



Building Machine Learning models with TensorFlow

Data Engineering on Google Cloud Platform



©Google Inc. or its affiliates. All rights reserved. Do not distribute.
May only be taught by Google Cloud Platform Authorized Trainers.

Notes:

30 slides + 4 labs: 2.5 hours

Agenda

What Is TensorFlow? + Lab

TensorFlow for Machine Learning + Lab

Gaining more flexibility + Lab

Train and evaluate + Lab

A tensor is an N-dimensional array of data



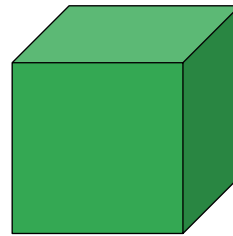
Rank 0
Tensor
scalar



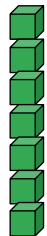
Rank 1
Tensor
vector



Rank 2
Tensor
matrix



Rank 3
Tensor



Rank 4
Tensor

So where **does** the name ('tensorflow') come **s** from?

In math, a simple number like 3 or 5 is called a scalar.

A vector is a one-dimensional array of numbers. In physics, a vector is something with magnitude and direction. But in Computer Science, we use vector to mean 1D arrays.

A two-dimensional array is a matrix.

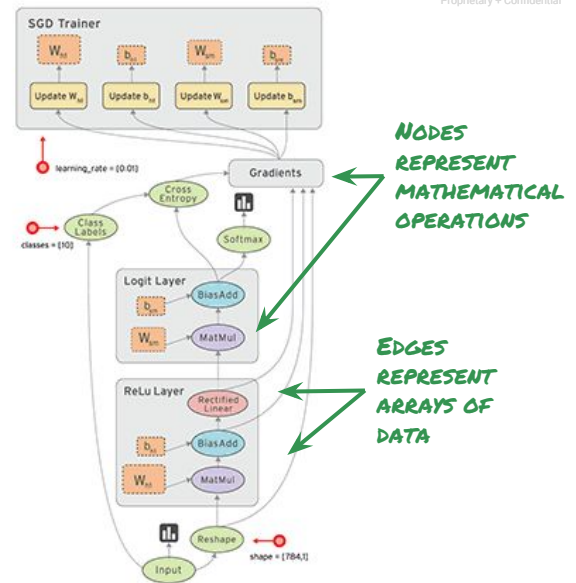
A three-dimensional array? We just call it **is** a 3D tensor.

So scalar, vector, matrix, 3D Tensor, 4D Tensor, etc.

A tensor is **an** n-dimensional array of data.

So, your data in TensorFlow are tensors.
They flow through the graph. Hence, TensorFlow.

TensorFlow is an open-source high-performance library for numerical computation that uses directed graphs



TensorFlow is an open-source high-performance library for numerical computation. Not just about machine learning. Any numeric computation. In fact, people have used TensorFlow for all kinds of GPU computing; for example, you can use TensorFlow to solve partial differential equations -- these are useful in domains like fluid dynamics. TensorFlow as a numeric programming library is appealing because you can write your computation code in a high-level language -- Python -- and have it be executed in a fast way.

The way TensorFlow works is that you create a directed graph (a DG) to represent your computation. In this schematic, the nodes represent mathematical operations -- things like adding, subtracting, and multiplying. Also more complex functions. Here, for example, softmax, and matrix-multiplication, etc. are mathematical operations that are part of the directed graph.

Connecting the nodes are the edges. The input and output of the mathematical operations. The edges represent arrays of data. Essentially, the result of computing the cross-entropy is one of the three inputs to the BiasAdd operation, and the output of the BiasAdd operation is sent along to the matrix-multiplication operation (MatMul). The other input to MatMul -- you need two inputs to a matrix multiplication -- the other input is a variable, the weight.

- Portable across GPUs, CPUs, mobile ...
- Developed at Google
- Uses data flow graphs: neural network training and evaluation can be represented as data flow graphs

<http://www.tensorflow.org/>

GIF: https://www.tensorflow.org/images/tensors_flowimg.gif

TensorFlow toolkit hierarchy

Run TF at scale

High-level “out-of-box” API
does distributed training



tf.estimator



Components useful when
building custom NN models



tf.layers, tf.losses, tf.metrics

Python API gives you full control



Core TensorFlow (Python)

C++ API is quite low-level



Core TensorFlow (C++)

TF runs on different hardware



CPU

GPU

TPU

Android

Cloud ML Engine

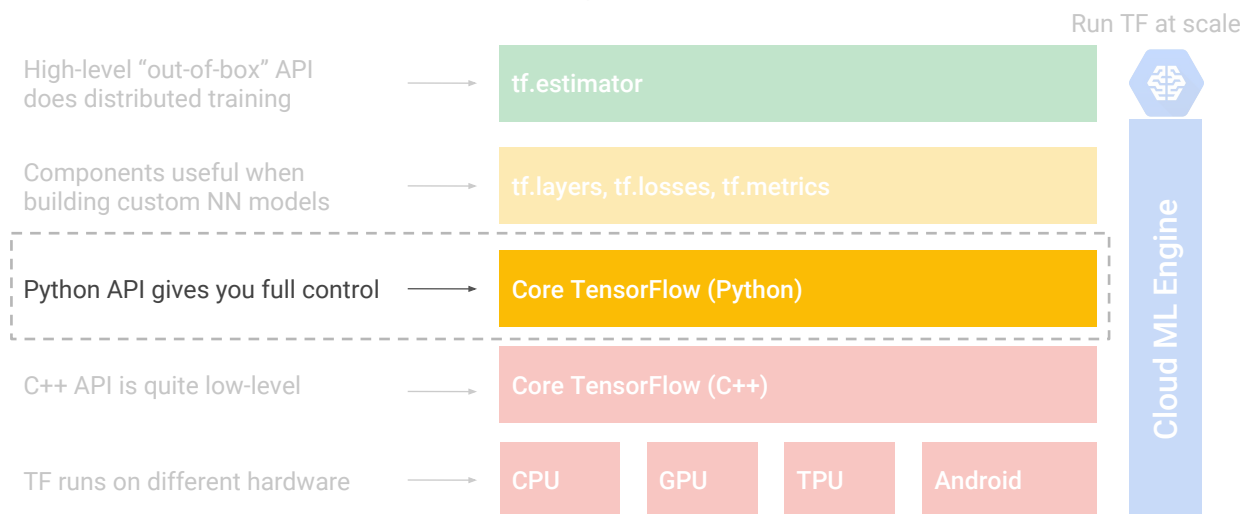
Google Cloud

Training and Certification

Notes:

Akin to how Java programs run on many different types of hardware because of the JVM. TensorFlow C++ plays the role of the JVM here, providing hardware instruction sets.

TensorFlow toolkit hierarchy



Notes:

Let's look at what the Python API does.

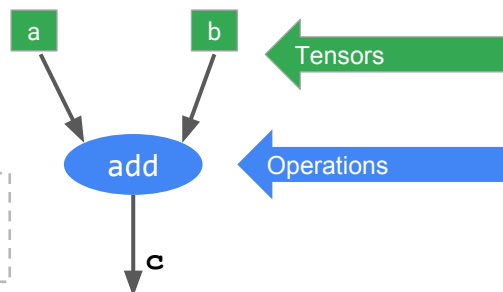
The Python API lets you build and run Directed Graphs

```
...
c = tf.add(a, b)
```

Build

```
session = tf.Session()
numpy_c = session.run(c, feed_dict= ...)
```

Run



Notes:

So, let's look at the code on this slide. At first glance, this looks just like, say, numpy. You want to add two tensors *a* and *b*. So, you write `tf.add(a, b)`. It returns a tensor *c*.

Unlike typical Python code, though, running the `tf.add()` doesn't execute it. It only builds the DG.

In the DG -- in the directed graph -- *a*, *b* and *c* are tensors. Add is an operation. In order to run this code, in order to execute the DG, you need to run it. And you run it as part of what is called a Session. So, you say you want the value of *c* and you ask the session to evaluate *c* for you.

That's what runs the DG and you get back a traditional numeric array in Python that contains the values for *c*.

Programming TensorFlow involves programming a DG. There are two steps.

First step: create a graph.

Second step: run it.

The graph definition is separate from the training loop because this is a lazy evaluation model. It minimizes the Python/C++ context switches and enables the computation to be very efficient. Conceptually, this is like writing a program, compiling it, and then running it on some data. Don't take the analogy too far, though. There is

no explicit compile phase here.

Note that `c` after you call `tf.add` is not the actual values -- you have to evaluate `c` in the context of a TensorFlow session to get a numpy array of values (`'numpy_c'`).

Note: We use the term **Directed Graph (DG)** rather than **Directed Acyclic Graph (DAG)** because TensorFlow supports cyclic graphs.

For more on this topic:

<https://stackoverflow.com/questions/37551389/cyclic-computational-graphs-with-tensorflow-or-theano/37551496#37551496>

<http://kias.dyndns.org/comath/33.html>

TensorFlow does lazy evaluation: you need to run the graph to get results*

numpy

```
a = np.array([5, 3, 8])
b = np.array([3, -1, 2])
c = np.add(a, b)
print c
```

```
[ 8  2 10]
```

***TF EAGER, HOWEVER,
ALLOWS YOU TO EXECUTE
OPERATIONS IMPERATIVELY**

Tensorflow

```
a = tf.constant([5, 3, 8])
b = tf.constant([3, -1, 2])
c = tf.add(a, b)
print c
```

Build

```
Tensor("Add_7:0", shape=(3,), dtype=int32)
```

```
with tf.Session() as sess:
    result = sess.run(c)
    print result
```

Run

```
[ 8  2 10]
```

<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/eager>

Notes:

So, to reiterate, TensorFlow does lazy evaluation. You write a DG. Then you run the DG in the context of a session to get results. Now, there is a different mode in which you can run TensorFlow, called `tf.eager`, where the evaluation is immediate and it is not lazy. But eager mode is typically not used in production programs; it's only for development. We'll look at `tf.eager` a little bit later in this course, but for the most part, we'll focus on the lazy evaluation paradigm. And almost all the code we create and run in production will be in lazy evaluation mode.

In numpy, which is what the majority lion's share of Python numeric software is written in, `a` and `b` are numpy arrays. Numpy gets its speed by being implemented in C. so when you call `np.add()`, that add gets done in C. But it does get done [is this correct?] when the CPU runs the line of code `np.add(a, b)` and the numpy array `c` gets populated with the sums. So, when you print "`c`", you get the 8, 2, 10. 8 is the sum of 5 and 3. Add 3 and -1 to get 2. Etc. The point is the `np.add()` is evaluated immediately.

Unlike with numpy, in TensorFlow `c` is not the actual values. Instead, `c` is a tensor -- you have to evaluate `c` in the context of a TensorFlow session to get a numpy array of values ('result').

So, when the CPU or GPU or other whatever hardware evaluates `tf.add(a, b)`, a Tensor gets created in the directed graph, in the DG. But the addition is not carried out until the `sess.run()` gets called.

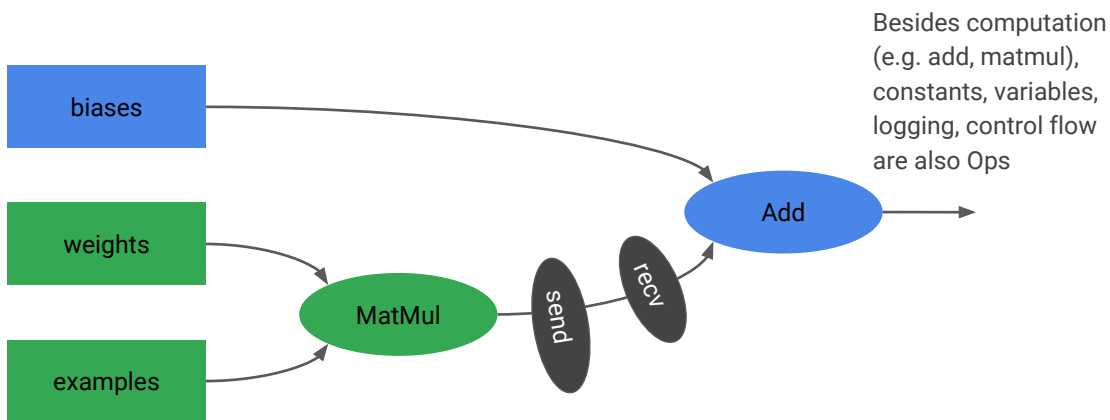
So, if we call `print c`, what gets printed out in the first box is the “debug” output of the `Tensor` class. It includes a system-assigned unique name for the node in the DG (“ADD_7”) and the shape and data type of the value that will show up when the DG is run.

After the session is run and `c` is evaluated in the context of the session, we can print the result and we’ll get `[8, 2, 10]`. So, two stages: a build stage and a run stage.

But why? Why does TensorFlow do lazy evaluation? That’s the next lesson.

<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/eager>

Graphs can be processed, compiled, remotely executed, assigned to devices



Notes:

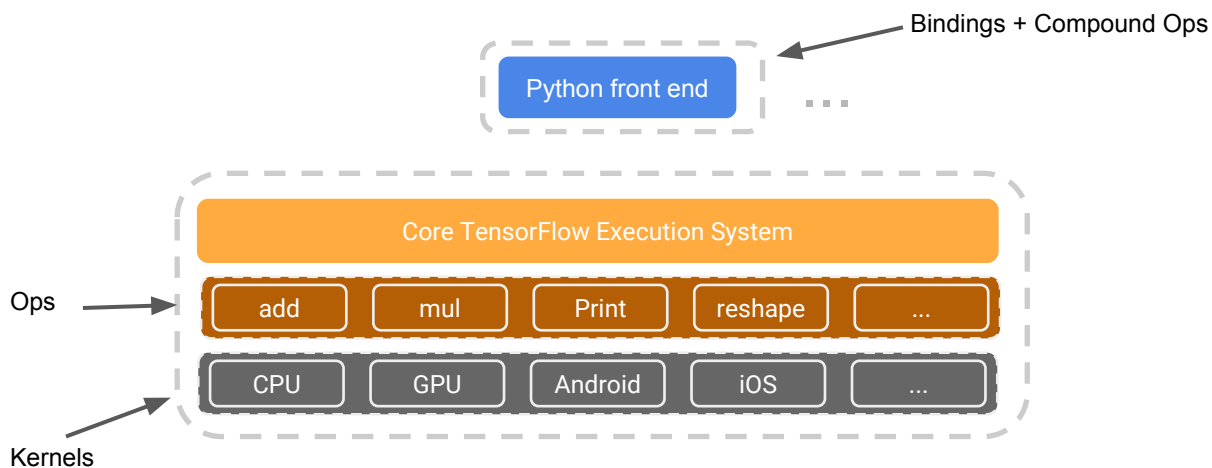
Example of each mentioned use-case:

Processed -- add quantization or data types, add debug nodes; create summaries to write values out so that Tensorboard can read ...

Compiled -- fuse ops to improve performance. For example, two consecutive add nodes can be fused into a single one.

Remotely executed, assigned to devices -- **note colors**, several parts of the graph can be on different devices, doesn't matter whether GPU or several computers. Automatically insert send/rcv nodes.

TensorFlow can distribute computation



Notes:

One key benefit of this model -- to be able to distribute computation across many machines, and many types of machines. No need to assign a Print op to a GPU.

Lab 2: Getting Started with TensorFlow

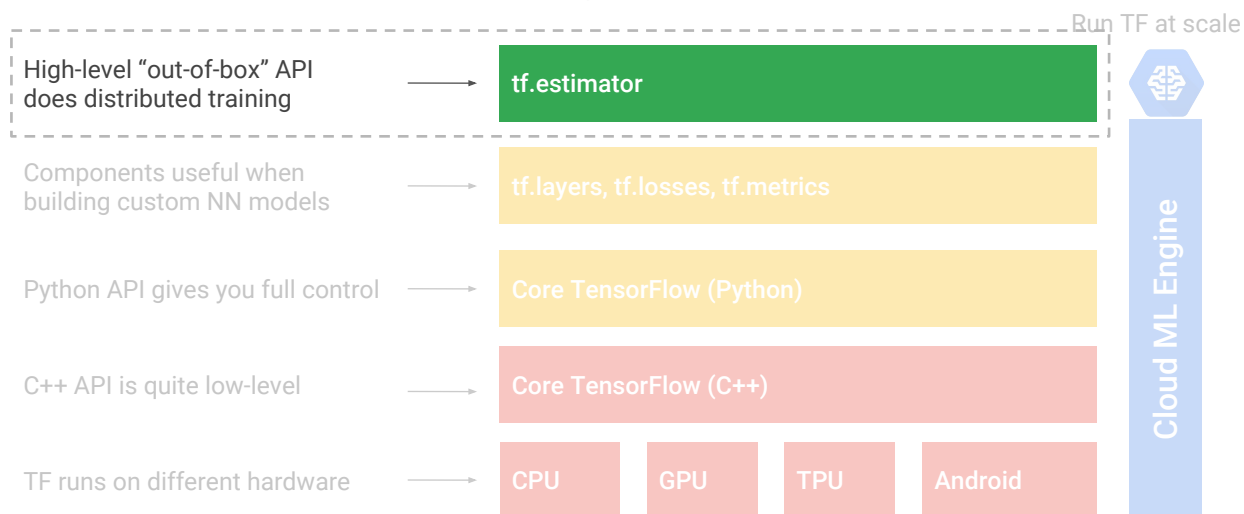
In this lab you will:

- Explore the TensorFlow Python API
 - Building a graph
 - Running a graph
 - Feeding values into a graph
 - Find area of a triangle using TensorFlow

Agenda

TensorFlow for Machine Learning + Lab

TensorFlow toolkit hierarchy



Notes:

Let's look at what the Python API does.

Working with Estimator API

Set up machine learning model

1. Regression or classification?
2. What is the label?
3. What are the features?



Carry out ML steps

1. Train the model
2. Evaluate the model
3. Predict with the model

Notes:

Two parts to it. One is static -- how to set the ML problem up.

The second is the ML steps you carry out.

Imagine that you want to create a ML model to predict cost of a house given the sq footage. Answer the first three questions.

The structure of an Estimator API ML model

```
import tensorflow as tf
#Define input feature columns
featcols = [
    tf.feature_column.numeric_column("sq_footage") ]

#Instantiate Linear Regression Model
model = tf.estimator.LinearRegressor(featcols, './model_trained')

#Train
def train_input_fn():
    features = {
        "sq_footage": tf.constant([1000, 2000]) }
    labels = tf.constant([50, 100]) # in thousands
    return features, labels
model.train(train_input_fn, steps=100)

#Predict
def pred_input_fn():
    features = {
        "sq_footage": tf.constant([1500, 1800]) }
    return features
out = trained.predict(pred_input_fn)
```

**SQUARE
FOOTAGE**

**ML
MODEL**

PRICE

Notes:

Conceptually, square footage shows up twice. Once as the placeholder ("feature_column") and next as an input feed ("input_fn").

Steps to define an Estimator API model

```
import tensorflow as tf
#Define input feature columns
featcols = [
    tf.feature_column.numeric_column("sq_footage") ]
```

1. SET UP FEATURE COLUMN

(OTHER TYPES EXIST: WE'LL LOOK AT THEM IN LATER CHAPTER)

```
#Instantiate Linear Regression Model
model = tf.estimator.LinearRegressor(featcols, './model_trained')
```

2. CREATE A MODEL, PASSING IN THE FEATURE COLUMNS

```
#Train
def train_input_fn():
    features = {
        "sq_footage": tf.constant([1000, 2000]) }
    labels = tf.constant([50, 100]) # in thousands
    return features, labels
model.train(train_input_fn, steps=100)
```

3. WRITE INPUT_FN (RETURNS FEATURES, LABELS) FEATURES IS A DICT

```
#Predict
def pred_input_fn():
    features = {
        "sq_footage": tf.constant([1500, 1800]) }
    return features
out = trained.predict(pred_input_fn)
```

Steps to do Machine Learning with model

```
import tensorflow as tf
#Define input feature columns
featcols = [
    tf.feature_column.numeric_column("sq_footage") ]

#Instantiate Linear Regression Model
model = tf.estimator.LinearRegressor(featcols, './model_trained')
```

```
#Train
def train_input_fn():
    features = {
        "sq_footage": tf.constant([1000, 2000]) }
    labels = tf.constant([50, 100]) # in thousands
    return features, labels
model.train(train_input_fn, steps=100)
```

4. TRAIN THE MODEL

```
#Predict
def pred_input_fn():
    features = {
        "sq_footage": tf.constant([1500, 1800]) }
    return features
out = trained.predict(pred_input_fn)
```

5. USE TRAINED MODEL TO PREDICT

Notes:

`estimator.predict()` returns a generator function, not the actual values. You have to iterate through it to get the values. Or use `list()` to get all the values in one go.

Beyond linear regression with Estimators

- Deep neural network

```
model = DNNRegressor(feature_columns=[...],  
                      hidden_units=[128, 64, 32])
```

- Classification

```
model = LinearClassifier(feature_columns=[...])  
model = DNNClassifier(feature_columns=[...], hidden_units=[...])
```

Notes:

The hourglass indicates that we'll cover this later in the course.

<https://pixabay.com/en/hourglass-sandglass-patience-time-297765/> (cc0)

Lab 3: Machine Learning using tf.estimator

In this lab you will:

- Read from Pandas Dataframe into `tf.constant`
- Create feature columns for estimator
- Linear Regression with `tf.Estimator` framework
- Deep Neural Network regression
- Benchmark dataset

Notes:

It's easy to ML --- just write 4 lines of Python code, throw some data into the hopper and voila!

But wait ... the ML model doesn't perform that well. More effort is needed.

If you are curious about why the ML model is performing so poorly: the problem is nonlinear in the inputs, but we are not scaling the inputs, and so the optimizer is not able to find the optimal weights. Also, the form of the solution requires some feature engineering on the raw inputs. Some preprocessing and feature engineering will enable us to get to better performance -- it's important to realize that if you don't do the necessary legwork, just making the model more complex is not going to work.

(the .csv files have on the order of 7700 samples, which should be enough for the models we are trying, so it's not a case of not having enough data).

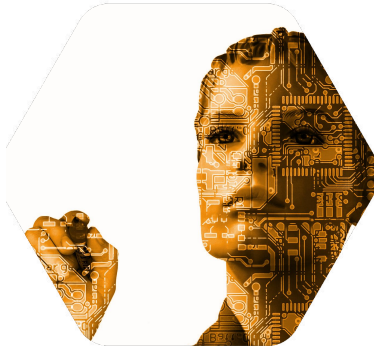
Agenda

Gaining more flexibility + Lab

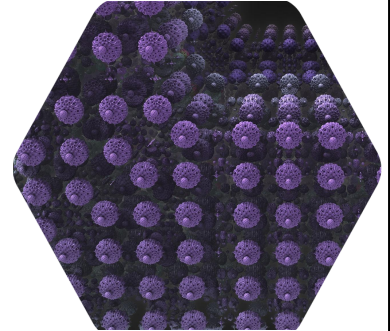
What's left? Ways to build effective ML



Big Data



Feature Engineering



Model Architectures

Notes:

<https://pixabay.com/en/large-data-dataset-word-895563/> (cc0)

<https://pixabay.com/en/fractal-complexity-render-3d-1232494/> (cc0)

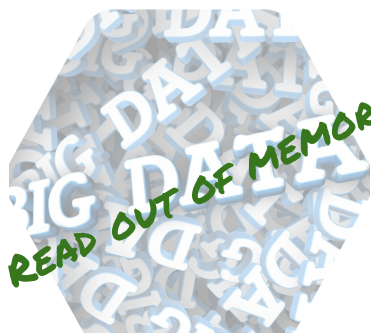
<https://pixabay.com/en/robot-artificial-intelligence-woman-507811/> (cc0)

Now that you know *how* to build ML, let's learn how to do it well in the rest of the course.

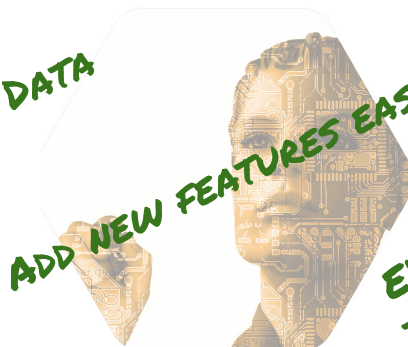
Ordered from easiest to most difficult.

Before we can do that though, we need to refactor our simple TensorFlow model.

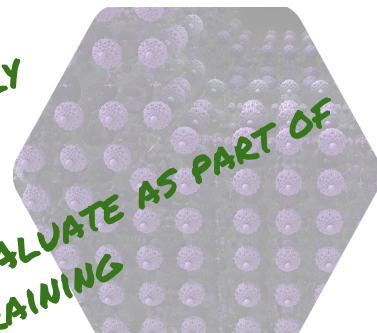
We need to refactor our Estimator model



Big Data



Feature Engineering



Model Architectures

Notes:

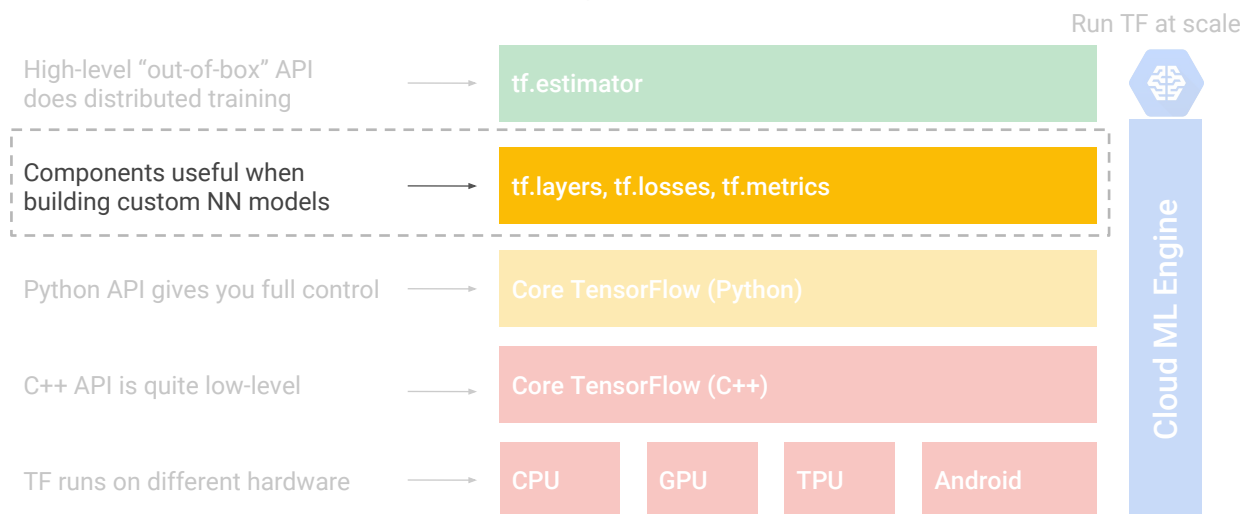
Refactor == improve design of model without adding any new capability. That's what we are going to do in the next two labs.

To handle big data, we need to be able to read out-of-memory datasets.

To do feature engineering, we need to be able add new feature columns easily.

To try out new model architectures, we have to move evaluation as a bonafide part of training, so that we can test other architectures systematically.

TensorFlow toolkit hierarchy



Notes:

We'll use the next lower level down in refactoring.

To read sharded CSV files, create a TextLineDataset giving it a function to decode the CSV into features, labels

```
CSV_COLUMNS =  
['amount', 'pickuplon', 'pickuplat', 'dropofflon', 'dropofflat', 'passengers']  
LABEL_COLUMN = 'amount'  
DEFAULTS = [[0.0], [-74.0], [40.0], [-74.0], [40.7], [1.0]]  
  
def read_dataset(filename, mode, batch_size=512):  
    def decode_csv(value_column):  
        columns = tf.decode_csv(value_column, record_defaults=DEFAULTS)  
        features = dict(zip(CSV_COLUMNS, columns))  
        label = features.pop(LABEL_COLUMN)  
        return features, label  
  
    dataset = tf.data.TextLineDataset(filename).map(decode_csv)  
  
    ...  
    return ...
```

Code for reading a CSV file

Directly supports a list of filenames if you have sharded files

Repeat the data and send it along in chunks

```
def read_dataset(filename, mode, batch_size=512):  
    ...  
  
    dataset = tf.data.TextLineDataset(filename).map(decode_csv)  
    if mode == tf.estimator.ModeKeys.TRAIN:  
        num_epochs = None # indefinitely  
    else:  
        num_epochs = 1 # end-of-input after this  
    dataset = dataset.repeat(num_epochs).batch(batch_size)  
  
    return dataset.make_one_shot_iterator().get_next()
```

Code for reading a CSV file

Directly supports a list of filenames if you have sharded files

Lab 4: Refactoring to add batching and feature-creation

In this lab you will:

- Refactor the input
- Refactor the way the features are created
- Create and train the model
- Evaluate model

Agenda

Train and evaluate + Lab

We need to make our ML pipeline more robust



In our Estimator examples so far, we:

1. ran the training_op for num_steps or num_epochs iterations
2. saved checkpoints during training
3. Used final checkpoint as model



For realistic, real-world ML models, we need to:

1. Use a fault-tolerant distributed training framework
2. Choose model based on validation dataset
3. Monitor training, especially if it will take days
4. Resume training if necessary

Notes:

<https://pixabay.com/en/hang-out-plush-toys-kermit-1521663/> (cc0)
<https://pixabay.com/en/london-england-hdr-boats-ships-123778/> (cc0)

Our tf.learn examples have been a toy and is okay if the only thing we are going to be handling are plush-toys. We can't just hang a clothesline and call it a bridge. In the real world, we need to make things more robust.

Monitoring: Experiment will export summaries, so that we can see them in TensorBoard.

Shuffling is important for distributed training

```
def read_dataset(filename, mode, batch_size=512):  
    ...  
    dataset = tf.data.TextLineDataset(filename).map(decode_csv)  
    if mode == tf.estimator.ModeKeys.TRAIN:  
        num_epochs = None # indefinitely  
        dataset = dataset.shuffle(buffer_size=10*batch_size)  
    else:  
        num_epochs = 1 # end-of-input after this  
        dataset = dataset.repeat(num_epochs).batch(batch_size)  
    return dataset.make_one_shot_iterator().get_next()
```

This is so that different workers don't see the same data in the same order.

Estimator comes with a method that handles distributed training and evaluation

```
estimator = tf.estimator.LinearRegressor(  
    model_dir=output_dir,  
    feature_columns=feature_cols)  
  
...  
  
tf.estimator.train_and_evaluate(estimator,  
    train_spec,  
    eval_spec)
```

PASS IN:

1. **ESTIMATOR**
2. **TRAIN SPEC**
3. **EVAL SPEC**

Distribute the graph

Share variables

Evaluate every once in a while

Handle machine failures

Create checkpoint files

Recover from failures

Save summaries for TensorBoard

Which begs the question ... what is a train-spec and eval-spec?

The TrainSpec consists of the things that used to be passed into the `train()` method

```
Train_spec = tf.estimator.TrainSpec(  
    input_fn=read_dataset('gs://.../train*'),  
    mode=tf.contrib.learn.ModeKeys.TRAIN),  
    ...  
    tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

THINK "STEPS", NOT "EPOCHS" WITH PRODUCTION-READY, DISTRIBUTED MODELS.

1. GRADIENT UPDATES FROM SLOW WORKERS COULD GET IGNORED

2. WHEN RETRAINING A MODEL WITH FRESH DATA, WE'LL RESUME FROM EARLIER NUMBER OF STEPS (AND CORRESPONDING HYPER-PARAMETERS)

The EvalSpec controls the evaluation and the checkpointing of the model since they happen at the same time

```
exporter = ...
eval_spec=tf.estimator.EvalSpec(
    input_fn=read_dataset('gs://.../valid*',
                          mode=tf.contrib.learn.ModeKeys.EVAL),
    steps=None,
    start_delay_secs=60, # start evaluating after N seconds
    throttle_secs=600, # evaluate every N seconds
    exporters=exporter)

tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

The way it works is that the training loop saves the model into a checkpoint
The eval loop restores the model from the checkpoint and uses it to evaluate the model.

So checkpointing is needed for evaluation.

But when we checkpoint, we want to make sure that the model we save is complete and can be used for prediction ...

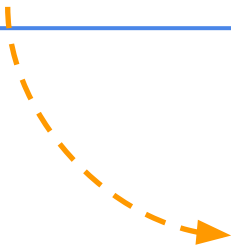
Because it is possible that in the steps that follow, the model will start to overfit.

In that case, we'll want to use the current checkpoint as the best model.

Hence, we think of checkpointing as "exporting" and we'll use an exporter to do this.

Change logging level from WARN

```
tf.logging.set_verbosity(tf.logging.INFO)
```



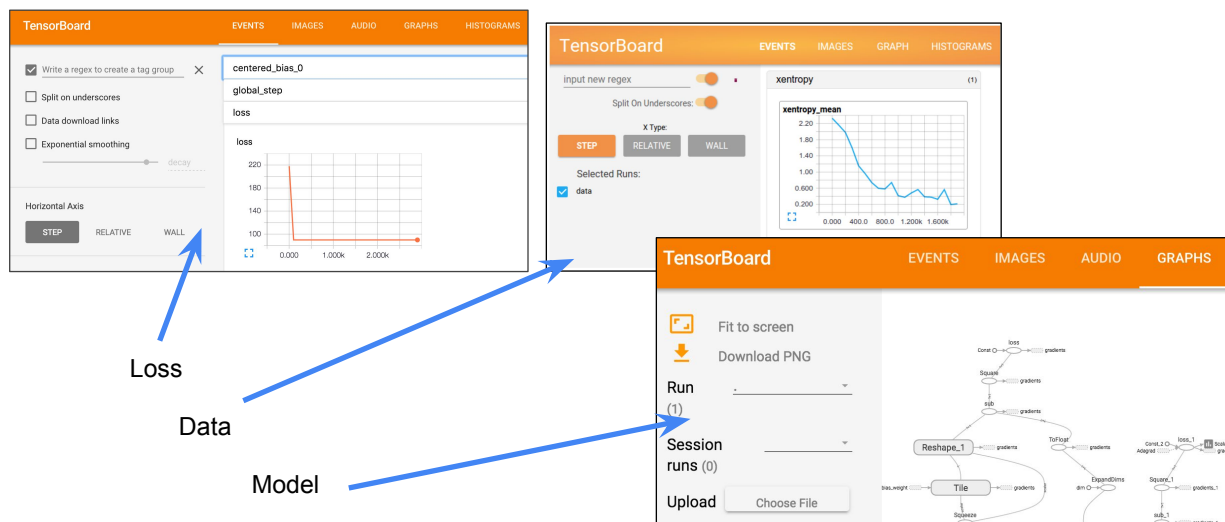
```
INFO:tensorflow:Transforming feature_column _RealValuedColumn(column_name=
alue=None, dtype=tf.float32)
INFO:tensorflow:Create CheckpointSaver
INFO:tensorflow:Step 1: loss = 218.036
INFO:tensorflow:Step 101: loss = 89.9517
INFO:tensorflow:Step 201: loss = 89.9487
INFO:tensorflow:Saving checkpoints for 300 into taxi_model/model.ckpt.
INFO:tensorflow:Step 301: loss = 89.9468
INFO:tensorflow:Step 401: loss = 89.9453
INFO:tensorflow:Step 501: loss = 89.944
INFO:tensorflow:Saving checkpoints for 600 into taxi_model/model.ckpt.
INFO:tensorflow:Step 601: loss = 89.9429
INFO:tensorflow:Step 701: loss = 89.9419
INFO:tensorflow:Step 801: loss = 89.941
INFO:tensorflow:Saving checkpoints for 900 into taxi_model/model.ckpt.
INFO:tensorflow:Step 901: loss = 89.9402
```

Notes:

The default is WARN; change it to INFO to see logs as TF trains.

The levels are DEBUG, INFO, WARN, ERROR, and FATAL.

Use TensorBoard to monitor training



Notes:

TensorBoard is a collection of visualization tools designed specifically to help you visualize TensorFlow.

- TensorFlow graph
- Plot quantitative metrics
- Pass and graph additional data

Events at top left shows "loss".

Graphs at bottom right shows the linear model graph as built by TensorFlow.

Point TensorBoard at model output directory.

https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard

Lab 5: Distributed training and monitoring

In this lab you will:

- Create features out of input data
- Train and evaluate
- Monitor with Tensorboard

Resources

| | |
|--|---|
| tf.contrib.learn API | https://www.tensorflow.org/versions/master/api_docs/python/contrib.learn.html |
| TensorFlow (all) | https://www.tensorflow.org/api_docs/python/index.html |
| Understanding neural networks with TensorFlow playground | https://cloud.google.com/blog/big-data/2016/07/understanding-neural-networks-with-tensorflow-playground |
| TensorFlow examples | https://github.com/aymericdamien/TensorFlow-Examples |
| TensorFlow MNIST | https://www.youtube.com/watch?v=vq2nnJ4g6N0&t=3686s |
| Another TF Datalab | https://github.com/kazunori279/TensorFlow-Intro/ |



cloud.google.com

Images by Google