TFX: A TensorFlow-Based Production-Scale Machine Learning Platform

Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, Martin Zinkevich
Google Inc.*

ABSTRACT

Creating and maintaining a platform for reliably producing and deploying machine learning models requires careful orchestration of many components—a learner for generating models based on training data, modules for analyzing and validating both data as well as models, and finally infrastructure for serving models in production. This becomes particularly challenging when data changes over time and fresh models need to be produced continuously. Unfortunately, such orchestration is often done ad hoc using glue code and custom scripts developed by individual teams for specific use cases, leading to duplicated effort and fragile systems with high technical debt.

We present TensorFlow Extended (TFX), a TensorFlow-based general-purpose machine learning platform implemented at Google. By integrating the aforementioned components into one platform, we were able to standardize the components, simplify the platform configuration, and reduce the time to production from the order of months to weeks, while providing platform stability that minimizes disruptions.

We present the case study of one deployment of TFX in the Google Play app store, where the machine learning models are refreshed continuously as new data arrive. Deploying TFX led to reduced custom code, faster experiment cycles, and a 2% increase in app installs resulting from improved data and model analysis.

KEYWORDS

large-scale machine learning; end-to-end platform; continuous training

1 INTRODUCTION

It is hard to overemphasize the importance of machine learning in modern computing. More and more organizations adopt machine learning as a tool to gain knowledge from data across a broad spectrum of use cases and products, ranging from recommender systems [6, 7], to clickthrough rate prediction for advertising [13, 15], and even the protection of endangered species [5].

The conceptual workflow of applying machine learning to a specific use case is simple: at the training phase, a *learner* takes a dataset as input and emits a learned model; at the inference phase, the *model* takes features as input and emits predictions. However, the actual workflow becomes more complex when machine learning needs to be deployed in production. In this case, additional components are required that, together with the learner and model, comprise a *machine learning platform*. The components provide automation to deal with a diverse range of failures that can happen in production and to ensure that model training and serving happen reliably. Building this type of automation is non-trivial, and it becomes even more challenging when we consider the following complications:

- Building *one* machine learning platform for *many* different learning tasks: Products can have substantially different needs in terms of data representation, storage infrastructure, and machine learning tasks. The machine learning platform must be generic enough to handle the most common set of learning tasks as well as be extensible to support one-off atypical use-cases.
- *Continuous training and serving*: The platform has to support the case of training a single model over fixed data, but also the case of generating and serving up-to-date models through continuous training over evolving data (e.g., a moving window over the latest n days of a log stream).
- *Human-in-the-loop*: The machine learning platform needs to expose simple user interfaces to make it easy for engineers to deploy and monitor the platform with minimal configuration. Furthermore, it also needs to help users with various levels of machine-learning expertise understand and analyze their data and models.
- *Production-level reliability and scalability*: The platform needs to be resilient to disruptions from inconsistent data, software, user configurations, and failures in the underlying execution environment. In addition, the platform must scale gracefully to the high data volume that is common in training, and also to increases in the production traffic to the serving system.

*Corresponding authors: Heng-Tze Cheng, Clemens Mewald, Neoklis Polyzotis, and Steven Euijong Whang: {hengtze,clemensm,npolyzotis,swhang}@google.com

KDD'17, August 13–17, 2017, Halifax, NS, Canada.
 978-1-4503-4887-4/17/08.
DOI: http://dx.doi.org/10.1145/3097983.3098021

Having this type of platform enables teams to easily deploy machine learning in production for a wide range of products, ensures best practices for different components of the platform, and limits the technical debt arising from one-off implementations that cannot be reused in different contexts.

This paper presents the anatomy of end-to-end machine learning platforms and introduces TensorFlow Extended (TFX), one implementation of such a platform that we built at Google to address the aforementioned challenges. We describe the key platform components and the salient points behind their design and functionality. We also present a case study of deploying the platform in Google Play, a commercial mobile app store with over one billion active users and over one million apps, and discuss the lessons that we learned in this process. These lessons reflect best practices for machine-learning platforms in a diverse set of contexts and are thus of general interest to researchers and practitioners in the field.

2 PLATFORM OVERVIEW

2.1 Background and Related Work

Prior art has addressed a subset of the challenges in deploying machine learning in production. Related work has reported that the learning algorithm is only one component of a machine learning platform that represents a small fraction of the code [19, 20]. Data and model parallelism require distributed systems and orchestration that exceed capabilities of many single-machine solutions [12, 16]. Beyond simply stitching together components, a machine learning pipeline also needs to be simple to set up [16], maybe even support automated pipeline construction [20]. Once a team can train multiple models it needs to keep track of their experiment history in a centralized database [21]. Ideally, the platform automatically surveys different machine learning techniques and suggests the best solution, allowing even non-experts access to machine learning [10]. However, putting together several disjoint components to do the job can result in significant technical debt in forms of hard-to-maintain glue code, hidden dependencies, feedback loops, etc. [19].

2.2 Platform Design and Anatomy

In this paper we expand on existing literature and address the challenges outlined in the introduction by presenting a reusable machine learning platform developed at Google. Our design adopts the following principles:

One machine learning platform for many learning tasks. We chose to use TensorFlow [4] as the trainer but the platform design is not limited to this specific library. One factor in choosing (or dismissing) a machine learning platform is its coverage of existing algorithms [12]. TensorFlow provides full flexibility for implementing any type of model architecture. To name just a few, we have seen implementations of linear, deep, linear and deep combined, tree-based, sequential, multi-tower, multi-head, etc. architectures. This allows users of TFX to switch out the learning algorithm without migrating the entire pipeline to a different stack. However, it also imposes requirements on all other components. Data analysis, validation, and visualization tools need to support sparse, dense, or sequence data. Model validation, evaluation, and serving tools need to support all kinds of inference types, including (among others) regression, classification, and sequences.

Continuous training. Most machine learning pipelines are set up as workflows or dependency graphs (e.g. [14, 20]) that execute specific operations or jobs in a defined sequence. If a team needs to train over new data, the same workflow or graph is executed again. However, many real-world use-cases require continuous training. TFX supports several continuation strategies that result from the interaction between data visitation and warm-starting options. Data visitation can be configured to be static or dynamic (over a rolling range of directories). Warm-starting initializes a subset of model parameters from a previous state.

Easy-to-use configuration and tools. Providing a unified configuration framework is only possible if components also share utilities that allow them to communicate and share assets. A TFX user is only exposed to one common configuration that is passed to all components and shared where necessary. Utilities that are used by all components enable enforcement of global garbage collection policies, unified debugging and status signals, etc.

Production-level reliability and scalability. Only a small fraction of a machine learning platform is the actual code implementing the training algorithm [19]. If the platform handles and encapsulates the complexity of machine learning deployment, engineers and scientists have more time to focus on the modeling tasks. Since it is difficult to predict whether a learning algorithm will behave reasonably on new data [8], model validation is critical. In turn, model validation must be coupled with data validation in order to detect corrupted training data and thus prevent bad (yet, validated) models from reaching production. To give an example, training data that accidentally includes the label will lead to a good quality model that passes validation, but would not perform well in production where the label is not available. Validating the serving infrastructure before pushing to the production environment is vital to the reliability and robustness of any machine learning platform. Our platform provides implementations of these components that encode best practices observed in many production pipelines. Moreover, our experience shows that the distributed data processing model offered by Apache Beam [1] (and similar internal infrastructure) is a good fit for handling the large volume of data during training, model evaluation, and batch inference.

Figure 1 shows a high-level component overview of a machine learning platform and highlights the components discussed in the following sections: data analysis (Sections 3.1), data transformation (Section 3.2), data validation (Section 3.3), trainer (Section 4), model evaluation and validation

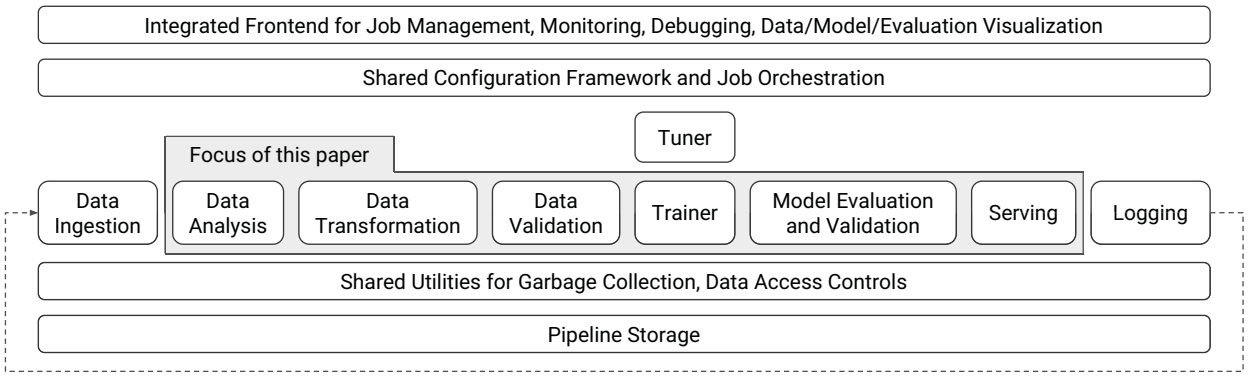

Figure 1: High-level component overview of a machine learning platform.

(Section 5), and serving (Section 6). In isolation, these components implement high-level functionality that is typical in machine-learning platforms, e.g., data sampling, feature generation, training, and evaluation [12, 14, 16]. However, it is worth pointing out two differentiations. First, we built these components to adhere to the aforementioned principles, which introduced several technical difficulties. Second, the integration of these components in a single platform, with shared configuration and utilities, enabled key improvements over existing alternatives. To give an example, transformations applied in the trainer and at serving time may need statistics generated by the data analysis component. Integrating these components ensures consistency across the pipeline and guarantees that the same transformations are applied at training and serving, which in turn prevents one form of training-serving skew (a common production headache in machine learning). Although almost all of the components can be considered optional, TFX users find it beneficial to adopt the full stack in order to achieve more robust and reliable production systems and take advantage of all management and visualization tools in the integrated frontend.

Throughout the paper, we refer to the engineers and ML practitioners using our platform as "users". In the case study of Google Play (Section 7), we refer to people who visit the Google Play store as "Play users".

3 DATA ANALYSIS, TRANSFORMATION, AND VALIDATION

Machine learning models are only as good as their training data, so understanding the data and finding any anomalies early is critical for preventing data errors downstream, which are more subtle and harder to debug. Often the data is generated by adhoc pipelines involving multiple products, systems, and usage logs. Faults (e.g., code bugs, system failures, or human errors, to name a few) can occur at multiple points of this generation process, which makes anomalies in the data not an exception, but more the norm. As a machine learning platform scales to larger data and runs continuously, there is a strong need for a reusable component that enables rigorous checks for data quality and promotes best practices for data management in the context of machine learning platforms [11]. Small bugs in the data can significantly degrade model quality over a period of time in a way that is hard to detect and diagnose (unlike catastrophic bugs that cause spectacular failures and are thus easy to track down), so constant data vigilance should be a part of any long running development of a machine learning platform.

Building such a component is challenging for several reasons. First, the component needs to support a wide range of data-analysis and validation cases that correspond to machine learning applications. The component must also be easy to deploy for a basic set of useful checks without requiring excessive customization, with additional checks being possible at the cost of some setup by the user. Moreover, the component should help the users monitor and react to data quality problems in a way that is non-intrusive (users do not receive "spammy" data-anomaly alerts) and actionable (users understand how to debug a particular data anomaly). Our experience has shown that users tend to switch off data-quality checks if they receive a large number of false-negative alerts or if the alerts are hard to understand.

The following subsections describe the implementation of this component in TFX and how it addresses the aforementioned challenges. The component treats data analysis, transformation, and validation as separate yet closely related processes, with complementary roles.

3.1 Data Analysis

For data analysis, the component processes each dataset fed to the system and generates a set of descriptive statistics on the included features. These statistics cover the presence of each feature in the data, e.g., the distribution of the number of values per example or the number of examples with and without the feature. The component also gathers statistics over feature values: for continuous features, the statistics include quantiles, equi-width histograms, the mean and standard deviation, to name a few, whereas for discrete features they include the top-K values by frequency. The component also supports statistics on configurable slices of the data (e.g., on negative and positive examples in a binary classification problem) and cross-feature statistics (e.g.,

correlation and covariance between features). By looking at these feature statistics, users can gain insights into the shape of each dataset. Note that it is possible to extend the component with further statistics, but we found that this subset provides good coverage for the needs of our users.

In a continuous training and serving environment, the above statistics must be computed efficiently at scale. Unavailable feature statistics may result in missed opportunities to correct data anomalies, while outdated feature-to-integer mappings and feature value distributions may result in a drop in model quality. On large training data, some of these statistics become difficult to compute exactly, and the component resorts to distributed streaming algorithms that give approximate results [9, 17].

3.2 Data Transformation

Our component implements a suite of data transformations to allow feature wrangling for model training and serving. For instance, this suite includes the generation of *feature-to-integer mappings*, also known as *vocabularies*. In most machine learning platforms that deal with sparse categorical features, both training and serving require mappings from string values of a sparse feature to integer IDs. The integer IDs allow operations like looking up model weights or embeddings given a specific value. Representing features in ID space often saves memory and computation time as well. Since there can be a large number (∼1–100B) of unique values per sparse feature, it is a common practice to assign unique IDs only to the most "relevant" values. The less relevant values are either dropped (i.e., no IDs assigned) or are assigned IDs from a fixed set of IDs. There are different ways to define relevance, including the common approach of using the frequency of appearance in the data.

A crucial issue is ensuring consistency of the transformation logic during training and serving. Any discrepancies imply that the model receives prediction requests with differently transformed input features, which typically hurts model quality. TFX exports any data transformations as part of the trained model, which in turn avoids problems with inconsistency.

3.3 Data Validation

After completing the analysis of the data, the component deals with the task of validation: is the data healthy or are there anomalies that need to be flagged to the user?

To perform validation, the component relies on a *schema* that provides a versioned, succinct description of the expected properties of the data. The following are examples of the properties that can be encoded in the schema:

- Features present in the data.
- The expected type of each feature.
- The expected presence of each feature, in terms of a minimum count and fraction of examples that must contain the feature.
- The expected valency of the feature in each example, i.e., minimum and maximum number of values.
- The expected domain of a feature, i.e., the small universe of values for a string feature, or range for an integer feature.

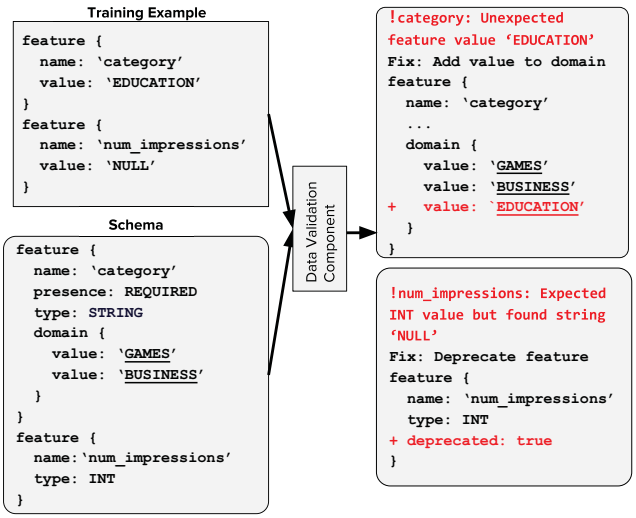

Figure 2: Sample validation of an example against a simple schema for an app store application. The schema indicates that the expected type for the 'category' feature is STRING and that for the 'num_impressions' feature is INT. Furthermore, the category feature must be present in all examples and assume values from the specified domain. On validating the example against this schema, the module detects two anomalies with simple explanations as well as suggested schema modifications. The first suggestion reflects a schema change to account for an evolution of the data (the appearance of a new value). In contrast, the second suggestion reflects the fact that there is an underlying data problem that needs to be fixed, so the feature should be marked as problematic while the problem is being investigated.

Using the schema, the component can validate the properties of specific (training and serving) datasets, flag any deviations from the schema as potential anomalies, and in most cases, provide actionable suggestions to fix the anomaly. These actions may include recommending the user to block training on particular features by marking them as "deprecated", or for expected deviations in the data, updating the schema itself to match the data. We assume that teams are responsible for maintaining the schema and updating it to newer versions as needed. We also provide tooling to help generate the first version automatically by analyzing a sample of the data as well as suggest concrete fixes to the schema as data evolves. An example of a simple schema for an app store application, the anomalies detected using this schema, and the actionable suggestions are shown in Figure 2.

While the above list of properties captures a large class of data errors that occur in practice, the schema can also encode more elaborate properties, e.g., constrain the distribution of

the values in a specific feature, or describe a specific interpretation of a feature's values (say, a string feature may have the values "TRUE" and "FALSE", which are interpreted as booleans according to the schema). However, such complex properties are often hard to specify since they involve thresholds that are hard to tune, especially since some churn in the characteristics of the data is expected in any real-life deployment.

Based on our engagements with users, both in deciding the properties that the schema should permit and in designing the end-to-end data validation component, we relied on the following key design principles:

- The user should understand at a glance which anomalies are detected and their coverage over the data.
- Each anomaly should have a simple description that helps the user understand how to debug and fix the data. One such example is an anomaly that says a feature's value is out of a certain range. As an antithetical example, it is much harder to understand and debug an anomaly that says the KL divergence between the expected and actual distributions has exceeded some threshold.
- In some cases the anomalies correspond to a natural evolution of the data, and the appropriate action is to change the schema (rather than fix the data). To accommodate this option, our component generates for each anomaly a corresponding schema change that can bring the schema up-to-date (essentially, make the anomaly part of the normal state of the data).
- We want the user to treat data errors with the same rigor and care that they deal with bugs in code. To promote this practice, we allow anomalies to be filed just like any software bug where they are documented, tracked, and eventually resolved.

These principles have affected both the logic to detect anomalies and the presentation of anomalies in the UI component of TFX.

Beyond detecting anomalies in the data, users can also look at the schema (and its versions) in order to understand the evolution of the data fed to the machine learning platform. The schema also serves as a stable description of the data that can drive other platform components, e.g., automatic feature-engineering or data-analysis tools.

4 MODEL TRAINING

One of the core design philosophies of TFX is to streamline (and automate as much as possible) the process of training production quality models which can support all training use cases. We chose to design the model trainer such that it supports training any model configured using TensorFlow [4], including implementations necessary for continuous training. It takes minimal, one-time effort to integrate modeling code written in TensorFlow with the trainer. Once done, users can seamlessly switch from one learning algorithm to another without any efforts to re-integrate with the trainer.

While continuously training and exporting machine learning models is a common production use case, often such models need to train on huge datasets to generate good quality models. This practice is becoming infeasible for many teams because it is both time and resource intensive to retrain these models. At Google, we have, on several occasions, leveraged warm-starting to attain high quality models without spending too many resources.

4.1 Warm-Starting

For many production use cases, freshness of machine learning models is critical (e.g., Google Play store, where thousands of new apps are added to the store every day). A lot of such use cases also have huge training datasets (O(100B) data points) which may take hours (or days in some cases) of training to attain models meeting desired quality thresholds. This results in a trade-off between model quality and model freshness. Warm-starting is a practical technique to offset this trade-off and, when used correctly, can result in models of same quality as one would obtain after training for many hours in much less time and fewer resources.

Warm-starting is inspired by *transfer learning* [18]. One of the approaches to transfer learning is to first train a base network on some base dataset, then use the 'general' parameters from the base network to initialize the target network, and finally train the target network on the target dataset [23]. The effectiveness of this technique depends on the generality of the features whose corresponding parameters are transferred. The same approach can be applied in the context of continuous training. In this approach, we identify a few general features of the network being trained (e.g., embeddings of sparse features). When training a new version of the network, we initialize (or warm-start) the parameters corresponding to these features from the previously trained version of the network and fine tune them with the rest of the network. Since we are transferring parameters between different versions of the same network, this technique results in much quicker convergence of the new version, thus resulting in the same quality models using fewer resources.

In the past, many teams wrote and maintained custom binaries to warm-start new models. This incurred a lot of duplicated effort that went into writing and maintaining similar code. While building TFX, the ability to selectively warm-start selected features of the network was identified as a crucial component and its implementation in TensorFlow was subsequently open sourced.

Together with features like warm-starting, TensorFlow provides a high-level unified API to configure model training using various learning techniques (e.g., deep learning, wide and deep learning, sequence learning).

4.2 High-Level Model Specification API

We decided to use an established high-level TensorFlow model specification API [22]. Our experience points to large productivity gains via a higher-level abstraction layer that hides implementation details and encodes best practices.

One of the useful abstractions we leveraged is `FeatureColumns`. `FeatureColumns` help users focus on *which features* to use

in their machine learning model. They are a declarative way of defining the input layer of a model. Another component of abstraction layer we relied on is the concept of an `Estimator`. For a given model, `Estimator` handles training and evaluation. It can be used on a local machine as well as for distributed training. Following is a simplified example of how training a feed-forward dense neural network looks like in this framework:

```
# Declare a numeric feature:
num_rooms = numeric_column('number-of-rooms')
# Declare a categorical feature:
country = categorical_column_with_vocabulary_list(
    'country', ['US', 'CA'])
# Declare a categorical feature and use hashing:
zip_code = categorical_column_with_hash_bucket(
    'zip_code', hash_bucket_size=1K)
# Define the model and declare the inputs
estimator = DNNRegressor(
    hidden_units=[256, 128, 64],
    feature_columns=[
        num_rooms, country,
        embedding_column(zip_code, 8)],
    activation_fn=relu,
    dropout=0.1)
# Prepare the training data
def my_training_data():
  # Read, parse training data and convert it into
  # tensors. Returns a mini-batch of data every
  # time returned tensors are fetched.
  return features, labels
# Prepare the validation data
def my_eval_data():
  # Read, parse validation data and convert it into
  # tensors. Returns a mini-batch of data every
  # time returned tensors are fetched.
  return features, labels
estimator.train(input_fn=my_training_data)
estimator.evaluate(input_fn=my_eval_data)
```

From our experience, users find it easier to first train a simple model in the available setting (e.g., single machine or distributed system) before experimenting with various optimization settings [24, Rule #4]. Once a baseline is established, users can experiment with these settings. A tuner integrated with the trainer can also automatically optimize the hyperparameters based on users' objectives and data.

5 MODEL EVALUATION AND VALIDATION

Machine-learned models are often parts of complex systems comprising a large number of data sources and interacting components, which are commonly entangled together [19]. This creates large surfaces on which bugs can grow and unexpected interactions can develop, potentially to the detriment of end-user experiences via the degradation of the machine-learned model. Some examples of such bugs include: different components expecting different serialized model formats, or bugs in training or serving code causing binary crashes.

These issues can be difficult for humans to detect, especially in a continuous training setting where new models are refreshed and pushed to production frequently. Having a reusable component that automatically evaluates and validates models to ensure that they are "good" before serving them to users can help prevent unexpected degradations in the user experience.

5.1 Defining a "good" model

The model evaluation and validation component of TFX is designed for this purpose. The key question that the component helps answer is: is this specific model a "good" model? We suggest two pieces: that a model is *safe to serve*, and that it has the desired *prediction quality*.

By *safe to serve*, we mean obvious requirements such as: the model should not crash or cause errors in the serving system when being loaded, or when sent bad or unexpected inputs, and the model shouldn't use too many resources (such as CPU or RAM). One specific problem we have encountered is when the model is trained using a newer version of a machine learning library than is used at serving time, resulting in a model representation that cannot be used by the serving system. Product teams care about user satisfaction and product health, which are better captured by measures of *prediction quality* (such as app install rate) on live traffic than by the objective function on the training data.

5.2 Evaluation: human-facing metrics of model quality

Evaluation is used as part of the interactive process where teams try to iteratively improve their models. Since it is costly and time-consuming to run A/B experiments on live traffic, models are evaluated offline on held-out data to determine if they are promising enough to start an online A/B experiment. The evaluation component provides proxy metrics such as AUC or cost-weighted error that approximate business metrics more closely than training loss, but are computable offline. Once teams are satisfied with their models' offline performance, they can conduct product-specific A/B experiments to determine how their models actually perform on live traffic on relevant business metrics.

5.3 Validation: machine-facing judgment of model goodness

Once a model is launched to production and is continuously being updated, automated validation is used to ensure that the updated models are good. We validate that a model is safe to serve with a simple canary process. We evaluate prediction quality by comparing the model quality against a fixed threshold as well as against a baseline model (e.g., the current production model). Any new model failing any of these checks is not pushed to serving, and product teams are alerted.

One challenge with validating safety is that our canary process will not catch all potential errors. Another challenge with validation in a continuously training pipeline is that it is hard to distinguish expected and unexpected variations in a model's behaviour. When the training data is continuously changing, some variation in the model's behaviour and its performance on business metrics is to be expected.

Hence, there is a tension between being too conservative and alerting users to small changes in these metrics, which results in users tiring of and eventually ignoring the alerts; and being too loose and failing to catch unexpected changes. From our experience working with several product teams, most bugs cause dramatic changes to model quality metrics that can be caught by using loose thresholds. However, there is a strong selection bias here, since more subtle issues may not have drawn our attention.

5.4 Slicing

One of the features the evaluation component offers is the ability to compute metrics on slices of data. We define a *slice* as a subset of the data containing certain features. For instance, a product team might be concerned about the performance of their model in the US, so they might wish to compute metrics on the subset of data that contains the feature "Country = US".

This is useful in both evaluating and validating models in the case where product teams have specific slices which they are concerned about, especially small slices, since metrics on the entire dataset can fail to reflect the performance on these small slices [15]. Slicing can help product teams understand and improve performance on these slices, and also avoid serving models that sacrifice quality on these slices for better overall performance.

5.5 User Attitudes towards Validation

In the process of deploying the model and validation component, we made an interesting discovery regarding user attitudes towards validation in machine learning platforms. Our general sense is that the value of validation is not immediately apparent to users; however, the costs in terms of additional configuration and greater resource consumption immediately stand out to them. As an illustration, no product teams actively requested the validation function when the component was first built, and when the feature was explained to them, few activated it. The fact that the validation feature did not directly improve their machine-learned models' performance, and on the contrary, could result in them serving old models if the checks did not pass, also added to their hesitation.

However, encountering a real issue in production which could have been prevented by validation made the value of the validation apparent to the teams, who were then eager to activate validation. As a result of this observation, we plan to provide a configuration-free validation setup that is enabled by default for all users.

6 MODEL SERVING

Reaping the benefits of sophisticated machine-learned models is only possible when they can be served effectively. TensorFlow Serving [2] provides a complete serving solution for machine-learned models to be deployed in production environments. TensorFlow Serving's framework is also designed to be flexible, supporting new algorithms and experiments.

Scaling to varied traffic patterns is an important goal of TensorFlow Serving. By providing a complete yet customizable framework for machine learning serving, TensorFlow Serving aims to reduce the boilerplate code needed to deploy a production-grade serving system for machine-learned models.

Serving systems for production environments require low latency, high efficiency, horizontal scalability, reliability and robustness. This section elaborates on two specific system challenges: low latency and high efficiency.

6.1 Multitenancy with Isolation

Multitenancy in the context of TensorFlow Serving means enabling a single instance of the server to serve multiple machine-learned models concurrently. Serving multiple models at production scale can lead to cross-model interference, which is a challenging problem to solve. TensorFlow Serving provides soft model-isolation, so that the performance characteristics of one model has minimal impact on other models. While deploying servers that handle a high number of queries per second, we encountered interference between the request processing and model-load processing flows of the system. Specifically, we observed latency peaks during the interval when the system was loading a new model or a new version of an existing model.

To enhance isolation between these operations, we implemented a feature that allows the configuration of a separate dedicated threadpool for model-loading operations. This is built upon a feature in TensorFlow that allows any operation to be executed with a caller-specified threadpool. As a result, we were able to ensure that threads performing request processing would not contend with the long operations involved with loading a model from disk.

Empirically, we found that setting the threadpool size for model-loading operations to 1 or 2 was ideal for system performance. This configuration supports faster request processing consistently, trading off slower model-loads. Prior to defining a separate threadpool for load operations, for a specific model, we observed that the 99.9-percentile inference request latency measured during loads was in the range of ∼500 to ∼1500 msec. However, with the specification of a separate threadpool, the 99.9-percentile inference request latency during loads reduced to a range of ∼75 to ∼150 msec.

6.2 Fast Training Data Deserialization

Unlike previous machine learning libraries at Google, each using custom input formats and parsing code, TensorFlow uses a common data format. This approach enables the community to share their data, models, tools, visualizations, optimizations, and other techniques.

On the other hand, the common format was a suboptimal solution for some sources of data. Choosing a common format involves tradeoffs such as size of the data, cost of parsing, and the need to write format-conversion code. We decided on the tensorflow.Example [3] protocol buffer format (cross-language, serializable data-structure).

Non neural network (e.g., linear) models are often more data intensive than CPU intensive. For such models, data input, output, and preprocessing tend to be the bottleneck. Using a generic protocol buffer parser proved to be inefficient.

To resolve this, a specialized protocol buffer parser was built based on profiles of various real data distributions in multiple parsing configurations. Lazy parsing was employed, including skipping complete parts of the input protocol buffer that the configuration specified as unnecessary. In addition, to ensure the protocol buffer parsing optimizations were consistently useful, a benchmarking suite was built. This suite was useful in ensuring that optimizing for one type of data distribution or configuration did not negatively impact performance for other types of data distributions or configurations. While implementing this system, extreme care was taken to minimize data copying. This was especially challenging for sparse data configurations.

The application of the specialized protocol buffer parser resulted in a speedup of 2-5 times on benchmarked datasets.

7 CASE STUDY: GOOGLE PLAY

One of the first deployments of TFX is the recommender system for Google Play, a commercial mobile app store. The goal of the Google Play recommender system is to recommend relevant Android apps to the Play app users when they visit the homepage of the store, with an aim of driving discovery of apps that will be useful to the user. The input to the system is a "query" that includes the information about the app user and context. The recommender system returns a list of apps, which the user can either click on or install. Since the corpus contains over a million apps, it is intractable to score every app for every query. Hence, the first step in this system is retrieval, which returns a short list of apps based on various signals. Once we have this short list of apps, the ranking system uses the machine-learned model to compute a score per app and presents a ranked list to the user. In this case study, we focus on the ranking system.

The machine learning model that ranks the items is trained continuously as fresh training data arrives (usually in batches). The typical training dataset size is hundreds of billions of examples where each example has query features (e.g., the user's context) as well as impression features (e.g., ratings and developer of app being ranked). After rigorous validation (e.g., comparing quality metrics with models serving live traffic), the trained models are deployed through TensorFlow Serving in data centers around the globe and collectively serve thousands of queries per second with a strict latency requirement of tens of milliseconds. Due to fresh models being trained daily, the servers have to reload multiple models (both the production models, as well as other experimental models) per day. This is done without any degradation in latency.

As we moved the Google Play ranking system from its previous version to TFX, we saw an increased velocity of iterating on new experiments, reduced technical debt, and improved model quality. To list a few, the overall product has benefitted in the following ways:

- The data validation and analysis component helped in discovering a harmful training-serving feature skew. By comparing the statistics of serving logs and training data on the same day, Google Play discovered a few features that were always missing from the logs, but always present in training. The results of an online A/B experiment showed that removing this skew improved the app install rate on the main landing page of the app store by 2%.
- Warm-starting helped improve model quality and freshness while reducing the time and resources spent on training over hundreds of billions of examples. Training from scratch can take several days to converge to a high-quality model, making it hard for the model to recommend trending or recently added apps that were missing from the training data. One option to produce new models more frequently is to reduce the number of training iterations, at the expense of lower quality models. This is the same trade-off between model quality and model freshness described in Section 4. Hence, Google Play, for whom it is infeasible to train every new model from scratch, adopted the technique of warm-starting to selectively initialize a subset of model parameters (e.g., embeddings) from a previously trained model. This enabled Google Play to push a high-quality fresh model for serving frequently.
- Model validation helped in understanding and troubleshooting performance differences between the old and new models. The model validation component tests the new model against the production model, preventing issues like accidentally pushing partially-trained models to serving because of system failures.
- The model serving component enabled deploying the trained model to production, while guaranteeing high performance and flexibility. Specifically, Google Play benefitted from optimizations in the serving system, including support for isolation in a multi-tenant environment and fast custom proto parsing, described in Section 6.

8 CONCLUSIONS

We discussed the anatomy of general-purpose machine learning platforms and introduced TFX, an implementation of such a platform with TensorFlow-based learners and support for continuous training and serving with production-level reliability. The key approach is to orchestrate reusable components (data analysis and transformation, data validation, model training, model evaluation and validation, and serving infrastructure) effectively and provide a simple unified configuration for users. TFX has been successfully deployed in the Google Play app store, reducing the time to production and increasing its app install rate by 2%.

Many interesting challenges remain. While TFX is general-purpose and already supports a variety of model and data types, it must be flexible and accommodate new innovations from the machine learning community. For example, sequence-to-sequence models have recently been used to produce state-of-the-art results in machine translation. Supporting these models required us to carefully think about capturing sequential information in the common data format and posed new challenges for the serving infrastructure and model validation, among others. In general, each addition may affect multiple components of TFX, so all components must be extensible. Furthermore, as machine learning becomes more prevalent, there is a strong need for understandability where a model can explain its decision and actions to users. We believe the lessons we learned from deploying TFX provide a basis for building an interactive platform that provides deeper insights to users.

REFERENCES

[1] *Apache Beam: An Advanced Unified Programming Model*. https://beam.apache.org/, accessed 2017-06-05.

[2] *Running your models in production with TensorFlow Serving*. https://research.googleblog.com/2016/02/running-your-models-in-production-with.html, accessed 2017-02-08.

[3] *TensorFlow - Reading data*. https://www.tensorflow.org/how_tos/reading_data/, accessed 2017-02-08.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.

[5] Rami Abousleiman, Guangzhi Qu, and Osamah A. Rawashdeh. 2013. North Atlantic Right Whale Contact Call Detection. *CoRR* abs/1304.7851 (2013).

[6] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *DLRS*. 7–10.

[7] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *RecSys*. 191–198.

[8] Yann Dauphin, Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. 2014. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR* abs/1406.2572 (2014).

[9] Philippe Flajolet, ric Fusy, Olivier Gandouet, and et al. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*.

[10] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. 2013. MLbase: A Distributed Machine-learning System. In *CIDR*.

[11] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *PVLDB* 9, 12 (2016), 948–959.

[12] Sara Landset, Taghi M. Khoshgoftaar, Aaron N. Richter, and Tawfiq Hasanin. 2015. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. *Journal of Big Data* 2, 1 (2015), 24.

[13] Cheng Li, Yue Lu, Qiaozhu Mei, Dong Wang, and Sandeep Pandey. 2015. Click-through Prediction for Advertising in Twitter Timeline. In *KDD*. 1959–1968.

[14] Jimmy J. Lin and Alek Kolcz. 2012. Large-scale machine learning at twitter. In *SIGMOD*. 793–804.

[15] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. 2013. Ad Click Prediction: A View from the Trenches. In *KDD*. 1222–1230.

[16] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2015. MLlib: Machine Learning in Apache Spark. *CoRR* abs/1505.06807 (2015).

[17] J.I. Munro and M.S. Paterson. 1980. Selection and sorting with limited storage. *Theoretical Computer Science* 12, 3 (1980), 315–323.

[18] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Trans. on Knowl. and Data Eng.* 22, 10 (Oct. 2010), 1345–1359.

[19] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *NIPS*. 2503–2511.

[20] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2016. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. *CoRR* abs/1610.09451 (2016).

[21] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: a system for machine learning model management. In *HILDA@SIGMOD*. 14.

[22] Cassandra Xia, Clemens Mewald, D. Sculley, David Soergel, George Roumpos, Heng-Tze Cheng, Illia Polosukhin, Jamie Alexander Smith, Jianwei Xie, Lichan Hong, Martin Wicke, Mustafa Ispir, Philip Daniel Tucker, Yuan Tang, and Zakaria Haque. 2017. Train and Distribute: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks. KDD (under review).

[23] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *NIPS*. 3320–3328.

[24] Martin Zinkevich. 2016. Rules of Machine Learning. In *NIPS Workshop on Reliable Machine Learning*. Invited Talk.