serving/core/basic_manager.cc:739]
  Successfully reserved resources to load
servable {name: movie version: 1556583584}
2019-04-30 00:21:56.487043: I
tensorflow_serving/core/loader_harness.cc:66]
  Approving load for servable version {name:
movie version: 1556583584}
2019-04-30 00:21:56.487071: I
tensorflow_serving/core/loader_harness.cc:74]
  Loading servable version {name: movie version:
1556583584}
...
2019-04-30 00:22:08.839375: I
tensorflow_serving/core/loader_harness.cc:119]
  Unloading servable version {name: movie
version: 1556583236}
2019-04-30 00:22:10.292695: I
./tensorflow_serving/core/simple_loader.h:294]
  Calling MallocExtension_ReleaseToSystem()
after servable unload with 1262338988
2019-04-30 00:22:10.292771: I
tensorflow_serving/core/loader_harness.cc:127]
  Done unloading servable version {name: movie
version: 1556583236}
```

TensorFlow Serving will load the model with the highest version number. If you use the export methods shown earlier

in this chapter, all models will be exported in folders with the epoch timestamp as the folder name. Therefore, newer models will have a higher version number than older models.

Multi Model Configuration

You can configure TensorFlow Serving to load multiple models at the same time. To do that you need to create a configuration file to specify the models.

```
model_config_list {
  config {
    name: 'my_model'
    base_path: '/models/my_model/'
  }
  config {
    name: 'another_model'
    base_path: '/models/another_model/'
  }
}
```

The configuration file contains one or more `config` dictionaries, all listed below a `model_config_list` key.

In your Docker configuration you can mount the configuration file and load the model server with the configuration file instead of a single model.

```
$ docker run -p 8500:8500 \
             -p 8501:8501 \
             --mount
type=bind,source=/tmp/models,target=/models/my_m
odel \
             --mount
type=bind,source=/tmp/model_config,target=/model
s/model_config \ ❶
             -e MODEL_NAME=my_model \
             -t tensorflow/serving \
             --
model_config_file=/models/model_config ❷
```

❶ Mount the configuration file

❷ Specify the model configuration file

If you can TensorFlow Serving outside of a Docker container, you can point the model server to the configuration file with the additional argument `model_config_file` and the configuration will be loaded from the file

```
$ tensorflow_model_server --port=8500 \
                          --rest_api_port=8501 \
                          --
model_config_file=/models/model_config
```

CONFIGURE SPECIFIC MODEL VERSIONS

There are situations when you want to load not just the latest model version, about either all or specific

model versions. TensorFlow Serving, by default, always loads the latest model version. If you want to load all available model versions, you can extend the model configuration file with

```
  ...
  config {
    name: 'another_model'
    base_path:
'/models/another_model/'
    model_version_policy: {all:
{}}
  }
  ...
```

If you want to specify specific model versions, you can define them as well.

```
  ...
  config {
    name: 'another_model'
    base_path:
'/models/another_model/'
    model_version_policy {
      specific {
        versions: 1556250435
        versions: 1556251435
      }
    }
  }
  ...
```

You can even give the model versions labels. The

labels can extremely handy later when you want to make predictions from the models.

```
...
model_version_policy {
  specific {
    versions: 1556250435
    versions: 1556251435
  }
}
version_labels {
  key: 'stable'
  value: 1556250435
}
version_labels {
  key: 'testing'
  value: 1556251435
}
...
```

REST vs gRPC

In the configuration section, we discussed how TensorFlow Serving allows two different API types: REST and gRPC. Both protocols have their advantages and disadvantages, and we would like to take a moment to introduce both before we dive into how you can communicate with these endpoints.

Representational State Transfer

Representational State Transfer, or short REST, is a communication “protocol” used by today’s web services. It isn’t a formal protocol, but more a communication style which defines how clients communicate with web services. REST clients communicate with the server using the standard HTTP methods like GET, POST, DELETE, etc. The payloads of the requests are often encoded as XML or JSON data formats.

Google Remote Procedures Calls

Remote Procedures Calls, or short gRPC, is a remote procedure protocol developed by Google. While gRPC supports different data formats, the standard data format used with gRPC is Protobuf which we used throughout this book. gRPC provides low latency communication and smaller payloads if Protobuffers are used. gRPC was designed with APIs in mind, and the errors are more applicable to APIs. The downside is that the payloads are in a binary format which can make a quick inspection difficult.

WHICH PROTOCOL TO USE

> On the first hand, it looks very convenient to communicate with the model server over REST. The endpoints are easy to infer, the payloads can be easily inspected, and the endpoints can be tested with `curl` requests or browser tools.
>
> While gRPC APIs have a higher burden of entry initially, they can lead to significant performance improvements depending on the data structures required for the model inference. If your model experiences many requests, the reduced payload size from the Protobuf data formats can be useful.
>
> Internally, TensorFlow Serving converts JSON data structures submitted via REST to `tf.Example` data structures and this can lead to slower performance. Therefore, you might see better performance with gRPC requests if the conversion requires many type conversions (e.g. if you submit a large array with Float values).

Making predictions from the Model Server

Until now, we have entirely focused on the model server setup. In this section, we want to demonstrate how a client, e.g., a

web app, can interact with the model server. All code examples concerning REST or gRPC requests are executed on the client side.

Getting model predictions via REST

To call the model server over REST, you'll need a Python library to facilitate the communication for you. The standard library these days is `requests`. After you installed the library with

```
$ pip install requests
```

The example below showcases an example POST request.

```
>>> from requests import HTTPSession

>>> http = HTTPSession()❶
>>> url = 'http://some-domain.abc'
>>> payload = json.dumps({"key_1": "value_1"})

>>> r = http.request('post', url, payload)❷
>>> r.json()❸
{'data': ...}
```

Set up a connection pool
❶ Submit the request
❷ View the http response
❸

URL STRUCTURE

The url for your http request to the model server contains information which model and which version you would like to infer.

```
http://{HOST}:{PORT}/v1/models/{MODEL_NAME}
```

HOST

The host is the IP address or domain name of your model server. If you run your model server on the same machine where you run your client code, you can set the host to `localhost`

PORT

You'll need to specify the port in your request URL. The standard port for the REST API is 8501. If this conflicts with other services in your service ecosystem, you can change the port in your server arguments during the startup of the server.

MODEL_NAME

The model name needs to match the name of your model when you either configured your model configuration or when you started up the model server

```
http://{HOST}:{PORT}/v1/models/{MODEL_NAME}
[/versions/${MODEL_VERSION}]
```

MODEL_VERSION

If you want to make predictions from a specific model version, you'll need to extend the URL with the version identifier. Earlier we talked about *version labels*. The labels become very handy to specify an exact version.

If you want to submit a data example to a classification or regression model, you'll need to extend the URL with the `classify` or `regress` argument.

```
http://{HOST}:{PORT}/v1/models/{MODEL_NAME}
[/versions/${MODEL_VERSION}]:(classify|regress)
```

PAYLOADS

With the URLs in place, let's discuss the request payloads. TensorFlow Serving expects the input data as a JSON data structure shown in the following example.

```
{
  "signature_name": <string>,
  "instances": <value>
}
```

The `signature_name` is not required. If it isn't specified, the model server will infer the model graph signed with the default `serving` label.

The input data is expected either as a list of objects or as a list of input values. If you want to submit multiple data samples, you can submit them as a list under the `instances` key.

```
{
  "signature_name": <string>,
  "inputs": <value>
}
```

If you want to submit one data example for the inference, you can use the `inputs` and list all input values as a list. One of the keys, `instances` and `inputs`, have to be present, but never both at the same time.

EXAMPLE

With the following example, you can request a model inference. In our example, we only submit one data example for the inference, but your list could easily contain more examples.

```
>>> import json
>>> from requests import HTTPSession

>>> def rest_request(text):
>>>     url =
'http://localhost:8501/v1/models/movie:predict'
❶
>>>     payload = json.dumps({"instances":
[text]}) ❷
>>>     response = http.request('post', url,
payload)
>>>     return response

>>> rs_rest = rest_request(text="classify my
text")
>>> rs_rest.json()
```

❶ Exchange `localhost` with an IP address if the server is not running on the same machine

❷ Add more examples to the `instance` list if you want to infer more samples.

Inferring TensorFlow Serving via gRPC

If you want to infer the model over gRPC, the steps are slightly different to the REST API requests.

First, you establish a gRPC `channel`. The channel provides the connection to the gRPC server at a given host address and over a given port. If you require a secure connection, you need to establish a secure channel at this point. Once the channel is established, you'll create a `stub`. A stub is a local object which replicates the available methods from the server.

```
import grpc
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import
prediction_service_pb2_grpc
import tensorflow as tf

def create_grpc_stub(host, port='8500'):
    hostport = f'{host}:{port}'
    channel = grpc.insecure_channel(hostport)
    stub =
prediction_service_pb2_grpc.PredictionServiceStub
(channel)
    return stub
```

Once the gRPC stub is created, we can set the model and the signature to access predictions from the correct model and submit our data for the inference.

```
def grpc_request(stub, data_sample,
```

```
    model_name='my_model',
    signature_name='classification'):
        request = predict_pb2.PredictRequest()
        request.model_spec.name = model_name
        request.model_spec.signature_name =
    signature_name


    request.inputs['inputs'].CopyFrom(tf.make_tensor
    _proto(data_sample, shape=[1,1])) ❶
        result_future = stub.Predict.future(request,
    10) ❷
        return result_future
```

❶ `inputs` is the name of the input neurons of our neural network

❷ 10 is the max time before the function times out

With the two function now available, we can infer our example data sets with the two function calls

```
stub = create_grpc_stub(host, port='8500')
rs_grpc = grpc_request(stub, data)
```

GETTING PREDICTIONS FROM CLASSIFICATION AND REGRESSION MODELS

If you are interested in making predictions from classification and regression models, you can do that with the gRPC API.

If you would like to get predictions from a classification model, you need to swap out the following lines

```
from tensorflow_serving.apis import predict_pb2
...
request = predict_pb2.PredictRequest()
```

with

```
from tensorflow_serving.apis import
classification_pb2
...
request =
classification_pb2.ClassificationRequest()
```

If you want to get predictions from a regression model, you can use the following imports

```
from tensorflow_serving.apis import
regression_pb2
...
regression_pb2.RegressionRequest()
```

PAYLOADS

gRPC API uses ProtoBuffers as the data structure for the API request. By using ProtoBuffer payloads, the API requests are

compressed and therefore use less bandwidth. Also, depending on the model input data structure, you might experience faster inferences as with the REST endpoints. The performance difference is explained by the fact that the submitted JSON data will be converted to a `tf.Example` data structure. This conversion can slow down the model server inference, and you might encounter a slower inference performance than in the gRPC API case.

Your data submitted to the gRPC endpoints needs to be converted to the ProtoBuffer data structure. TensorFlow provides you a handy utility function to perform the conversion called `tf.make_tensor_proto`. `make_tensor_proto` allows various data formats, including scalars, lists, NumPy scalars, and NumPy arrays. The function will then convert the given Python or NumPy data structures to the ProtoBuffer format for the inference.

Model A/B Testing with TensorFlow Serving

A/B testing is an excellent methodology to either test different models in real life situations or a way to phase in newer models and expose them to a small number of users before

directing all model inferences to the newer model.

We discussed earlier that you could configure TensorFlow Serving to load multiple model versions and then specify the model version in your REST request URL or gRPC specifications.

TensorFlow Serving doesn't support A/B Testing from the server-side, but with a little tweak to our request URL, we can support random A/B testing from the client-side.

```
from random import random ❶

def get_rest_url(model_name, host='localhost',
port='8501',
                 verb='predict', version=None):
    url = f"http://{host}:
{port}/v1/models/{model_name}/"
    if version:
        url += f"versions/{version}"
    url += f":{verb}"
    return url

...

# submit 10% of all request from this client to
version 1
# 90% of the request should go to the default
models
threshold = 0.1
version = 1 if random() < threshold else None ❷
url =
get_rest_url(model_name='complaints_classificati
```

```
    on', version=version)
```

The `random` library will help us to pick a model
❶ If version == None, TensorFlow Serving will infer with the
❷ default version

As you can see, randomly changing the request URL for our model inference (in our REST API example), can provide you some basic A/B testing functionality. If you would like to extend the capabilities by performing the random routing of the model inference on the server side, we highly recommend routing tools like *Istio* [4] for that purpose. Originally designed for web traffic, the tool can be used to route traffic to specific models. You can phase in models, perform A/B tests or create policies for data routed to specific models.

When you perform A/B tests with your models, it is often useful to request information about the model from the model server. In the following section, we will explain how you can request the metadata information from TensorFlow Serving.

Requesting Model Meta Data from the Model Server

At the beginning of the book we laid out the model life cycle

and how we want to automate the machine learning life cycle. A critical component of the continuous life cycle is generating accuracy or general performance feedback about your model versions. We will deep dive into how to generate these feedback loops in a later chapter, but for now, imagine that your model classifies some data, e.g., the sentiment of the text, and then asks the user to rate the prediction. The information of whether a model predicted something correctly or incorrectly is precious to improve future model versions, but it is only useful if we know which model version has performed the prediction.

The metadata provided by the model server will contain the information to annotate your feedback loops.

REST Requests for Model Meta Data

Requesting model meta information is straight forward with TensorFlow Serving. TensorFlow Serving provides you an endpoint for model meta information.

```
http://{HOST}:{PORT}/v1/models/{MODEL_NAME}
[/versions/{MODEL_VERSION}]/metadata
```

Similar to the REST API inference requests we discussed earlier, you have the option to specify the model version in the request URL, or if you don't specify it, the model server will provide the information about the default model.

```
import json
from requests import HTTPSession

def metadata_rest_request(model_name,
host='localhost',
                          port='8501',
version=None):
    url = f"http://{host}:
{port}/v1/models/{model_name}/"
    if version:
        url += f"versions/{version}"
    url += f"/metadata"

    http = HTTPSession()
    response = http.request('get', url)
    return response
```

With one REST request, the model server will return the model specifications as a `model_spec` dictionary and the model siganture definitions as a `metadata` dictionary.

```
{
        "model_spec": {
                "name":
```

```
"complaints_classification",
                "signature_name": "",
                "version": "1556583584"
        },
        "metadata": {
                "signature_def": {
                        "signature_def": {

"classification": {

"inputs": {

"inputs": {

"dtype": "DT_STRING",

"tensor_shape": {
                ...
```

You can then attach the model meta information to the information you want to store about your model performance which can be analyzed at a later point in time.

gRPC Requests for Model Meta Data

Requesting model meta information is almost as easy as we have seen it in the REST API case. In the gRPC case, you file a `GetModelMetadataRequest`, add the model name to the specifications and submit the request via the

`GetModelMetadata` method of the `stub`.

```
from tensorflow_serving.apis import
get_model_metadata_pb2

def get_model_version(model_name, stub):
    request =
get_model_metadata_pb2.GetModelMetadataRequest()
    request.model_spec.name = model_name

request.metadata_field.append("signature_def")
    response = stub.GetModelMetadata(request, 5)
    return response.model_spec
```

```
>>> model_name = 'complaints_classification'
>>> stub = create_grpc_stub('localhost')
>>> get_model_version(model_name, stub)
```

```
name: "complaints_classification"
version {
  value: 1556583584
}
```

The gRPC response contains `ModelSpec` object which contains the version number of the loaded model.

More interesting is the use-case of obtaining the model signature information of the loaded models. With almost the same request functions we can determine the model meta information. The only difference is that we don’t access the

`model_spec` attribute of the response object, but the `metadata`. The information needs to be serialized to be human-readable; therefore we are using `SerializeToString` to convert the ProtoBuffer information.

```
from tensorflow_serving.apis import
get_model_metadata_pb2

def get_model_meta(model_name, stub):
    request =
get_model_metadata_pb2.GetModelMetadataRequest()
    request.model_spec.name = model_name

request.metadata_field.append("signature_def")
    response = stub.GetModelMetadata(request, 5)
    meta =  response.metadata['signature_def']
    return meta.SerializeToString().decode("utf-
8", 'ignore')

>>> model_name = 'complaints_classification'
>>> stub = create_grpc_stub('localhost')
>>> meta = get_model_meta(model_name, stub)

>>> print(meta)
type.googleapis.com/tensorflow.serving.Signature
DefMap
serving_default
complaints_classification_input
        input_1:0
               2@
complaints_classification_output(
dense_1/Softmax:0
              tensorflow/serving/predict
```

Batching Inference Requests

Batching inference requests is one of the most powerful features of TensorFlow Serving. During model training, batching accelerates our training because we can parallelize the computation of our training samples. At the same time, we can also use the computation hardware efficiently if we match the memory requirements of our batches with the available memory of the GPU.

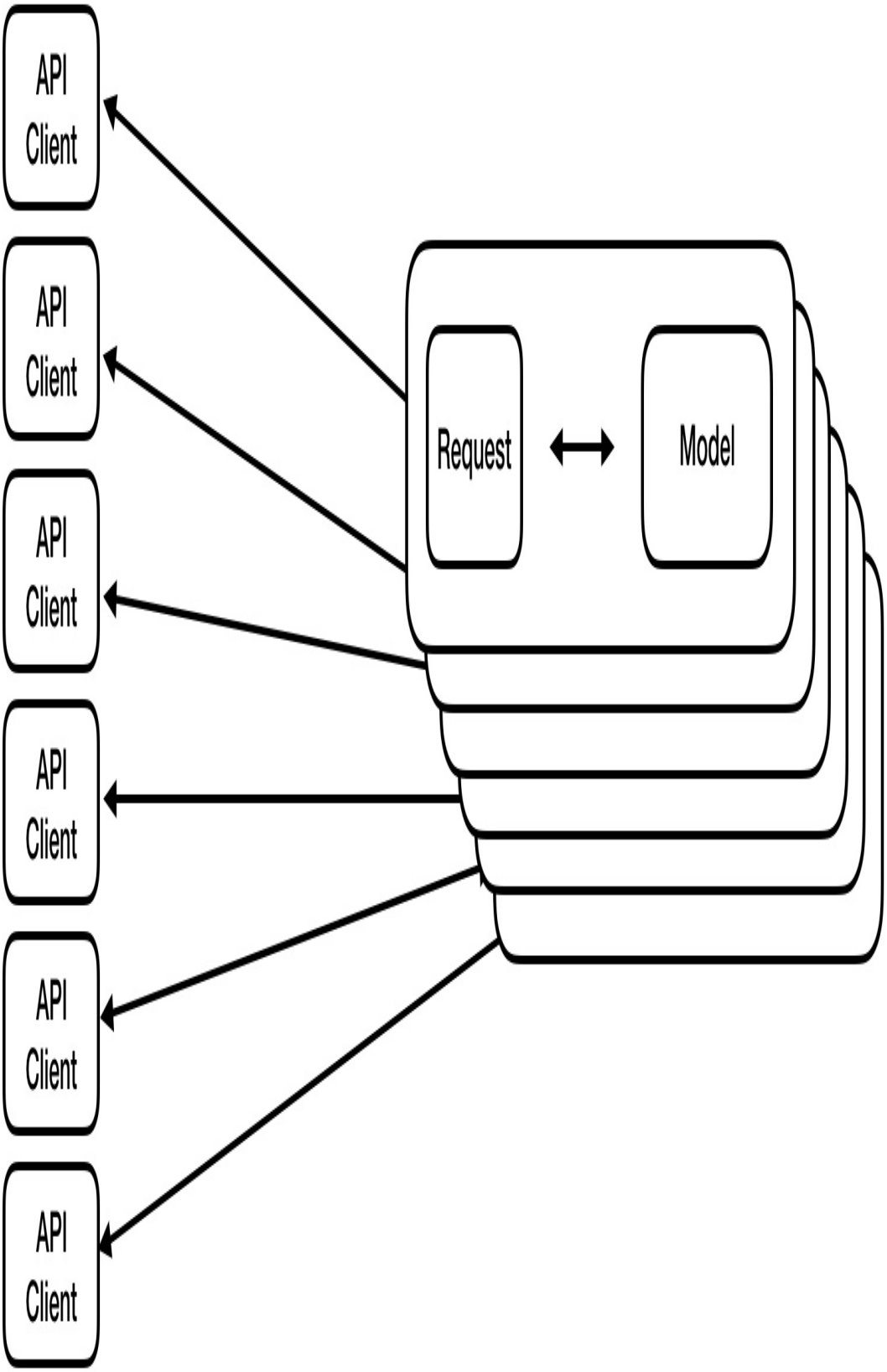

API Client
API Client
API Client
API Client
API Client
API Client
Request
Model

Figure 4-2. Overview of the TensorFlow Serving Without Batching

As shown in [Link to Come], if you run TensorFlow Serving without the batching enabled, every client request with one or more data samples creates an inference of the model regardless of whether the memory is optimally used or not.

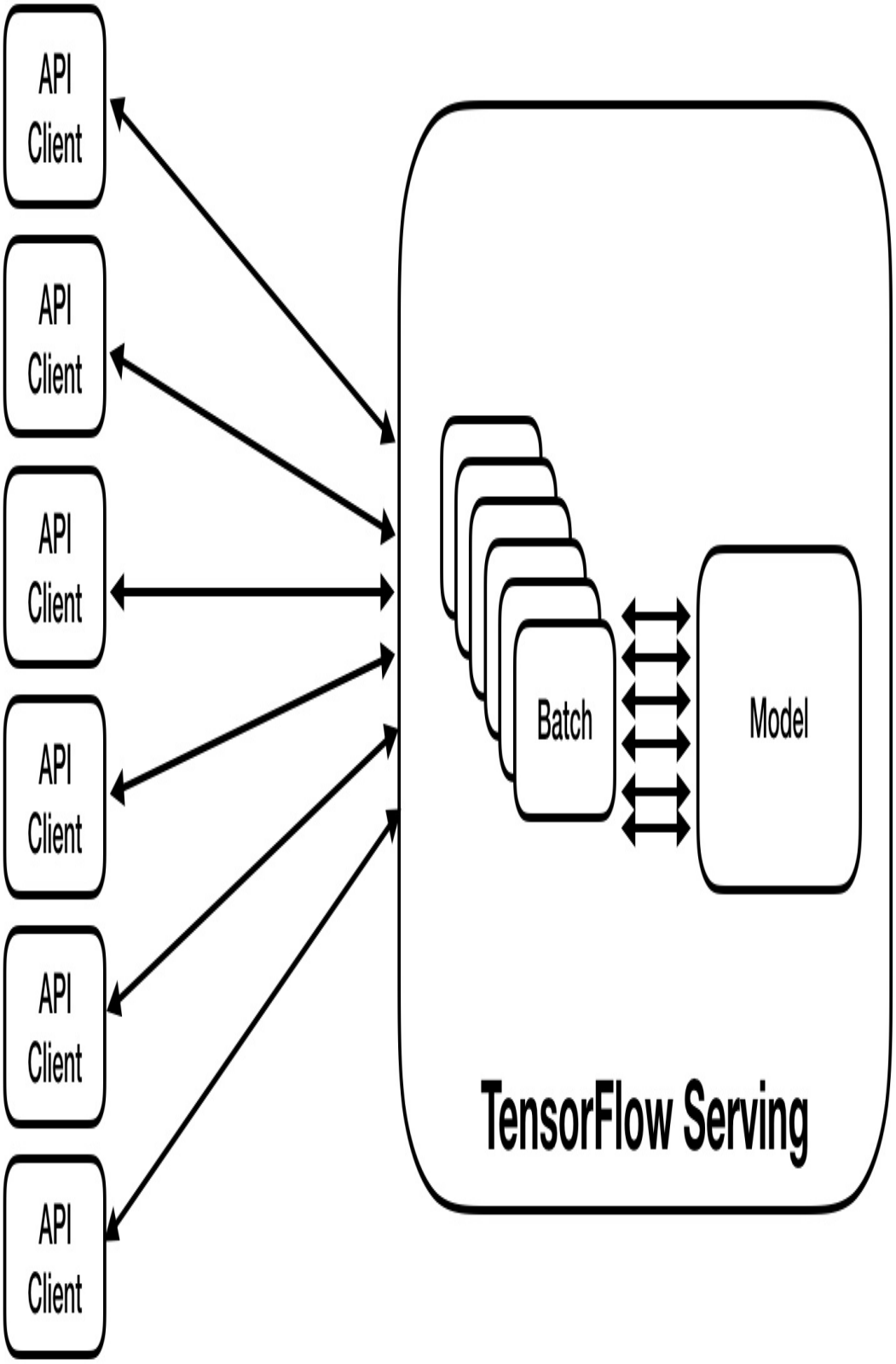

API
Client
API
Client
API
Client
API
Client
API
Client
API
Client
Batch
Model
TensorFlow Serving

Figure 4-3. Overview of the TensorFlow Serving Batching

As shown in [Link to Come], multiple clients can request model predictions and the model server batches the different client requests into one “block” to compute. Each request inferred through this batching step might take a bit longer than a single request. However, imagine a large number of individual requests hitting the server. In this case, you can process a more significant number of requests if processed as a batch.

Configure Batch Inferences

Batching inferences needs to be enabled for TensorFlow Serving and then configured for your use-case. You have five configuration options:

max_batch_size

> This parameter controls the batch size. Large batch sizes will increase the request latency and can lead to an exhausting of the GPU memory. Small batch size lose the benefit of the optimal computation resource usage.

batch_timeout_micros

> This parameter sets the maximum time to wait to fill a

batch. This parameter is handy to cap the latency for inference requests.

num_batch_threads

The number of thread configures how many CPU or GPU cores can be used in parallel.

max_enqueued_batches

This parameter sets the maximum number of batches queued for inferences. This configuration is beneficial to avoid an unreasonable backlog of requests. If the maximum number is reached, requests will be returned with an error instead of being queued.

pad_variable_length_inputs

This boolean parameter determines if input tensors with variable lengths will be padded to the same lengths for all input tensors.

As you can imagine, setting the parameters for the optimal batching requires some tuning and is application dependent. If you run online inference, you should aim for limiting the latency. It is often recommended to set `batch_timeout_micros` initially to zero and tune the timeout towards 10000 microseconds. In contrast, batch requests will benefit from longer timeouts (milli-seconds to a second) to constantly use the batch size for optimal

performance. TensorFlow Serving will make predictions on the batch when either the `max_batch_size` or the timeout is reached.

If you configure TensorFlow Serving for CPU inferences, set `num_batch_threads` to the number of CPU cores. If you configure a GPU setup, tune `max_batch_size` to get an optimal utilization of the GPU memory. While you tune your configuration, make sure that you set `max_enqueued_batches` to a huge number to avoid that some requests will be returned early without proper inference.

You can set the parameters in a text file as shown in the following example. In our example, we call the configuration file `batching_parameters.txt`.

```
max_batch_size { value: 32 }
batch_timeout_micros { value: 5000 }
pad_variable_length_inputs: true
```

If you want to enable batching, you need to pass two additional parameters to the Docker container running TensorFlow Serving. Set `enable_batching` to true to enable batching and set `batching_parameters_file` to the absolute path of the batching configuration file inside of

the container. Please keep in mind that you have to mount the additional folder with the configuration file if it isn't located in the same folder as the model versions.

Here is a complete example of the `docker run` command to start the TensorFlow Serving Docker container with batching enabled. The parameters will then be passed to the TensorFlow Serving instance.

```
docker run -p 8500:8500 \
           -p 8501:8501 \
           --mount
type=bind,source=/path/to/models,target=/models/
my_model \
           --mount
type=bind,source=/path/to/batch_config,target=/s
erver_config \
           -e MODEL_NAME=my_model -t
tensorflow/serving \
           --enable_batching=true
           --
batching_parameters_file=/server_config/batching
_parameters.txt
```

As explained earlier, the configuration of the batching will require additional tuning, but the performance gains should make up for the initial setup. We highly recommend enabling this TensorFlow Serving feature. It is especially useful for offline/batch processes to infer a large number of data samples.

Other TensorFlow Serving Optimizations

TensorFlow Serving comes with a variety of additional optimization features. Additional feature flags are:

--file_system_poll_wait_seconds=1

> TensorFlow Serving will poll if a new model version is available. You can disable the feature by setting it to `-1` or if you only want to load the model once and never update it, you can set it to `0`. The parameter expects an integer value.

--tensorflow_session_parallelism=0

> TensorFlow Serving will automatically determine how many threads to use for a TensorFlow session. In case, you want to set the number of a thread manually, you can overwrite it by setting this parameter to any positive integer value.

--tensorflow_intra_op_parallelism=0

> This parameter sets the number of cores being used for running TensorFlow Serving. The number of available threads determines how many operations will be parallelized. If the value is zero, all available cores will be used.

--tensorflow_inter_op_parallelism=0

> This parameter sets the number of available threads in a pool to execute TensorFlow ops. This is useful to maximize the execution of independent operations in a TensorFlow graph. If the value is set to zero, all available cores will be used and one thread per core allocated.

Similar to our earlier examples, you can pass the configuration parameter to the `docker run` command as shown in the following example:

```
docker run -p 8500:8500 \
           -p 8501:8501 \
           --mount
type=bind,source=/path/to/models,target=/models/
my_model \
           -e MODEL_NAME=my_model -t
tensorflow/serving \
           --tensorflow_intra_op_parallelism=4 \
           --tensorflow_inter_op_parallelism=4 \
           --file_system_poll_wait_seconds=10 \
           --tensorflow_session_parallelism=2
```

Using TensorRT with TensorFlow Serving

If you are running computationally intensive deep learning models on an NVidia GPU, you have an additional way of optimizing your model server. NVidia provides a library called TensorRT which optimizes the inference of deep learning models by reducing the precision of the numerical

representations of the network weights and biases. TensorRT supports `int8` and `float16` representations. The reduced precision will lower the inference latency of the model.

After your model is trained, you need to optimize the model with TensorRT's own optimizer [5] or with `saved_model_cli`. The optimized model can then be loaded into TensorFlow Serving. At the time of writing this chapter, TensorRT was limited to some NVidia products including Tesla V100 and P4.

First, we'll convert our deep learning model with `saved_model_cli`

```
$ saved_model_cli convert --dir saved_models/
                          --output_dir trt-
savedmodel/
                          --tag_set serve
tensorrt
```

and then load the model in our GPU setup of TensorFlow Serving

```
$ docker run --runtime=nvidia \
             -p 8500:8500 \
             -p 8501:8501 \
             --mount
type=bind,source=/path/to/models,target=/models/
```

```
my_model \
                -e MODEL_NAME=my_model
                -t tensorflow/serving:latest-gpu
```

If you are inferring on NVidia GPUs and your hardware is supported by TensorRT, switching to TensorRT can be an excellent way to lower your inference latencies further.

TensorFlow Serving Alternatives

TensorFlow Serving is a great way of deploying machine learning models. With the TensorFlow Estimators and Keras models, a large variety of machine learning concepts are covered. If you would like to deploy a legacy model or your machine learning framework of choice isn't TensorFlow/Keras, here are a couple of options for you.

Seldon

The UK start-up Seldon provides a variety of open source tools to manage model life cycles, and one of the core products is *Seldon Core* [6]. Seldon Core provides you a toolbox to wrap your models in a Docker image which is then deployed via Seldon in a Kubernetes Cluster.

At the time of writing this chapter, Seldon supported machine learning models written in Python, Java, NodeJS, and R.

Seldon comes with its own ecosystem which allows building the preprocessing into its own Docker images which are deployed in conjunction with the deployment images. It also provides its Routing service which allows you to perform A/B test or multiarm bandit experiments.

Seldon is highly integrated with the KubeFlow environment and, similar to TensorFlow Serving, is a way to deploy models with KubeFlow on Kubernetes.

GraphPipe

GraphPipe [7] is another way of deploying TensorFlow and non-TensorFlow models. Oracle drives the open source project. It allows you to deploy not just TensorFlow (inc. Keras) models, but also Caffe2 models and all machine learning models which can be converted to the ONNX format [8]. Through the ONNX format you can deploy PyTorch models with GraphPipe.

Besides providing a model server for TensorFlow, PyTorch, etc., GraphPipe also provides client implementation for

programming languages like Python, Java and Go.

Simple TensorFlow Serving

Simple TensorFlow Serving [9] supports more than just TensorFlow models. The current list of supported model frameworks includes ONNX, Scikit-learn, XGBoost, PMML, and H2O. It supports multiple models, inferences on GPUs and client code for a variety of languages.

One significant aspect of Simple TensorFlow Serving is that it supports authentication and encrypted connections to the model server. Authentication is currently not a feature of TensorFlow Serving and SSL/TLS supports requires a custom build of TensorFlow Serving.

MLflow

MLflow [10] supports the deployment of machine learning models, but it is only one aspect of the tool created by DataBricks. MLflow is designed to manage model experiments through MLflow Tracking. The tool has a model server built-in which provides REST API endpoints for the models managed through MLflow.

MLflow also provides interfaces to directly deploy the models from MLflow to Microsoft's AzureML platform and Amazon's Web Service SageMaker.

Deploying with Cloud Providers

All model server solutions we have discussed up to this point have to be installed and managed by you. However, all primary cloud providers, Google Cloud, Amazon Web Services and Microsoft Azure, offer machine learning products including the hosting of machine learning models.

In this section, we would like to walk you through one deployment option with a cloud provider.

Use Cases

Managed cloud deployments of machine learning models are a good alternative to running your model server instances if you want to deploy a model seamlessly and don't want to worry about the scaling of the model deployment. All cloud providers offer deployment options with the ability to scale depending on the number of inference requests.

However, the flexibility of your model deployment comes at a cost. Managed services provide effortless deployments, but they cost a premium. For example, two model versions running full-time (requires two computation nodes) are more expensive than a comparable compute instance which is running a TensorFlow Serving instance. Another downside of managed deployments are the limitations of the products. Some cloud providers require that you deploy via their own Software Development Kits, others have limits on the node size and how much memory your model can take up. These limitations can be a severe restriction for sizeable deep learning models, especially if the models contain very many layers or a layer contains language model information.

Example Deployment with Google Cloud Platforms

In this section, we will guide you through an example deployment with Google Cloud's AI Platform. Let's start with the model deployment, and later we'll explain how you can get predictions from the deployed model from your application client.

MODEL DEPLOYMENT

The deployment consists of three steps:

- Making the model accessible on Google Cloud
- Create a new model instance with Google Cloud's AI Platform
- Create a new version with the model instance

The deployment starts with uploading your exported TensorFlow/Keras model to a storage bucket. As shown in [Link to Come], you need to upload the entire exported model. Once the upload of the model is done, please copy the complete path of the storage location.

Figure 4-4. Uploading the Trained Model to a Cloud Storage

Once you have uploaded your machine learning model, head

over to the AI Platform of Google Cloud Platform to set up your machine learning model for deployment. If it is the first time that you use the AI Platform in your GCP project, you'll need to enable the API. The automated startup process by Google Cloud can take a few minutes.

When you create a new model, you need to give the model a unique identifier. Once you have created the identifier and created an optional project description, continue with the setup by clicking Create.

Figure 4-5. Creating a new Model Instance

Once the new model is registered, you can create a new model version within the model. To do that, please click on [Link to Come] in the overflow menu.

Figure 4-6. Creating a New Model Version

When you create a new model version, you configure a compute instance which is running your model. Google Cloud gives you a variety of configuration options. Important is the `version name` since you'll reference the `version name` later in the client setup. Please set the `Model URI` to the storage path you saved in the earlier step.

Google Cloud AI Platform supports a variety of machine learning frameworks including XGBoost and SciKit-Learn.

Create version

To create a new version of your model, make necessary adjustments to your saved model file before exporting and store your exported model in Cloud Storage. Learn more

Name

v1

Name cannot be changed, is case sensitive, must start with a letter, and may only contain letters, numbers, and underscores. 2 / 128

Description

demo version

Python version

3.5

Select the Python version you used to train the model

⚠ Model version with Python 3.0 and beyond can't be used for batch prediction jobs. Online prediction still works.

Framework

TensorFlow

Framework version

1.13.1

ML runtime version

1.13

Machine type

Single core CPU

Model URI *

gs:// demo_models/1549767607/ BROWSE

Cloud Storage path to the entire SavedModel directory. Learn more

Figure 4-7. Setting up the Instance Details

Google Cloud Platform also lets you configure how your model instance should scale in case your model experiences a large number of inference requests. As [Link to Come] shows, you can set the scaling behavior. You have two options:

- Manual scaling
- Auto-scaling

While the manual scaling gives you the option for setting the exact number of nodes available for the inferences of your model version, auto-scaling will give you the chance spin up and down the number of available nodes. If your nodes don't experience any requests, the number of nodes could even drop to zero. Please note that if the autoscaling is dropping the number of nodes to zero, it will take some time to re-instantiate your model version with the next request hitting the model version endpoint. Also if you run inference nodes in the autoscaling mode, you'll be billed in minute intervals with a minimum of 10 min. That means that one request will cost you at least 10 min of compute time.

Online prediction deployment

Scaling

Manual scaling

Choosing manual scaling requires you to specify the number of nodes in the input field below. The number of nodes you specify will always be ready, and you will be charged continuously for them. You should avoid manual scaling unless the number of requests your model receives inherently fluctuates faster than the automatic scaling can keep up. Learn more about nodes and prediction cost

Number of nodes

1

The number of nodes to allocate for this model

SAVE CLEAR CANCEL

Figure 4-8. Setting up the Scaling Details of the Model Instance

Once the entire model version is configured, Google Cloud spins up the instances for you. If everything is ready for model inferences, you see a green check icon next to the version name as shown in [Link to Come].

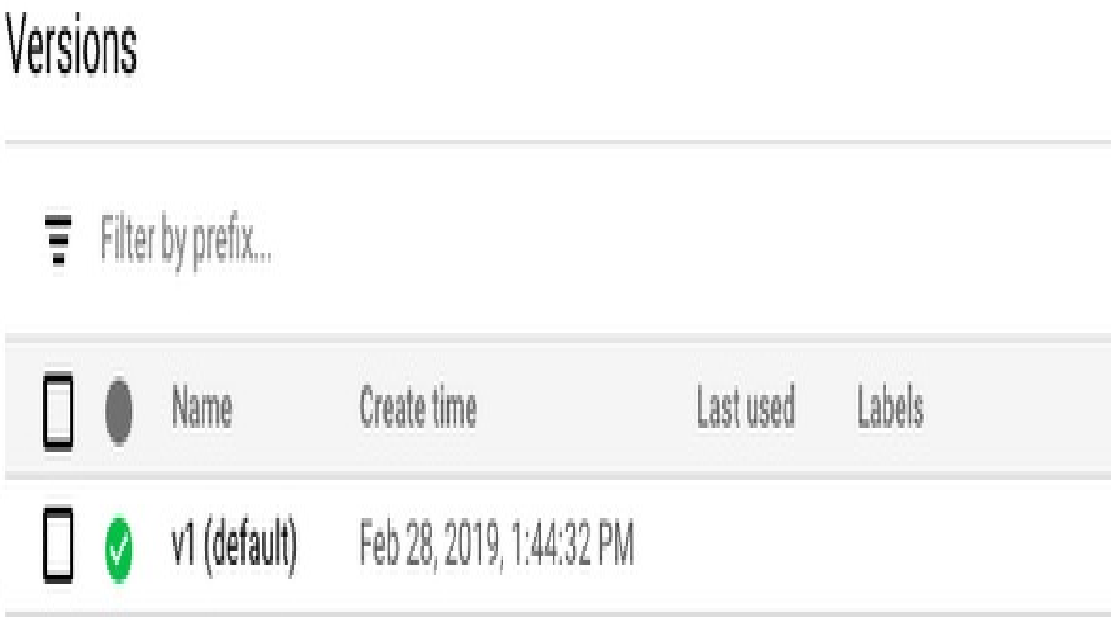


Figure 4-9. Completing the Deployment with a new Version available

You can run multiple model versions simultaneously. In the control panel of the model version, you set one version as the default version and any inference request without a version specified will be routed to the designated “default version”. Just note that each model version will be hosted on an individual node and accumulate Google Cloud Platform costs.

MODEL INFERENCE

Since TensorFlow Serving is battle tested at Google and used heavily internally, it is also used behind the scenes at Google Cloud Platform. You'll notice that the AI Platform isn't just using the same model export format as we have seen with our TensorFlow Serving instances, but the payloads have the same data structure as we have seen before.

The only significant difference is the API connection. As you'll see in this section, you'll connect to the model version via the GCP API which is handling the request authentication.

To connect with the Google Cloud API, you'll need to install the library `google-api-python-client` with

```
$ pip install google-api-python-client==1.7.8
```

All Google services can be connected via a service object. The helper function in the following code snippet highlights how to create the service object. The Google API client takes a `service name` and a `service version` and returns an object which provides all API functionalities via methods of the returned object.

```
import googleapiclient.discovery

def _connect_service():
    kwargs = {'serviceName': 'ml', 'version':
'v1'}
    return
googleapiclient.discovery.build(**kwargs)
```

Similar to our earlier REST and gRPC examples, we nest our inference data under a fixed `instances` key which carries a list of input dictionaries. We have created a little helper function to generate the payloads. This function can contain any preprocessing if you need to modify your input data before the inference.

```
def _generate_payload(sentence):
    return {"instances": [{"sentence":
sentence}]}
```

With the service object created on the client side and the payload generated, it's time to request the prediction from the Google Cloud hosted machine learning model.

The service object of the AI Platform service contains a predict method which accepts a `name` and a `body`. The name is a path string containing your Google Cloud Platform project

name, your model name and if you want to make predictions with a specific model version, your version name. If you don’t specify a version number, the default model version will be used for the model inference. The body contains the inference data structure we generated earlier.

```
project = 'yourGCPProjectName'
model_name = 'demo_model'
version_name = 'v1'
request = service.projects().predict(

name=f'projects/{project}/models/{model_name}/ve
rsions/{version_name}',
  body=_generate_payload(sentence)
)
response = request.execute()
```

The Google Cloud AI Platform response contains the predict scores for the different categories similar to a REST response from a TensorFlow Serving instance.

```
{'predictions': [{'label': [
   0.9000182151794434,
   0.02840868942439556,
   0.009750653058290482,
   0.06182243302464485]
   }]
}
```

Summary

In this chapter, we discussed how to setup TensorFlow Serving to deploy machine learning models and why a model server is a more scalable option than deploying machine learning models through a Flask web application. We stepped through the installation and configuration steps, introduced the two main communication option, REST and gRPC, and briefly discussed the advantages and disadvantages of both communication protocols.

Futhermore, we explained some of the great benefits of TensorFlow Serving, including the batching of model requests and how to obtain meta information about the different model versions. We also discussed how to set up a quick A/B test setup with TensorFlow Serving.

We closed this chapter with a brief introduction of a managed cloud service, using Google Cloud AI Platform as an example. This provides you the ability to deploy machine learning models without managing your own server instances.

---

1 *https://www.tensorflow.org/about/case-studies*

2 *https://github.com/NVIDIA/nvidia-docker#quick-start*

3 *https://www.tensorflow.org/tfx/serving/setup#building_from_source*

4 *https://istio.io/*

5 *https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html*

6 *https://www.seldon.io/open-source/*

7 *https://oracle.github.io/graphpipe/*

8 ONNX is a way of describing machine learning models. *https://onnx.ai/*

9 *https://stfs.readthedocs.io/*

10 *https://mlflow.org/*

About the Authors

Hannes Hapke is a VP of Engineering at Caravel, a machine learning company providing novel personalization products for the retail industry. Prior to joining Caravel, Hannes was a Senior Data Science Engineer at Cambia Health Solutions, a health solutions provider for 2.6 million people and a Machine Learning Engineer at Talentpair, Inc. where he developed novel deep learning model for recruiting companies. Hannes co-founded a renewable energy startup which applied deep learning to detect homes would be optimal candidates for solar power.

Additionally, Hannes has co-authored a publication about natural language processing and deep learning and presented at various conferences about deep learning and Python.

Catherine Nelson is a Senior Data Scientist for Concur Labs at SAP Concur, where she explores innovative ways to use machine learning to improve the experience of a business traveller. She is particularly interested in privacy-preserving ML and applying deep learning to enterprise data. In her previous career as a geophysicist she studied ancient volcanoes and explored for oil in Greenland. Catherine has a PhD in geophysics from Durham University and a Masters of Earth

Sciences from Oxford University.