Summary of “Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads”

Wenyi Wu (wenyiwu), Alexandra Spence (aspe)

Problem and Motivation

Since many machine learning models require a large amount of resources to run, many enterprises set up shared clusters to run models. Deep learning workloads create new requirements and constraints for cluster management systems. They often require multiple GPUs and require that tasks are scheduled to run at the same time. Locality is also important to ensure that model parameters can be synchronized across GPUs.

While many enterprises are setting up these clusters, at the time of this paper there were no systematic studies of these clusters. Because of this, the authors decided to collect data from Philly, a service used by Microsoft for training machine learning models, and analyze this data. The goals were to study how waiting for locality constraints influences queuing delays and to study how locality-aware scheduling affects the GPU utilization for jobs. They also studied reasons for job failures, with the goal of helping identify the main causes for failures and finding ways to reduce these failures.

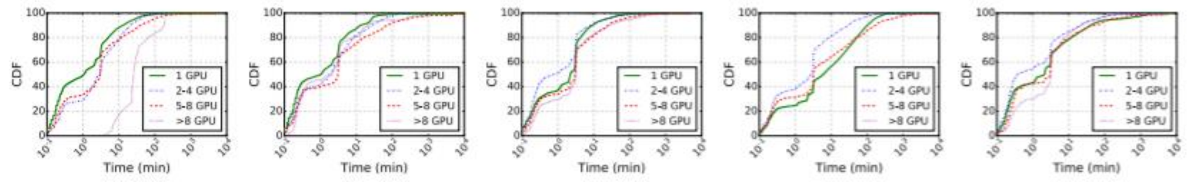
Hypothesis

It is expected that relaxing locality restraints on a system will reduce the queuing delays, especially for jobs that use many GPUs. Since most GPUs within a cluster are allocated to users, high cluster utilization is expected, but hardware utilization of GPUs is low. About 30% of jobs are either killed or fail. The exact reasons for each of these is unknown, but by examining logs from jobs submitted to the cluster, we can study how locality restraints influence queuing delays, determine what is causing low GPU utilization, and determine what is causing the failures.

Solution Overview

To analyze GPU clusters, the authors collect log files from the YARN scheduler, stdout and stderr, and the Ganglia monitoring system that reports statistics on hardware usage such as CPU, memory, network, and GPU. These logs were collected from October 2017 to December 2017 and contain jobs over 14 virtual clusters.

For locality and its effect on queueing delays, the authors first looked at queuing delay in five of the largest virtual clusters to see which jobs have the largest queueing delays. The results can be seen in the figure below.



(a) VC1 (b) VC2 (c) VC3 (d) VC4 (e) VC5

Figure 3: CDF of scheduler queueing delay for five of the largest virtual clusters in our deployment. Note that VC4 contains no jobs with >8 GPU.

Based on this figure, it appears that the queueing delays between different numbers of GPUs are not distinct. However, by looking at queueing delays for jobs with 5-8 GPUs compared to jobs with more than 8 GPUs, the authors determined that jobs that are spread across more servers have a lower queueing delay, which is due to the fact that Philly chooses to relax the locality restraint in exchange for lower queuing delay. They also found that there are two causes for queueing delay, fairness, which causes fair-share delay, and resource fragmentation, which is called fragmentation delay. They found that fragmentation delay was more common than fair-share delay in large jobs, while the two causes were split more evenly in small jobs.

In general, GPU usage tends to be low for most problems, with 47.7% of in-use GPU cycles wasted across all jobs. For jobs that require more than 8 GPUs, they will not fit in one server in this system, leading to communication overhead. By analyzing different configurations using ResNet-50, they determined that GPU utilization may be lowered by interference from other jobs and that GPU utilization is higher for jobs that are contained within one server than jobs spread across more than one server. A table of these results is shown below.

Metric	SameServer	DiffServer	IntraServer	InterServer
GPU util.	57.7	49.6	37.5	36.5
Images/s	114.8	98.0	75.6	74.1

Table 4: Mean GPU utilization and training performance of ResNet-50 over different locality/colocation configurations.

Another factor leading to inefficient GPU usage that the authors examined is the efficiency of training epochs. The authors found that while most jobs did use all of their epochs to reach their optimal loss, most jobs only needed 40% of their epochs to get within 0.1% of their lowest loss, suggesting that they can be ended early.

To examine job failures, the authors classified failures into infrastructure, AI engine, and user errors, and then examined runtime to failure, GPU demand, and the product of these to determine which errors were the most frequent. They found that failures caused by user error are dominant, and that often the same user will cause many similar errors. They also found that infrastructure failures occur infrequently, but have high runtime to failure.

Limitations and Possible Improvements

-Since many deep learning jobs run for many hours, it might be better to keep locality constraints instead of relaxing them for some drops. They also suggest migrating jobs to machines with better locality.

-There should be job placement policies that help mitigate inter-job interference. For example, they propose placing small jobs on dedicated servers to reduce sharing.

-Many job failures are caused by users. To help reduce these errors, they propose setting up a pool of cheaper virtual machines to pre-run jobs and adding a well-defined schema for datasets. They also suggest setting up a system to proactively observe failures to help prevent retrying for models that have failures that cannot be solved by rerunning the same code.

Summary of Class Discussion

Q: How much failure handling should be done by the user and how much should be automated?

A: The paper somewhat addressed this by discussing setting up a virtual machine to pre-run jobs. In general, it is expected that users will have errors in their code, so the scheduler should always try to detect errors. The error handling itself can be done by users.

Q: The paper proposed the strategy of migrating a job to machines with better locality in order to increase resource usage. What effect would migrating a job have on the run time and resource usage of the job?

A: For these types of jobs, all that would need to be transferred would generally be the loss. The training data is stored using HDFS, so it could be pulled from the nearest location instead of being migrated.

Q: Section 4.1 showed that many jobs only need around 40% of their iterations to get within 0.1% of their lowest loss. Should jobs automatically terminate when their loss appears to be at a minimum, and what are some ways to identify when training is complete?

A: There are other papers that have looked into determining when to stop the training process. For Resnet-50, the loss will hit a point where it stays stagnant for a while before the loss improves again, so it might seem like training has finished when it has not, which can make this difficult to determine. By having a user specification for desired loss, it would be possible to stop training once the loss is close to or at the user threshold.

Q: Why do jobs on more GPUs have more failures?

A: If failures are equally likely to occur on any GPU, having more GPUs will increase the failure rate.

Summary of "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform"

Wenyi Wu (wenyiwu), Alexandra Spence (aspe)

Problem and Motivation

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.

Hypothesis

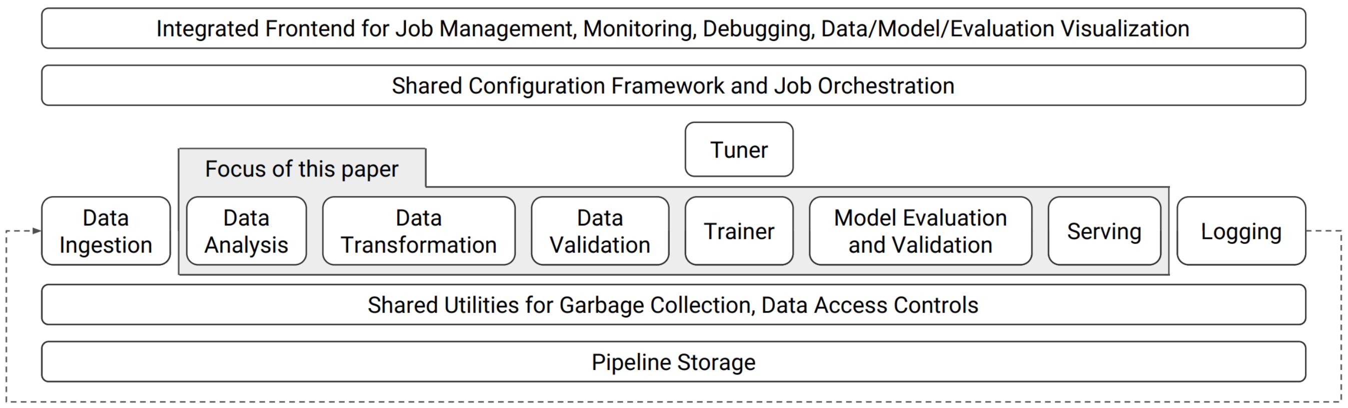
The conceptual workflow of applying machine learning to a specific use case is simple: at the training phase, a learner takes a dataset as input and emits a learned model; at the inference phase, the model takes features as input and emits predictions. However, the actual workflow becomes more complex 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.
- 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.
- 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.

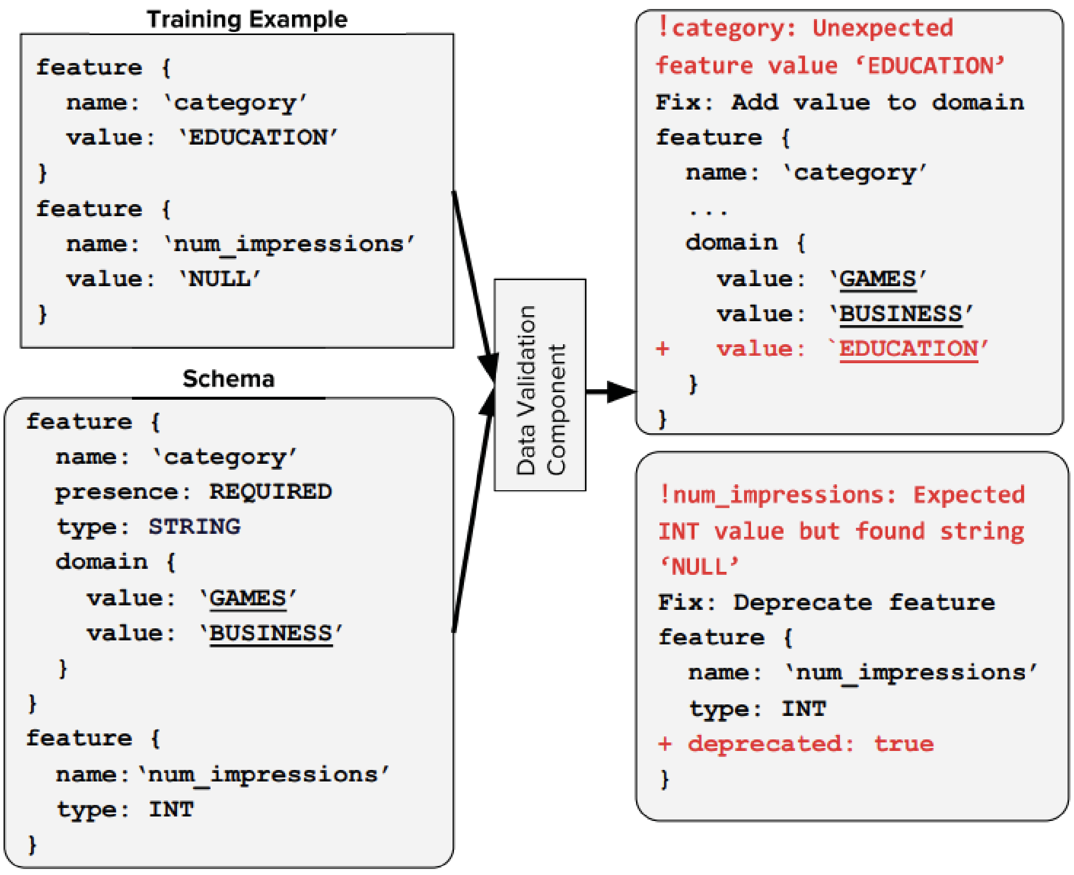
Solution Overview

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.
- Continuous training.
- Easy-to-use configuration and tools.
- Production-level reliability and scalability.


Integrated Frontend for Job Management, Monitoring, Debugging, Data/Model/Evaluation Visualization
Shared Configuration Framework and Job Orchestration
Tuner
Focus of this paper
Data Ingestion
Data Analysis
Data Transformation
Data Validation
Trainer
Model Evaluation and Validation
Serving
Logging
Shared Utilities for Garbage Collection, Data Access Controls
Pipeline Storage

- 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.
- 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 featureto-integer mappings, also known as vocabularies.



```
feature {
  name: 'category'
  value: 'EDUCATION'
}
feature {
  name: 'num_impressions'
  value: 'NULL'
}
```

```
feature {
  name: 'category'
  presence: REQUIRED
  type: STRING
  domain {
    value: 'GAMES'
    value: 'BUSINESS'
  }
}
feature {
  name:'num_impressions'
  type: INT
}
```

```
!category: Unexpected
feature value 'EDUCATION'
Fix: Add value to domain
feature {
  name: 'category'
  ...
  domain {
    value: 'GAMES'
    value: 'BUSINESS'
+   value: 'EDUCATION'
  }
}
```

```
!num_impressions: Expected
INT value but found string
'NULL'
Fix: Deprecate feature
feature {
  name: 'num_impressions'
  type: INT
+ deprecated: true
}
```

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.

- Data Validation
  - To perform validation, the component relies on a schema that provides a versioned, succinct description of the expected properties of the data.
- Model Training
  - Warm Starting
    - 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.

- High-Level Model Specification API

```
# 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)
```

- Model Validation
- Model Serving
  - 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.
  - A specialized protocol buffer parser was built based on profiles of various real data distributions in multiple parsing configurations.

Limitations and Possible Improvements

- While TFX is generalpurpose and already supports a variety of model and data types, it must be flexible and accommodate new innovations from the machine learning community.
- As machine learning becomes more prevalent, there is a strong need for understandability where a model can explain its decision and actions to users.

Summary of Class Discussion

Q: The title says “TFX” and this is Tensorflow based. Is this applicable to other machine learning frameworks?

A: I believe it is. There should be no obvious technical difficulties to apply this idea to other frameworks such as PyTorch.

Q: In warm-starting it says “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. ” How much effort is needed to select proper features?

A: The paper says that it can use embeddings of sparse features but I'm not convinced that those features are good choices for warm-starting. Users should be responsible for selecting the proper features because they are the most familiar with the dataset and the problem. So I think some level of human expertise is still required.

Q: When is this warm-starting used?

A: This is used only in conitnuous training. To be more specific, the warm-starting uses the previously trained version of the model to initialize the new version.

Q: Who's writing the schema for data validation?

A: Users are responsible for writing and updating the schema.