TensorFlow Estimators: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks

Heng-Tze Cheng[†]   Zakaria Haque[†]   Lichan Hong[†]   Mustafa Ispir[†]   Clemens Mewald[†*]
Illia Polosukhin[†]   Georgios Roumpos[†]   D Sculley[†]   Jamie Smith[†]   David Soergel[†]   Yuan
Tang[‡]   Philipp Tucker[†]   Martin Wicke[†*]   Cassandra Xia[†]   Jianwei Xie[†]

[†]Google, Inc.   [‡]Uptake Technologies, Inc.

ABSTRACT

We present a framework for specifying, training, evaluating, and deploying machine learning models. Our focus is on simplifying cutting edge machine learning for practitioners in order to bring such technologies into production. Recognizing the fast evolution of the field of deep learning, we make no attempt to capture the design space of all possible model architectures in a domain- specific language (DSL) or similar configuration language. We allow users to write code to define their models, but provide abstractions that guide developers to write models in ways conducive to productionization. We also provide a unifying `Estimator` interface, making it possible to write downstream infrastructure (e.g. distributed training, hyperparameter tuning) independent of the model implementation.

We balance the competing demands for flexibility and simplicity by offering APIs at different levels of abstraction, making common model architectures available out of the box, while providing a library of utilities designed to speed up experimentation with model architectures. To make out of the box models flexible and usable across a wide range of problems, these canned `Estimators` are parameterized not only over traditional hyperparameters, but also using *feature columns*, a declarative specification describing how to interpret input data.

We discuss our experience in using this framework in research and production environments, and show the impact on code health, maintainability, and development speed.

1 INTRODUCTION

Machine learning, and in particular, deep learning, is a field of growing importance. With the deployment of large GPU clusters in datacenters and cloud computing services, it is now possible to apply these methods not only in theory, but integrate them successfully into production systems.

Engineers working on production systems have only recently gained the ability to apply advanced machine learning, driven in large part by the availability of machine learning frameworks that

---

[*]Corresponding authors: {clemensm,wicke}@google.com

implement the lower level numerical computations in efficient ways and allow engineers to focus on application-specific logic (see e.g., [2–5, 7, 8, 11, 14, 17–20]). However, the huge amounts of data involved in training, especially for deep learning models, as well as the complications of running high intensity computations efficiently on heterogeneous and distributed systems, has prevented the most advanced methods from being widely adopted in production.

As the field of deep learning is still young and developing fast, any framework hoping to remain relevant must be expressive enough to not only represent today's model architectures, but also next year's. If the framework is to be used for experimentation with model architectures (most serious product work requires at least some experimentation), it is also crucial to offer the flexibility to change details of models without having to change components that are deeply embedded, and which have a highly optimized, low level implementation.

There is a natural tension between such flexibility on the one hand, and simplicity and robustness on the other hand. We use simplicity in a broad sense: From a practitioner's point of view, implementing models should not require fundamentally new skills, assuming that the model architecture is known. Experimenting with model features should be transparent, and should not require deep insights into the inner workings of the framework used to implement the model. We talk of robustness both as a quality of the software development process, as well as a quality of the resulting software. We call a framework robust if it is easy to write correct and high-quality software using it, but hard to write broken or poorly performing software. A framework which nudges the developer to use best practices, and which makes it hard to "shoot yourself in the foot" is robust.

Because of the need to keep up with and enable research, many deep learning frameworks value flexibility above all else (e.g., [2, 11, 20]). They achieve this flexibility by providing relatively low-level primitive operations (e.g., `matmul`, `add`, `tanh`), and require the user to write code in a regular programming language in order to specify their model. To simplify life for their users and speed up development, these frameworks often provide some higher level components, such as layers (e.g., a fully connected neural network layer with an optional activation function). Development in a fully-fledged programming language is inherently dangerous. Working at a low level can also lead to a lot of code duplication, with the software maintenance headaches that come with that.

On the other end of the spectrum are systems which use a DSL to describe the model architecture (e.g., [3, 5, 13, 17]). Such systems are more likely to be geared for specific production use cases. They can

make common cases very simple to implement (the most common models may even be built-in primitives). Their higher level of abstraction allows these frameworks to make optimizations that are inaccessible to their more flexible peers. They are also robust: users are strongly guided towards model architectures that work, and it is hard to write down models that are fundamentally broken. Apart from the lack of flexibility when it comes to new model types and architectures, these DSL based systems can be hard to maintain in the face of an inexorably advancing body of new research. Adding more and more primitives to a DSL, or adding more and more options to existing primitives can be fatal. Google's own experience with such a system [13] prompted the development of TensorFlow [2].

TensorFlow is an open source software library for machine learning, and especially deep learning. It represents computation as a generalized data flow graph. The graph is first built, and then executed separately from graph construction. Operations such as mul, add, etc., are represented as nodes in the graph. Edges represent the data flowing between nodes as a Tensor containing a multi-dimensional array. In the following, we use *op* and Tensor interchangeably to denote a node in the graph (op) and the output that is created when the node is executed. Most ops are stateless tensor-in-tensor-out functions. State is represented in the graph as Variables, special stateful ops. Users can assign ops and variables to any device. A device can be a CPU, GPU, TPU, and can live on the local machine or a remote TensorFlow server. TensorFlow then seamlessly handles communication between these devices. This is one of the most powerful aspects of TensorFlow, and we rely on it heavily to enable scaling models from a single machine to datacenter-scale.

The framework described in this paper is implemented on top of TensorFlow[1], and has been made available as part of the TensorFlow open-source project. Faced with competing demands, our goal is to provide users with utilities that simplify common use cases while still allowing access to the full generality of TensorFlow. Consequently, we do not attempt to capture the design space of machine learning algorithms in a DSL. Instead, we offer a harness which removes boilerplate by providing best practice implementations of common code patterns. The components we provide are reusable, and integration points for users are strategically placed to encourage reusable user code. The user configuration is performed by writing regular TensorFlow code, but a number of lower level TensorFlow concepts are safely encapsulated and users do not have to reason about them, eliminating a source of common problems.

Some of the lower level components such as layers are closely related in similar frameworks aimed at simplifying model construction [10, 15, 16, 21].

The highest level object in our framework is an Estimator, which provides an interface similar to that of Scikit-learn [19], with some adaptations to simplify productionization. Scikit-learn has been used in a large number of small to medium scale machine learning tasks. Using a widely known interface allows practitioners who are not specialists in TensorFlow to start working productively immediately.

---

[1]While we hope that our description of the features in this paper is largely self-contained, basic familiarity with TensorFlow will give valuable context to the reader.

In the remainder of the paper, we will first discuss the overall design of our framework (Sec. 2), before describing in detail all major components (Sec. 3) and our mechanisms for distributed computations (Sec. 4). We then discuss case studies and show experimental results (Sec. 5).

2 DESIGN OVERVIEW

The design of our framework is guided by the overarching principle that users should be led to best practices, without having to abandon established idioms wherever this is possible. Because our framework is built on TensorFlow, we inherit a number of common design patterns: there is a preference for functions and closures over objects, wherever such closures are sufficient; callbacks are common. Our layer design is informed by the underlying TensorFlow style: our layer functions are also tensor-in-tensor-out operations. These preferences are stylistic in nature and have no impact on the performance or expressivity of the framework, but they allow users to easily transition if they are used to working with TensorFlow.

Because one of the greatest strengths of TensorFlow is its flexibility, it is crucial for us to not restrict what users can accomplish. While we provide guides that nudge people to best practices, we provide escape hatches and extension points that allow users to use the full power of TensorFlow whenever they need to.

Our requirements include simplifying model building in general, offering a harness that encourages best practices and guides users to a production-ready implementation, as well as implementing the most common types of machine learning model architectures, and providing an interface for developers of downstream frameworks and infrastructure. We are therefore dealing with three distinct (but not necessarily disjoint) classes of users: users who want to build custom machine learning models, users who want to use common models, and users who want to build infrastructure using the concept of a model, but without knowledge of the specifics.

These user classes inform the high level structure of our framework. At the heart is the Estimator class (see Section 3.2). Its interface (modeled after the eponymous concept in Scikit-learn [19]) provides an abstraction for a machine learning model, detailed enough to allow for downstream infrastructure to be written, but general enough to not constrain the type of model represented by an Estimator. Estimators are given input by a user-defined input function. We provide implementations for common types of inputs (e.g., input from numpy [12]).

The Estimator itself is configured using the model_fn, a function which builds a TensorFlow graph and returns the information necessary to train a model, evaluate it, and predict with it. Users writing custom Estimators only have to implement this function. It is possible, and in fact, common, that model_fn contains regular TensorFlow code that does not use any other component of our framework. This is often the case because existing models are being adapted or converted to be implemented in terms of an Estimator. We do provide a number of utilities to simplify building models, which can be used independently of Estimator (see Sec. 3.1). This mutual independence of the abstraction layers is an important feature of our design, as it enables users to choose freely the level of abstraction best suited for the problem at hand.

It is worth noting that an `Estimator` can be constructed from a `Keras` Model. Users of this compatibility feature cannot use all features of Estimator (in particular, one cannot specify a separate inference graph with this method), but it is nevertheless useful for comparisons, and to use existing models inside downstream infrastructure (such as [6]).

We also provide a number of `Estimator` implementations for common machine learning algorithms, which we called Canned Estimators (these are subclasses of `Estimator`, see Section 3.3). In our implementations, we use the same mechanisms that a user who writes a custom model would use. This ensures that we are users of our own framework. To make them useful for a wide variety of problems, canned `Estimators` expose a number of configuration options, the most important of which is the ability to specify input structure using feature columns.

3 COMPONENTS

In this section we will describe in detail the various components that make up our framework and their relationships. We start with layers, lower-level utilities that can be used independently of `Estimator`, before discussing various aspects of `Estimator` itself.

3.1 Layers

One of the advantages of Deep Learning is that common model architectures are built up from composable parts. For deep neural networks, the smallest of these components are called network layers, and we have adopted this name even though the concept is more widely applicable. A layer is simply a reusable part of code, and can be as simple as a fully connected neural network layer or as complex as a full inception network. We provide a library of layers which is well tested and whose implementation follow best practices. We have given our layers a consistent interface in order to ease the cognitive burden on users. In our framework, layers are implemented as free functions, taking Tensors as input arguments (along with other parameters), and returning Tensors. TensorFlow itself contains a large number of ops that behave in the same manner, so layers are a natural extension of TensorFlow and should feel natural to users of TensorFlow. Because layers accept and produce regular Tensors, layers and regular TensorFlow ops can be mixed without requiring special care.

We implement layer functions with best practices in mind: layers are generally wrapped in a `variable_scope`. This ensures that they are properly grouped in the `TensorBoard` visualization tool, which is essential when inspecting large models. All variables that are created as part of a layer are obtained using `get_variable`, which ensures that variables can be reused or shared in different parts of the model. All layers assume that the first dimension of input tensors is the batch dimension, and accept variable batch size input. This allows changing the batch size as a hyperparameter during tuning, and it ensures that the model can be reused for inference, where inputs don't necessarily arrive in batches.

As an example, let's create a simple convolutional net to classify an image. The network comprises three convolutional and three pooling layers, as well as a final fully connected layer. We have set sensible defaults on many arguments, so the invocations are compact unless uncommon behavior is desired:

```
1  # Input images as a 4D tensor (batch, width,
2  # height, and channels)
3  net = inputs
4  # instantiate 3 convolutional layers with pooling
5  for _ in range(3):
6      net = layers.conv2d(net,
7                          filters=4,
8                          kernel_size=3,
9                          activation=relu)
10     net = layers.max_pooling2d(net,
11                          pool_size=2,
12                          strides=1)
13 logits = layers.dense(net, units=num_classes)
```

We separate out some classes of layers that share a more restricted interface. *Losses* are functions which take an input, a label, and a weight, and return a scalar loss. These functions, such as `l1_loss` or `l2_loss` are used to produce a loss for optimization.

Metrics are another special class of layers commonly used in evaluation: they take again a label, a prediction, and optionally a weight, and compute a metric such as log-likelihood, accuracy, or a simple mean squared error. While superficially similar to losses, they support aggregating a metric across many minibatches, an important feature whenever the evaluation dataset does not fit into memory. Metrics return two Tensors: update_op, which should be run for each minibatch, and a value_op which computes the final metric value. The update_op does not return a value, and only updates internal variables, aggregating the new information contained in the input minibatch. The value_op uses only the internal state to compute a metric value and returns it. The `Estimator`'s evaluation functionality relies on this usage pattern (see below). Properly implementing metrics is nontrivial, and our experience shows that metrics that are naively implemented from scratch lead to problems when using large datasets (using TensorFlow queues in evaluation requires extra finesse to avoid losing examples to logging or TensorBoard summary writing).

3.2 Estimator

At the heart of our framework is `Estimator`, a class that both provides an interface for downstream infrastructure, as well as a convenient harness for developers. The interface for users of `Estimator` is loosely modeled after Scikit-learn and consists of only four methods: train trains the model, given training data. evaluate computes evaluation metrics over test data, predict performs inference on new data given a trained model, and finally, export_savedmodel exports a SavedModel, a serialization format which allows the model to be used in TensorFlow Serving, a prebuilt production server for TensorFlow models [1].

The user configures an `Estimator` by passing a callback, the model_fn, to the constructor. When one of its methods is called, `Estimator` creates a TensorFlow graph, sets up the input pipeline specified by the user in the arguments to the method (see Sec. 3.2), and then calls the model_fn with appropriate arguments to generate the graph representing the model. The `Estimator` class itself contains the necessary code to run a training or evaluation loop, to predict using a trained model, or to export a prediction model for use in production.

Estimator hides some TensorFlow concepts, such as Graph and Session, from the user. The `Estimator` constructor also receives a configuration object called RunConfig which communicates everything that this `Estimator` needs to know about the environment

in which the model will be run: how many workers are available, how often to save intermediate checkpoints, etc.

To ensure encapsulation, `Estimator` creates a new graph, and possibly restores from checkpoint, every time a method is called. Rebuilding the graph is expensive, and it could be cached to make it more economical to run, say, `evaluate` or `predict` in a loop. However, we found it very useful to explicitly recreate the graph, trading off performance for clarity. Even if we did not rebuild the graph, writing such loops is highly suboptimal in terms of performance. Making this cost very visible discourages users from accidentally writing badly performing code.

A schematic of `Estimator` can be found in Figure 1. Below, we first describe how to provide inputs to the `train`, `evaluate`, and `predict` methods using input functions. Then we discuss model specification with `model_fn`, followed by how to specify outputs within the `model_fn` using Heads.
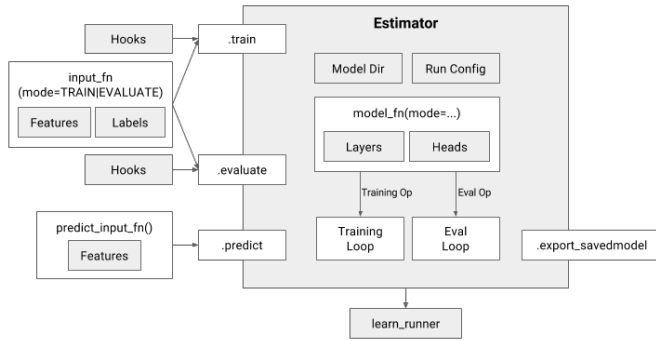

Figure 1: Simplified overview of the Estimator interface.

Specifying inputs with input_fn. The methods `train`, `evaluate`, and `predict` all take an input function, which is expected to produce two dictionaries: one containing `Tensors` with inputs (features), and one containing `Tensors` with labels. Whenever a method of `Estimator` is called, a new graph is created, the `input_fn` passed as an argument to the method call is called to produce the input pipeline of the `Estimator`, and then the `model_fn` is called with the appropriate mode argument to build the actual model graph. Decoupling the core model from input processing allows users to easily swap datasets. If used in larger infrastructure, being able to control the inputs completely is very valuable to downstream frameworks. A typical `input_fn` has the following form:

```
1  def my_input_fn(file_pattern):
2      feature_dict = learn.io.read_batch_features(
3          # path to data in tf.Example format
4          file_pattern=file_pattern,
5          batch_size=BATCH_SIZE,
6          # whether sparse or dense ...
7          features=FEATURE_SPEC,
8          # such as TFRecordReader
9          reader=READER,
10         ...)
11
12 estimator.train(input_fn=lambda:
13   my_input_fn(TRAINING_FILES), ...)
14 estimator.evaluate(input_fn=lambda:
15   my_input_fn(EVAL_FILES), ...)
```

Specifying the model with model_fn. We chose to configure `Estimator` with a single callback, the `model_fn`, which returns ops for training, evaluation, or prediction, depending on which graph is being requested (which method of `Estimator` is being called). For example, if the `train` method is called, `model_fn` will be called with an argument `mode=TRAIN`, which the user can then use to build a custom graph in the knowledge that it is going to be used for training.

Conceptually, three entirely different graphs can be built, and different information is returned, depending on the mode parameter representing the called method. Nevertheless, we found it useful to require only a single function for configuration. One of the main sources of error in production systems is training/serving skew. One type of training/serving skew happens when a different model is trained than is later served in production. Of course, models are routinely trained slightly differently than they are served. For instance, dropout and batch normalization layers are only active during training. However, it is easy to make mistakes if one has to rewrite the whole model three times. Therefore we chose to require a single function, effectively encouraging the model developer to write the model only once. For complex models, appropriate Python conditionals can be used to ensure that legitimate differences are explicitly represented in the model. A typical `model_fn` for a simple model may look like this:

```
1  def model_fn(features, target, mode, params):
2    predictions = tf.stack(tf.fully_connected,
3                           [50, 50, 1])
4    loss = tf.losses.mean_squared_error(target,
5                                        predictions)
6    train_op = tf.train.create_train_op(
7      loss, tf.train.get_global_step(),
8      params['learning_rate'], params['optimizer'])
9    return EstimatorSpec(mode=mode,
10                        predictions=predictions,
11                        loss=loss,
12                        train_op=train_op)
```

Specifying outputs with Heads. The Head API is an abstraction for the part of the model behind the last hidden layer. The key goals of the design are to simplify writing `model_fn`, to be compatible with a wide range of models, and to simplify supporting multiple heads. A `Head` knows how to compute loss, relevant evaluation metrics, predictions and metadata about the predictions that other systems (like serving, model validation) can use. To support different types of models (e.g., DNN, linear, Wide & Deep [9], gradient boosted trees, etc.), Head takes logits and labels as input and generates Tensors for loss, metrics, and predictions. Heads can also take the activation of the last hidden layer as input to support DNN with large number of classes where we want to avoid computing the full logit Tensor. A typical `model_fn` for a simple single objective model may look like this:

```
1  def model_fn(features, target, mode, params):
2    last_layer = tf.stack(tf.fully_connected,
3                          [50, 50])
4    head = tf.multi_class_head(n_classes=10)
5    return head.create_estimator_spec(
6        features, mode, last_layer,
7        label=target,
8        train_op_fn=lambda loss:
9          my_optimizer.minimize(
10             loss, tf.train.get_global_step()))
```

The abstraction is designed in a way that combining multiple Heads for multi objective learning is as simple as creating a special type of `Head` with a list of other heads. Model functions can take `Head` as a parameter while remaining agnostic to what kind of `Head` they are using. A typical `model_fn` for a simple model with two multi class objectives can look like this:

```
def model_fn(features, target, mode, params):
  last_layer = tf.stack(tf.fully_connected,
                        [50, 50])
  head1 = tf.multi_class_head(n_classes=2,
            label_name='y', head_name='h1')
  head2 = tf.multi_class_head(n_classes=10,
            label_name='z', head_name='h2')
  head = tf.multi_head([head1, head2])
  return head.create_model_fn_ops(features,
      features, mode, last_layer,
    label=target,
    train_op_fn=lambda loss:
      my_optimizer.minimize(
          loss, tf.train.get_global_step())
```

Executing computations. Once the graph is built, the `Estimator` then initializes a Session, prepares it appropriately, and runs the training loop, evaluation loop, or iterates over the inputs to produce predictions.

Most machine learning algorithms are iterative nonlinear optimizations, and therefore have a particularly simple algorithmic form: a single loop which runs the same computation over and over again, with different input data in each iteration. When used during training, this is called the training loop. In evaluation using mini-batches, much the same structure is used, except that variables are not updated, and typically, more metrics than just the loss are computed.

An idealized training loop implemented in TensorFlow is simple: start a `Session`, then run a training op in a loop. However, we have to at least initialize variables and special data structures like tables which are used in embeddings. Queue runners (implemented as Python threads) have to be started, and should be stopped at the end to ensure a clean exit. Summaries (which provide data to the TensorBoard visualization tool) have to be computed and written to file. The real challenge begins when distributed training is taken into account. While TensorFlow takes care of distribution of the computation and communication between workers, it requires many coordinated steps before a model can be successfully trained. The distributed computation introduces a number of opportunities for users to make mistakes: certain variables must be initialized on all workers, most only on one. The model state should be saved periodically to ensure that the computation can recover when workers go down, and needs to be recovered safely when they restart. End-of-input signals have to be handled gracefully.

Because the training loop is so ubiquitous, a good implementation removes a lot of duplicated user code. Because it is simple only in theory, we can remove a source of error and frustration for users. Therefore, `Estimator` implements and controls the training loop. It automatically assigns `Variables` to parameter servers to simplify distributed computation, and it gives the user only limited access to the underlying TensorFlow primitives. Users must specify the graph, and the op(s) to run in each iteration, and they may override the device placement.

Code injection using Hooks. Hooks make it impossible to implement advanced optimization techniques that break the simple loop abstraction in a safe manner. They are also useful for custom processing that has to happen alongside the main loop, for recordkeeping, debugging, monitoring or reporting. Hooks let users define custom behaviour at `Session` creation, before and after each iteration, and at the end of training. They also let users add ops other than those specified by the `model_fn` to be run within the same `Session.run` call. For example, a user who wants to train not for a given number of steps, but a given amount of wall time, could implement a `Hook` as follows:

```
class TimeBasedStopHook(tf.train.SessionRunHook):
  def begin(self):
    self.started_at = time.time()
  def after_run(self, run_context, run_values):
    if time.time() - self.started_at >= TRAIN_TIME:
      run_context.request_stop()
```

Hooks are activated by passing them to the `train` call. When the `Hook` shown above is passed to `train`, the model training will end after the set time. Much of the functionality that `Estimator` provides (for instance, summaries, step counting, and checkpointing) is internally implemented using such Hooks.

3.3 Canned Estimators

There are many model architectures commonly used by researchers and practitioners. We decided to provide those architectures as canned `Estimators` so that users don't need to rewrite the same models again and again. Canned `Estimators` are a good example of how to use `Estimator` itself. They are direct subclasses of `Estimator` that only override their constructors. As such, users of canned `Estimators` would only need to know how to use an `Estimator`, and how to configure the canned `Estimator`. This means that canned `Estimators` are mainly restricted to define a canned `model_fn`. There are two main reasons behind this restrictive design. First, we are expecting an increasing number of canned `Estimators` to be implemented. To minimize the cognitive load on users, all these canned `Estimators` should behave identically. Second, this restriction makes the canned `Estimator` developer a user of `Estimator`. This leads to an implicit comprehensive flexibility test of our API.

Neural networks rely on operations which take dense `Tensors` and output dense `Tensors`. Many machine learning problems have sparse features such as query keywords, product id, url, video id, etc. For models with many inputs, specifying how these features are attached to the model often consumes a large fraction of the total setup time. Based on our experience, one of the most error prone parts of building a model is converting these features into a single dense `Tensor`.

We offer the `FeatureColumn` abstraction to simplify input ingestion. `FeatureColumns` are a declarative way of specifying inputs. Canned `Estimators` take `FeatureColumns` as a constructor argument and handle the conversion of sparse or dense features of all types to a dense `Tensor` usable by the core model. As an example, the following code shows a canned `Estimator` implementation for the Wide & Deep architecture [9]. The deep part of the model uses embeddings while the linear part uses the crosses of base features.

```
1   # Define wide model features and crosses.
2   query_x_docid = crossed_column(
3       ["query", "docid"], num_buckets)
4   wide_cols = [query_x_docid, ...]
5
6   # Define deep model features and embeddings.
7   query = categorical_column_with_hash_bucket(
8       "query", num_buckets)
9   docid = categorical_column_with_hash_bucket(
10      "docid", num_buckets)
11  query_emb = embedding_column(query, dimension=32)
12  docid_emb = embedding_column(docid, dimension=32)
13  deep_cols = [query_emb, docid_emb, ...]
14  # Define model structure and start training.
15  estimator = DNNLinearCombinedClassifier(
16      wide_cols, deep_cols,
17      dnn_hidden_units=[500, 200, 100])
18  estimator.train(input_fn, ...)
```

4 DISTRIBUTED EXECUTION

With the built-in functionalities and utilities mentioned above, `Estimators` are ready for training, evaluating and exporting the model on a single machine. For production usages and models with large amounts of training data, utilities for distributed execution are also provided together with Estimators, which takes the advantage of TensorFlow's distributed training support. The core of distributed execution support is the `Experiment` class, which groups the `Estimator` with two input functions for training and evaluation. The architecture is summarized in Figure 2.
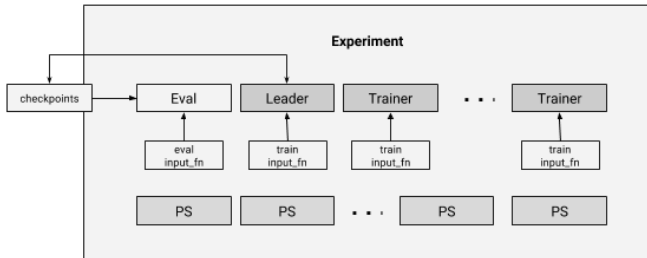

Figure 2: Simplified overview of the Experiment interface.

In each TensorFlow cluster, there are several parameter servers and several worker tasks. Most workers are handing the training process, which basically calls the `Estimator train` method with the training `input_fn`. One of the workers is designated leader and is responsible for managing checkpoints and other maintenance work. Currently, the primary mode of replica training in Tensor-Flow Estimators is between-graph replication and asynchronous training. However, it could be easily extended to support other replicated training settings. With this architecture, gradient descent training can be executed in parallel.

We have evaluated scaling of TensorFlow Estimators by running different numbers of workers with fixed numbers of parameter servers. We trained a DNN model on a large internal recommendation dataset (100s of billions of examples) for 48 hours and present average number of training steps per second. Figure 3 shows that we achieve almost linear scaling of global steps per second with the number of workers.

There is a special worker handling the evaluation process for the `Experiment` to evaluate the performance and export the model. It runs in a continuous loop and calls the `Estimator` evaluate method

with the evaluation `input_fn`. In order to avoid race conditions and inconsistent model parameter states, the evaluation process always begins with loading the latest checkpoint and calculates the evaluation metrics based on the model parameters from that checkpoint. As a simple extension, the `Experiment` also supports the evaluation with the training `input_fn`, which is very useful to detect overfitting in deep learning in practice.

Furthermore, we also provide utilities, `RunConfig` and `runner`, to ease the way of using and configuring `Experiment` in a cluster for distributed training. `RunConfig` holds all the execution related configuration the `Experiment/Estimator` requires, including cluster specification, model output directory, checkpoints configuration, etc. In particular, `RunConfig` specifies the task type of the current task, which allows all tasks sharing the same binary but running a different mode, such as parameter server, training, or continual evaluation. The `runner` is simply a utility method to construct the `RunConfig`, e.g., by parsing the environment variable, and execute the `Experiment/Estimator` with that `RunConfig`. With this design, `Experiment/Estimator` could be easily shared by various execution frameworks including end-to-end machine learning pipelines [6] and even hyper-parameters tuning.
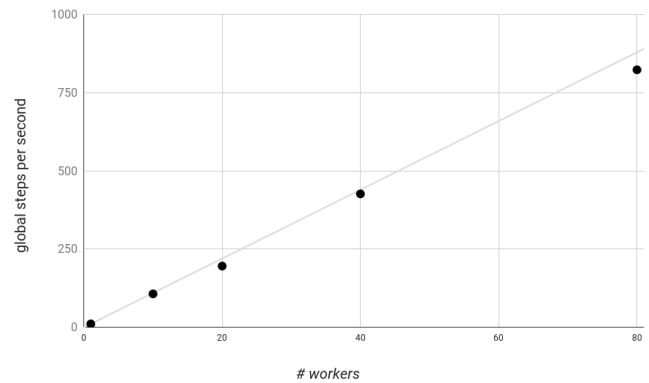

Figure 3: Measuring scaling of DNN model training implemented with TensorFlow Estimators, varying the number of workers. Shown are measurements as well as the theoretical perfect linear scaling.

5 CASE STUDIES AND ADOPTION

For machine learning practitioners within Google, this framework has dramatically reduced the time to launch a working model. Before TensorFlow Estimators, the typical model construction cycle involved writing custom TensorFlow code to ingest and represent features (sparse features were especially tricky), construction of the model layers itself, establishing training and validation loops, productionizing the system to run on distributed training clusters, adding evaluation metrics, debugging training NaNs, and debugging poor model quality.

TensorFlow Estimators simplify or automate all but the debugging steps. Estimators give the practitioner confidence that, when debugging NaNs or poor quality, these problems arise either from

their choice of hyperparameters or their choice of features — but not a bug in the wiring of the model itself.

When TensorFlow Estimators became available, several TensorFlow models under development greatly benefited from transitioning to the framework. One multiclass classification model attained 37% better model accuracy by switching from a custom model that performed multiple logistic regressions to a standard `Estimator` that properly used a softmax cross-entropy loss — the switch also reduced lines of code required from 800 to 200. A different TensorFlow CTR model was stuck in the debugging phase for several weeks, but was transitioned to the framework within two days and achieved launchable offline metrics.

It is worth noting that using `Estimators` and the associated machinery also requires considerably less expertise than would be required to implement the equivalent functionality from scratch. Recently, a cohort of Google data scientists with limited Python experience and no TensorFlow experience were able to bootstrap real models in a two-day class setting.

5.1 Experience in YouTube Watch Next

Using TensorFlow Estimators, we have productionized and launched a deep model (`DNNClassifier`) in the Watch Next video recommender system of YouTube. Watch Next is a product recommending a ranked set of videos for a user to choose from after the user is done watching the current video. One unique aspect about our model is that the model is trained over multiple days, with the training data being continuously updated.

Our input features consist of both sparse categorical features and real-valued features. The sparse features are further transformed into embedding columns before being fed into the hidden layers. The `FeatureColumn` API greatly simplifies how we construct the input layer of our model. Additionally, the train-to-serve support of TensorFlow Estimators considerably reduced the engineering effort to productionize the Watch Next model. Furthermore, the `Estimator` framework made it easy to implement new `Estimators` and experiment with new model architectures such as multiple-objective learning to accommodate specific product needs.

The initial version of the model pipeline was developed using low-level TensorFlow primitives prior to the release of Estimators. While debugging why the model quality failed to match our expectation, we discovered critical bugs related to how the network layers were constructed and how the input data were processed.

As an early adopter, Watch Next prompted the development of missing features such as shared embedding columns. Shared embedding columns allow multiple semantically similar features to share a common embedding space, with the benefit of transfer learning across features and smaller model size.

5.2 Adoption within Google

Software engineers at Google have a variety of choices for how to implement their machine learning models. Before we developed the higher-level framework in TensorFlow, engineers were effectively forced to implement one-off versions of the components in our framework.

An internal survey has shown that, since we introduced this framework and `Estimators` less than a year ago, close to 1,000 `Estimators` have been checked into the Google codebase and more than 120,000 experiments have been recorded (an experiment in this context is a complete training run; not all runs are recorded, so the true number is significantly higher). Of those, over half (57%) use implementations of canned `Estimators` (e.g., `LinearClassifier`, `DNNLinearCombinedRegressor`). There are now over 20 Estimator classes implementing various standard machine learning algorithms in the TensorFlow code base. Examples include `DynamicRnnEstimator` (implementing dynamically unrolled RNNs for classification or regression problems) and `TensorForestEstimator` (implementing random forests). Figure 4 shows the current distribution of `Estimator` usage. This framework allowed teams to build high-quality machine learning models within an average of one engineer-week, sometimes as fast as within 2 hours. 74% of respondents say that development with this framework is faster than other machine learning APIs they used before. Most importantly, users note that they can focus their time on the machine learning problem as opposed to the implementation of underlying basics. Among existing users, quick ramp-up, ease of use, reuse of common code and readability of a commonly used framework are the most frequently mentioned benefits.
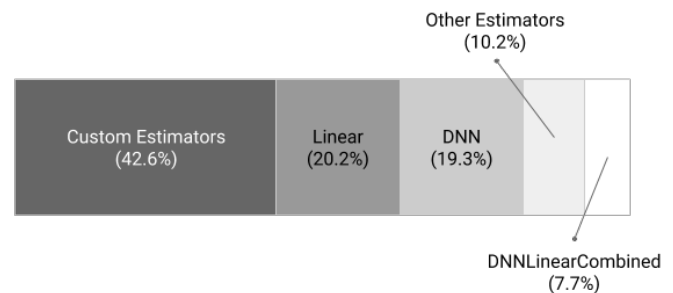

Figure 4: Current usage of Estimators at Google.

REFERENCES

[1] *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.
[2] 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.
[3] Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, Ryan Hoens, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Marko Padmilac, Hari Parthasarathi, Baolin Peng, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Yongqiang Wang, Huaming Wang, Kaisheng Yao, Dong Yu, Yu Zhang, and Geoffrey Zweig. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit.* Technical Report MSR-TR-2014-112. http://research.microsoft.com/apps/pubs/default.aspx?id=226641
[4] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman,

Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

[5] Amazon. 2016. Dsstne. https://github.com/amznlabs/amazon-dsstne. (2016).

[6] 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, and Martin Zinkevich. 2017. The Anatomy of a Production-Scale Continuously-Training ML Platform. KDD [under review]. (2017).

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). http://arxiv.org/abs/1603.02754

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). http://arxiv.org/abs/1512.01274

[9] 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.

[10] François Chollet. 2015. keras. https://github.com/fchollet/keras. (2015).

[11] Ronan Collobert, Samy Bengio, and Johnny Marithoz. 2002. Torch: A Modular Machine Learning Software Library. (2002).

[12] The Scipy community. 2012. *NumPy Reference Guide.* SciPy.org. http://docs.scipy.org/doc/numpy/reference/

[13] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12)*. Curran Associates Inc., USA, 1223–1231. http://dl.acm.org/citation.cfm?id=2999134.2999271

[14] Deeplearning4j Development Team. 2016. Deeplearning4j: Open-source distributed deep learning for the JVM, Apache Software Foundation License 2.0. http://deeplearning4j.org. (2016).

[15] Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, diogo149, Brian McFee, Hendrik Weideman, takacsg84, peterderivaz, Jon, instagibbs, Dr. Kashif Rasul, CongLiu, Britefury, and Jonas Degrave. 2015. Lasagne: First release. (Aug. 2015). DOI:http://dx.doi.org/10.5281/zenodo.27878

[16] Sergio Guadarrama and Nathan Silberman. 2016. TF Slim. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim. (2016).

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 675–678. DOI:http://dx.doi.org/10.1145/2647868.2654889

[18] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016), 1235–1241. http://dl.acm.org/citation.cfm?id=2946645.2946679

[19] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2825–2830. http://dl.acm.org/citation.cfm?id=1953048.2078195

[20] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a Next-Generation Open Source Framework for Deep Learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/papers/LearningSys_2015_paper_33.pdf

[21] Bart van Merriënboer, Dzmitry Bahdanau, Vincent Dumoulin, Dmitriy Serdyuk, David Warde-Farley, Jan Chorowski, and Yoshua Bengio. 2015. Blocks and Fuel: Frameworks for deep learning. *CoRR* abs/1506.00619 (2015). http://arxiv.org/abs/1506.00619