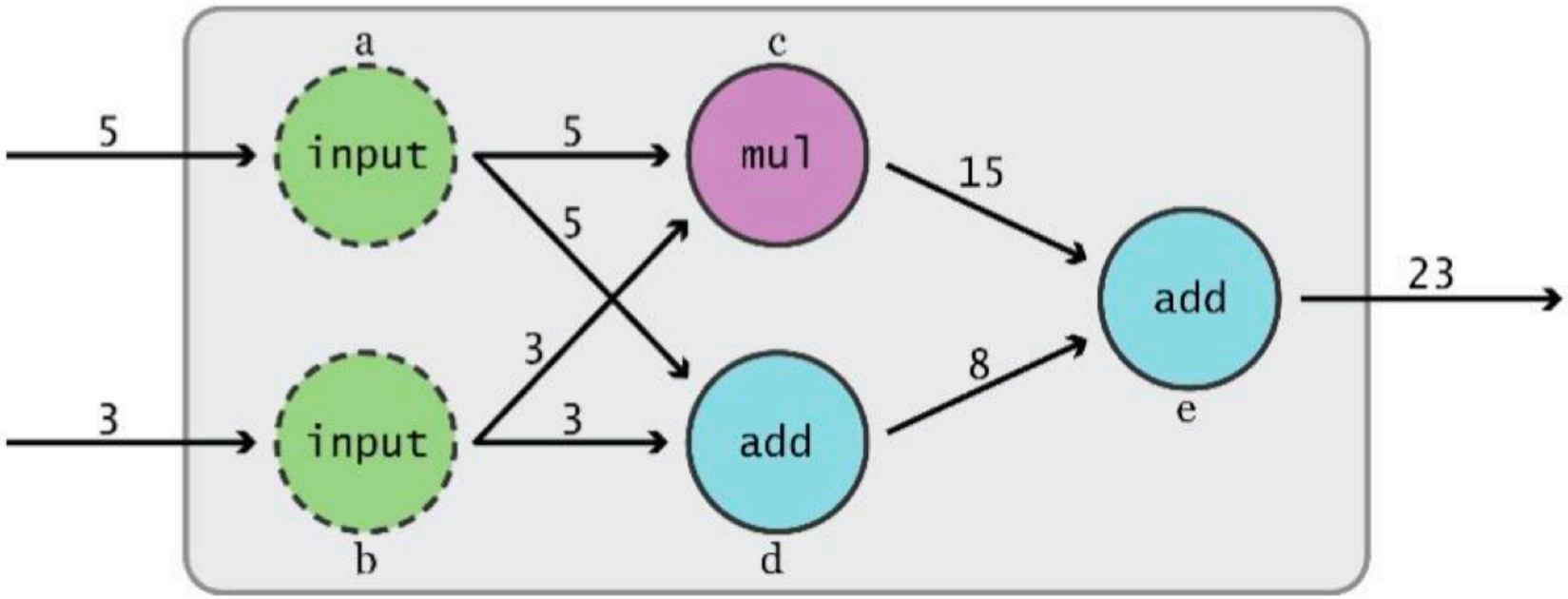
TensorFlow

Another framework for computing things.

Keep models like Spark, Flink, Hadoop, etc, in mind.

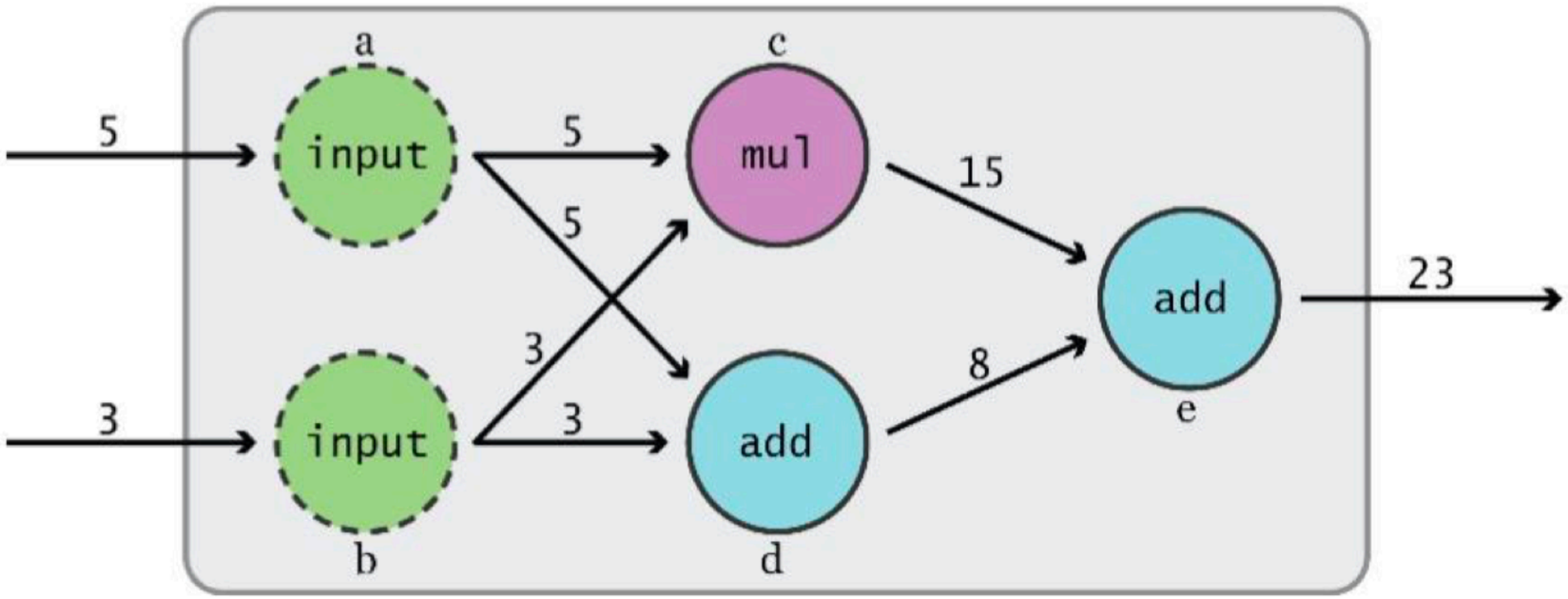
And let's take a glimpse of what TensorFlow looks like
before we go into what it's most often used for.

Basic Concepts



TensorFlow separates definition of computations from their execution

Basic Concepts



TensorFlow separates definition of computations from their execution

Phase 1: assemble a graph
Phase 2: use a session to execute operations in the graph.

Basic Concepts (1): Assembling the Graph

Ok, what's the graph?

Basic Concepts (1): Assembling the Graph

Ok, what's the graph?

TensorFlow graphs are made up of two parts.

1. **Edges**.
2. **Nodes**.

Basic Concepts (1): Assembling the Graph

Ok, what's the graph?

TensorFlow graphs are made up of two parts.

1. **Edges**.→ Tensors.
2. **Nodes**. → Operators, variables, constants.

Basic Concepts (1): Assembling the Graph

Ok, what's the graph?

TensorFlow graphs are made up of two parts.

1. **Edges**.→ Tensors. *Or, n-dimensional array.*
    - *0-d tensor: scalar (number)*
    - *1-d tensor: vector*
    - *2-d tensor: matrix*

2. **Nodes**. → Operators, variables, constants.

Basic Concepts (1): Assembling the Graph

Ok, what's the graph?

TensorFlow graphs are made up of two parts.

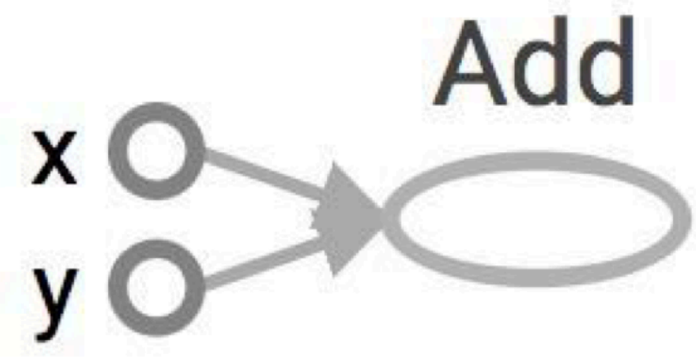
1. **Edges.** → Tensors. *Or, n-dimensional array.*
   - *0-d tensor: scalar (number)*
   - *1-d tensor: vector*
   - *2-d tensor: matrix*

2. **Nodes.** → Operators, variables, constants.

Tensors are data.
TensorFlow = tensor + flow = data + flow

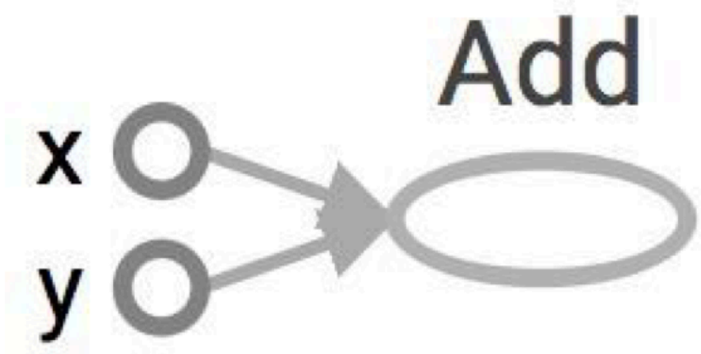
Basic Concepts: A Simple Graph

```
import tensorflow as tf
a = tf.add(3, 5)
```



Basic Concepts: A Simple Graph

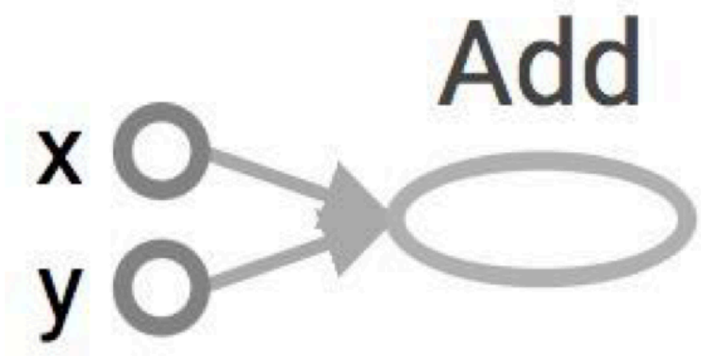
```
import tensorflow as tf
a = tf.add(3, 5)
```



Where did the names x and y come from?

Basic Concepts: A Simple Graph

```
import tensorflow as tf
a = tf.add(3, 5)
```



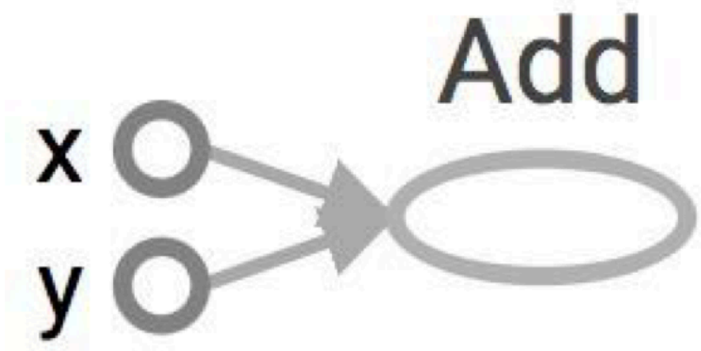
Where did the names x and y come from?

TensorFlow automatically names the nodes when you don't explicitly name them.

E.g.,
x = 3
y = 5

Basic Concepts: A Simple Graph

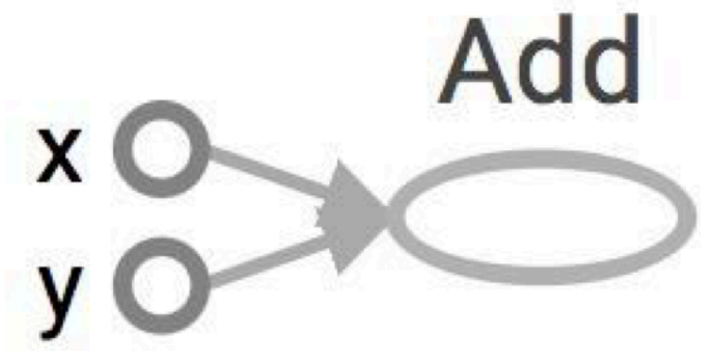
```
import tensorflow as tf
a = tf.add(3, 5)
```



So this is just a visualization of the computation that we want to do, right?

Basic Concepts: A Simple Graph

```
import tensorflow as tf
a = tf.add(3, 5)
```



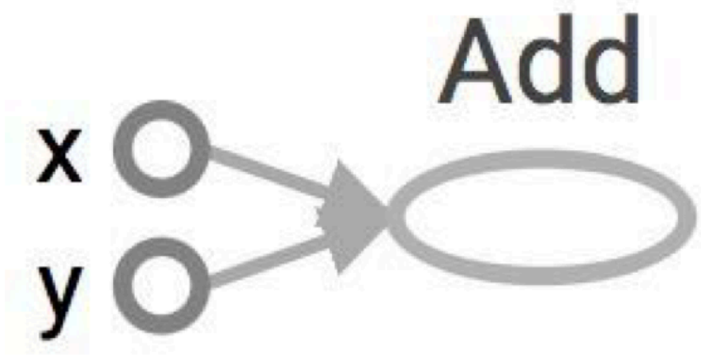
So this is just a visualization of the computation that we want to do, right?

No! this is actually part of our computation.

Tensors are data.

Basic Concepts: A Simple Graph

```
import tensorflow as tf
a = tf.add(3, 5)
```



So this is just a visualization of the computation that we want to do, right?

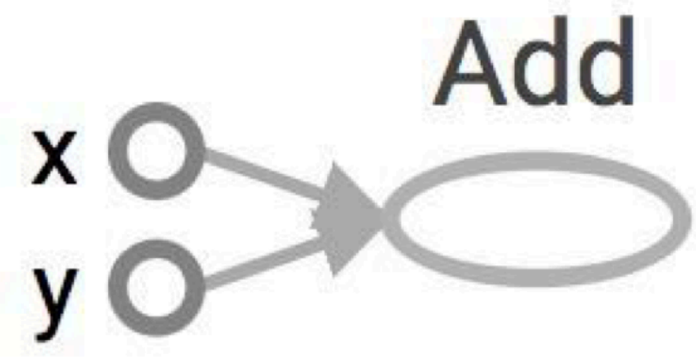
No! this is actually part of our computation.

Tensors are data.

TensorFlow = tensor + flow = data + flow

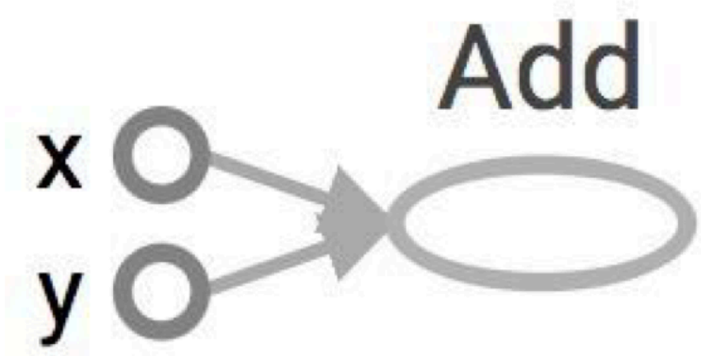
Basic Concepts: A Simple Graph

```
import tensorflow as tf
a = tf.add(3, 5)
print(a) # What does this do?
```



Basic Concepts: A Simple Graph

```
import tensorflow as tf
a = tf.add(3, 5)
print(a) # What does this do?
```



Result:

```
>> Tensor("Add:0", shape=(), dtype=int32)
```

The answer is not 8!

Basic Concepts: Sessions

A Session encapsulates the environment in which Operation objects are executed and Tensor objects are evaluated.

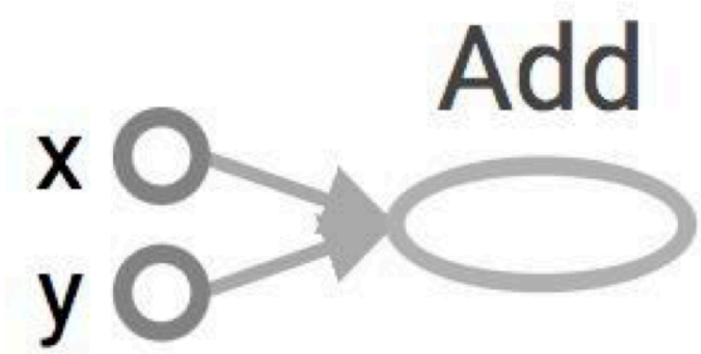
A Session will also do stuff like allocating memory to store the current values of variables.

Think of a Session as the thing which:

- *bundles up environment where your computations are run.*
- *is the handle to trigger the execution of your staged computations!*

How do you get the value of a?

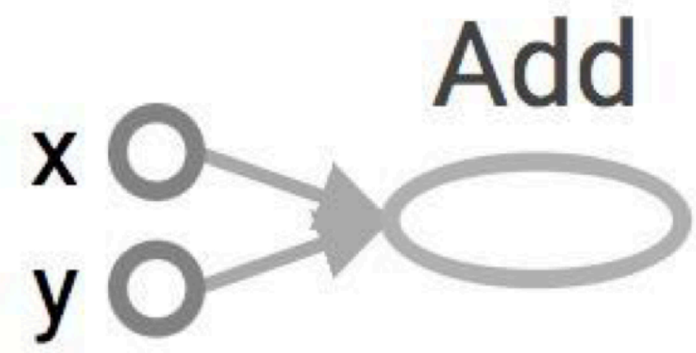
```
import tensorflow as tf
a = tf.add(3, 5)
print(a)
```



To actually compute a, we need to create a Session, assign it to a variable, and indicate that we would like it to run a.

Basic Concepts (2): Running the Graph

```
import tensorflow as tf
a = tf.add(3, 5)
with tf.Session() as sess:
      print(sess.run(a))
```

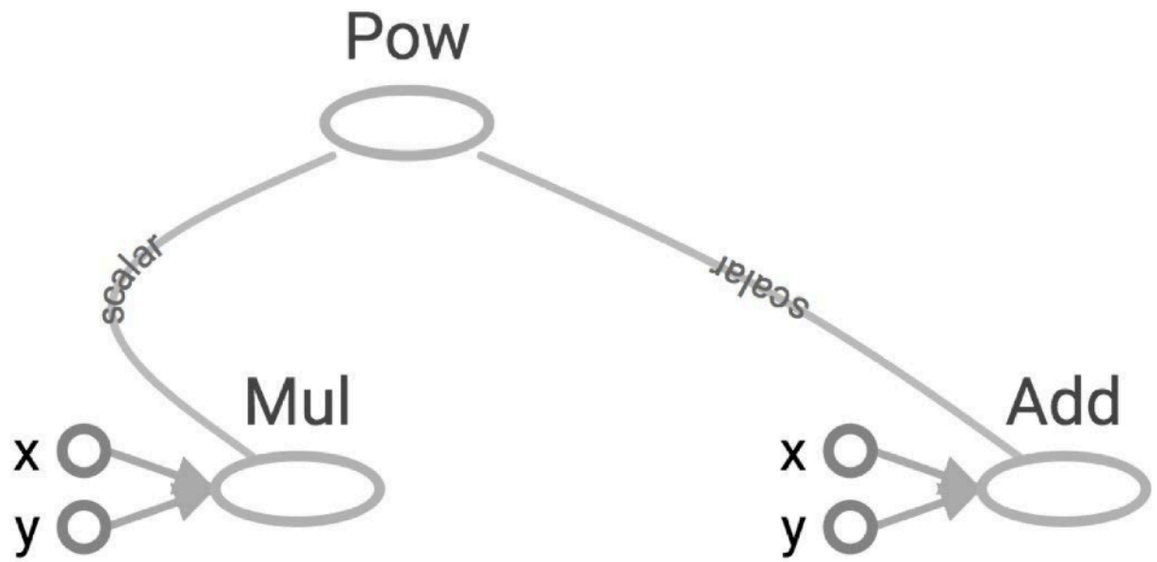


Result:

>> 8

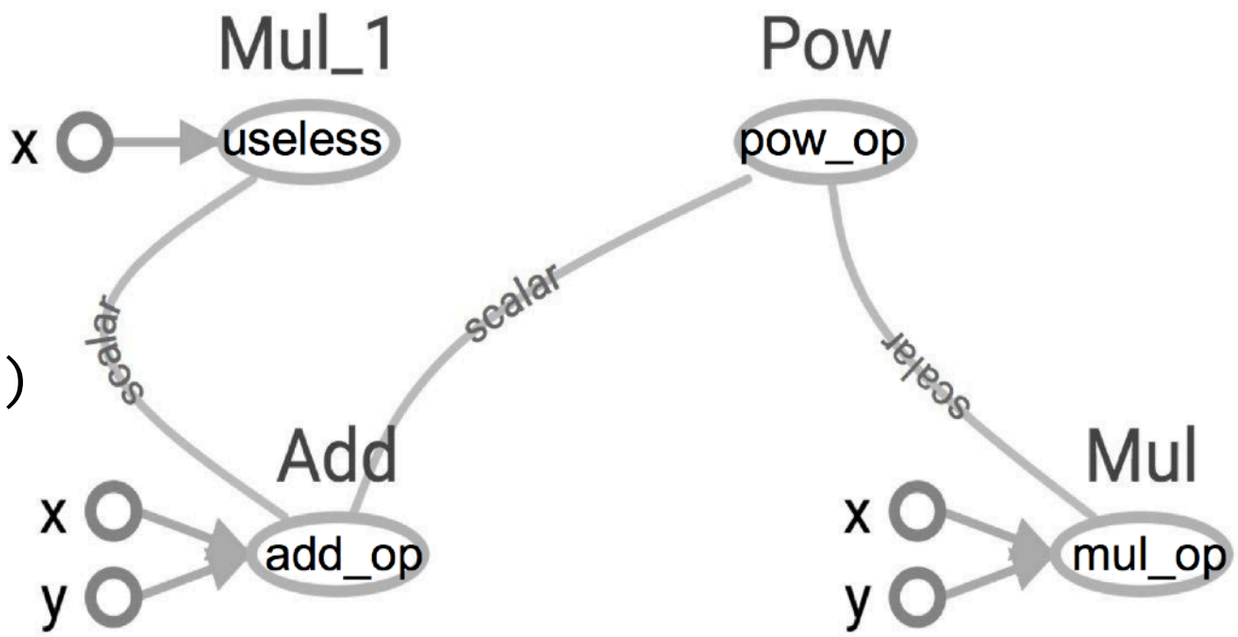
Another Graph

```
x=2
y=3
op1 = tf.add(x, y)
op2 = tf.multiply(x, y)
op3 = tf.pow(op2, op1)
with tf.Session() as sess:
      op3 = sess.run(op3)
```



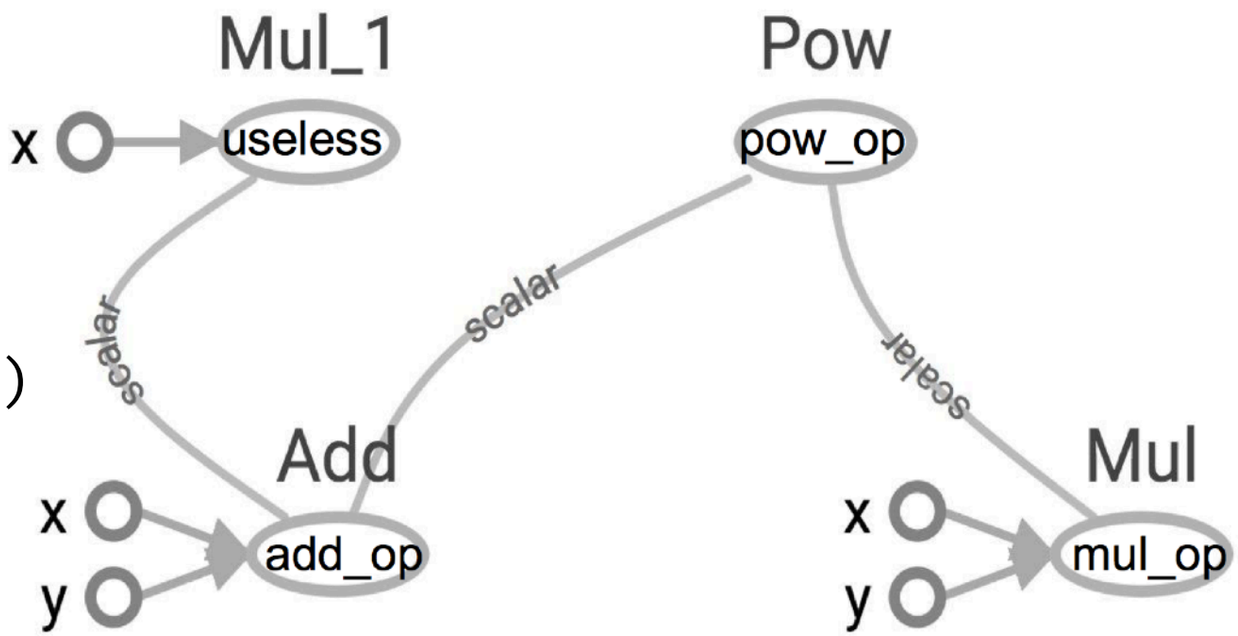
Another Another Graph

```
x=2
y=3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
      z = sess.run(pow_op)
```



Another Another Graph

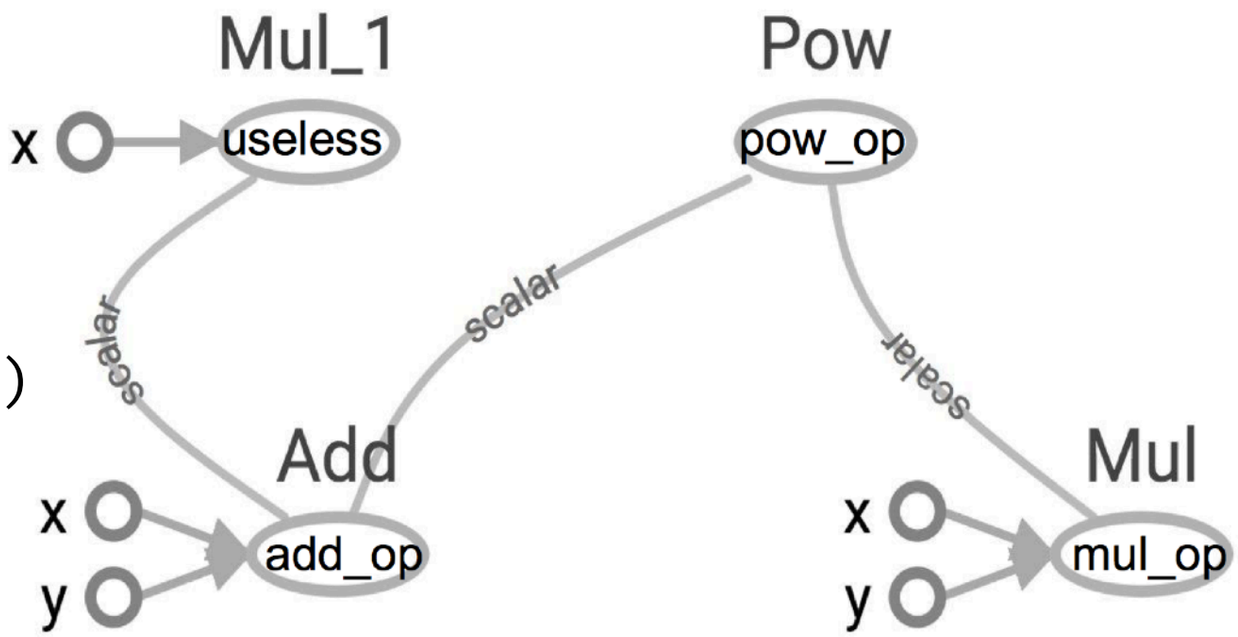
```
x=2
y=3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
      z = sess.run(pow_op)
```



What happens here?

Another Another Graph

```
x=2
y=3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
      z = sess.run(pow_op)
```



What happens here?

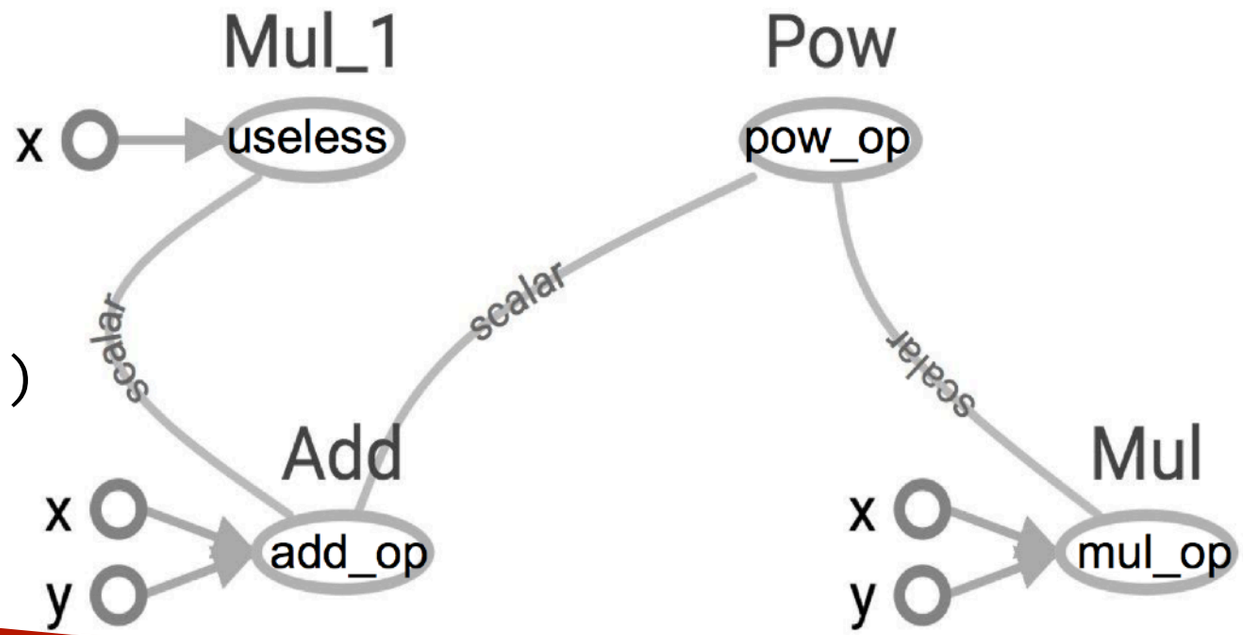
We only want the value of `pow_op`**.**

Since pow-op doesn't depend on `useless`**, we don't compute the value of** `useless`**!**

We can skip computations we don't absolutely need!

Another Another Graph

```
x=2
y=3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
      z = sess.run(pow_op)
```



What happen

Computations broken into subgraphs.

The `useless` *subgraph* is skipped!

We only want the value of `pow_op`.

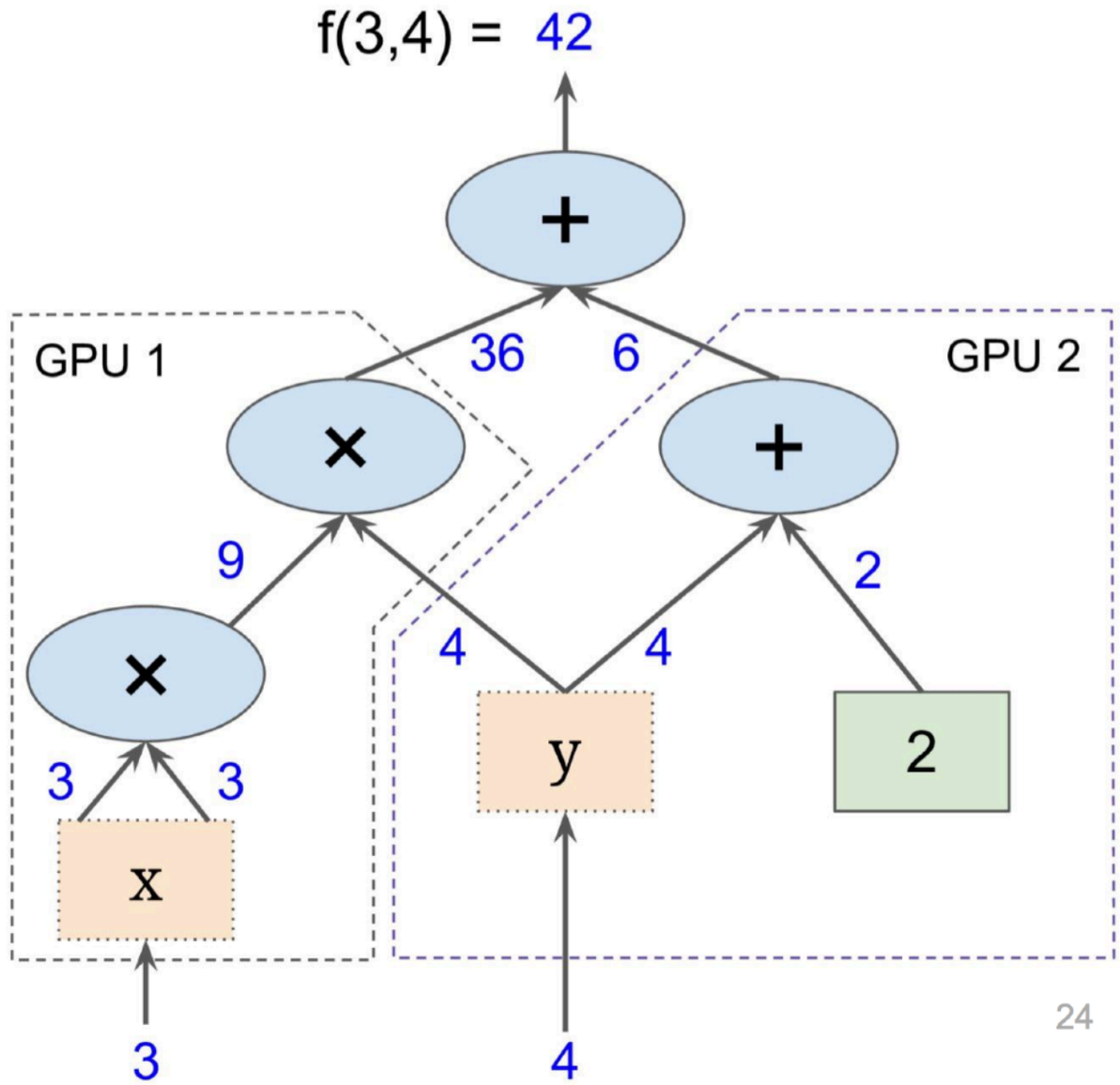
Since pow-op doesn't depend on `useless`, we don't compute the value of `useless`!

We can skip computations we don't absolutely need!

Subgraphs Can Be Distributed

It's possible to break graphs into several ***subgraphs*** and run them in parallel across:

- CPUs
- GPUs
- TPUs
- or other devices

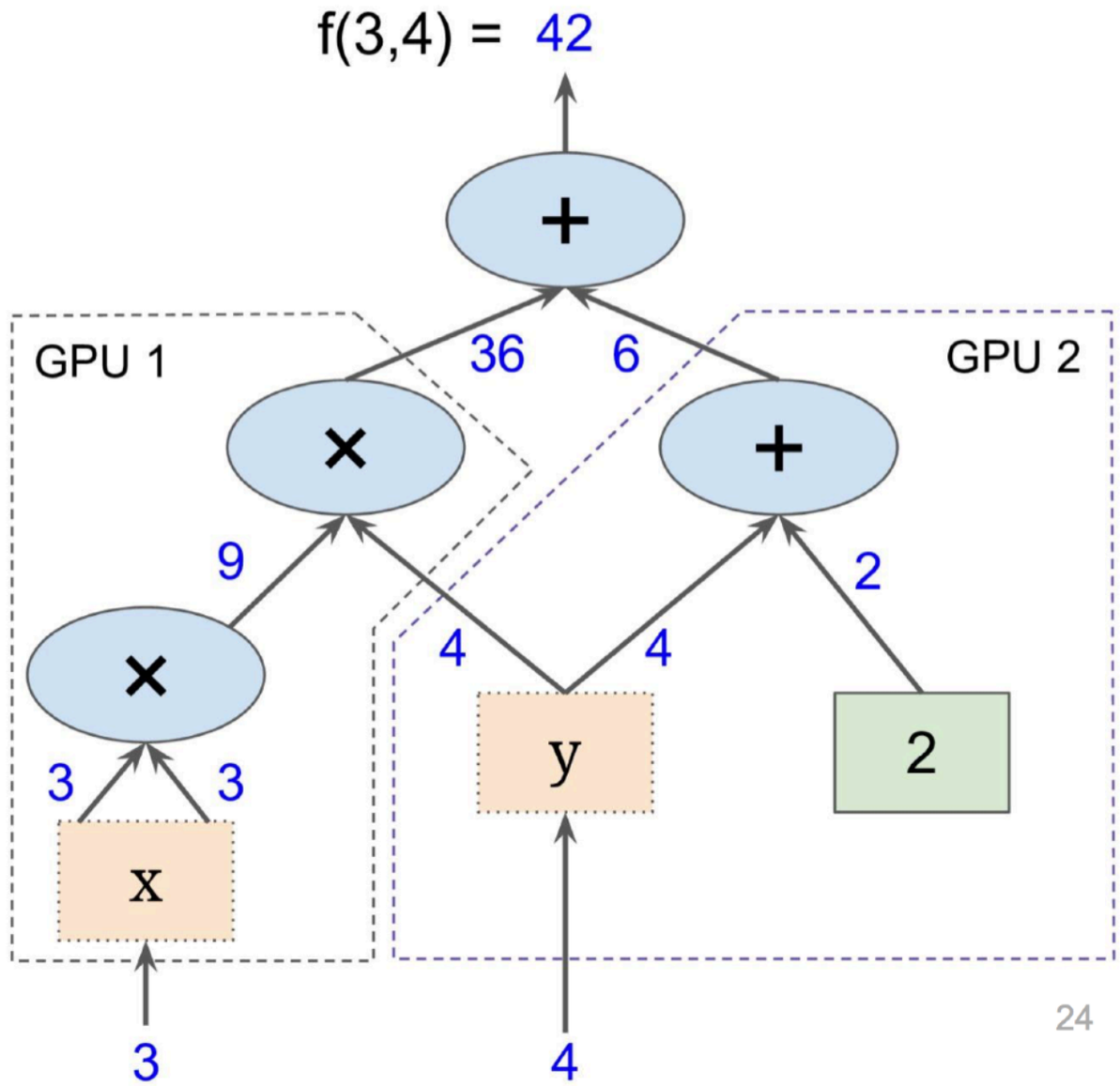


Subgraphs Can Be Distributed

It's possible to break graphs into several ***subgraphs*** and run them in parallel across:

- CPUs
- GPUs
- TPUs
- or other devices

A Tensor Processing Unit (TPU) is a custom ASIC tailored to TensorFlow.

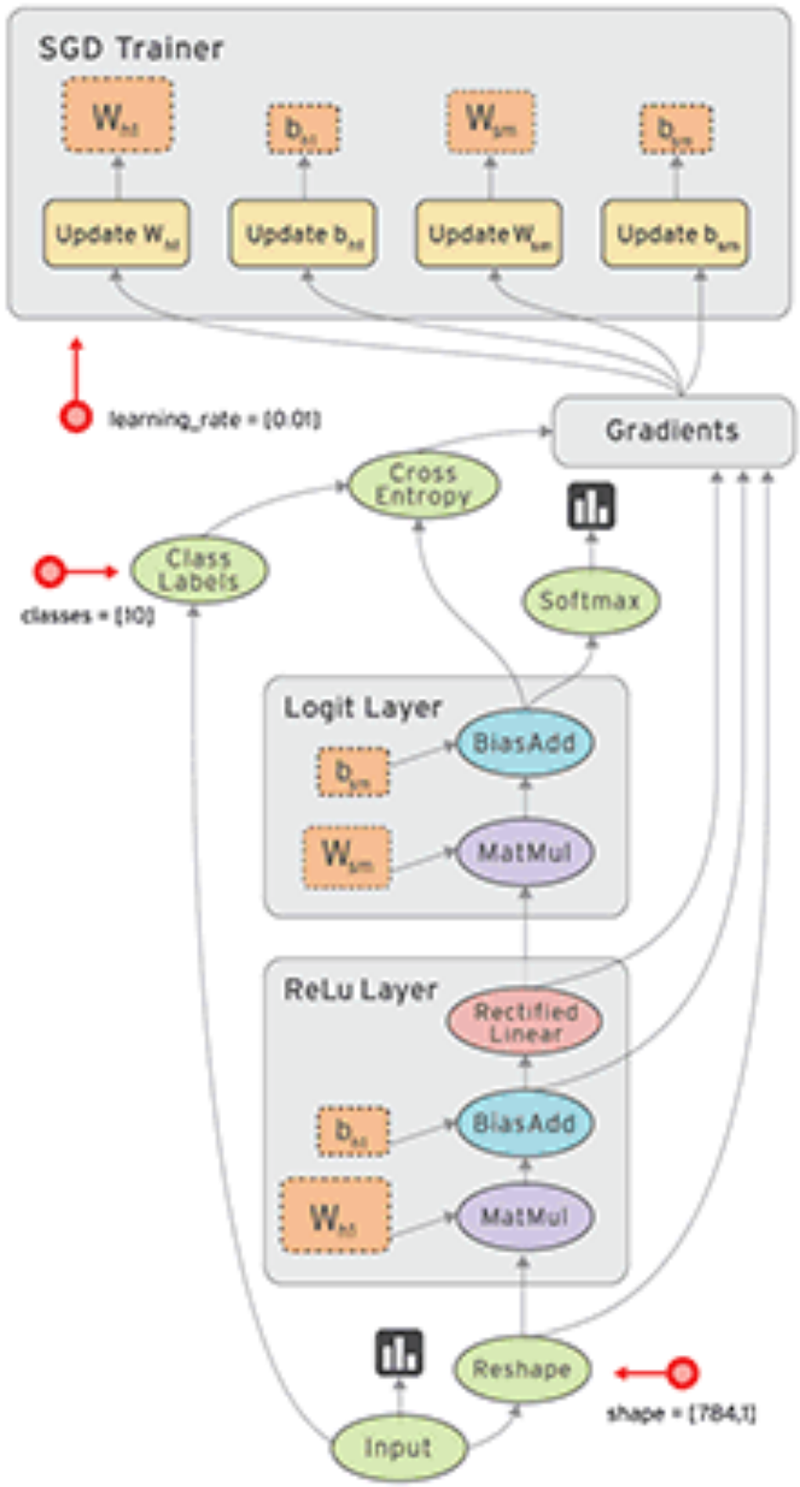


> *"More tolerant of reduced computational precision, which means it requires fewer transistors per operation. Because of this, we can squeeze more operations per second into the silicon, use more sophisticated and powerful machine learning models and apply these models more quickly, so users get more intelligent results more rapidly."*

Why Make Everything a Graph?

1. **We can save computation**. Only run subgraphs that lead to the values you actually want to fetch.
2. **Break up computation** into small, differential pieces to facilitate auto-differentiation.
3. **Facilitate distributed computation**. Spread work across multiple CPUs, GPUs, TPUs, or other devices.
4. **Many ML models are already graphs**. Most common ML algorithms are already taught and visualized as directed graphs.
5. **Compilation/performance**. TensorFlow's XLA Compiler can generate faster code, for example, by fusing together adjacent operations.

Why Make Everything a Graph?



What makes TensorFlow different from other models of computation?

Such as Spark, Flink, Hadoop, etc.?

It can do most simple data-flow computations that we've seen in the MapReduce programming model.

On top of that, it comes with extra support for deep learning/ machine learning tasks.

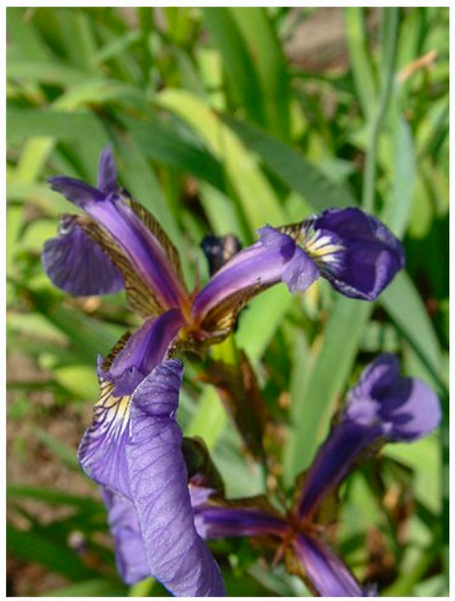
Shape of a Learning Problem: Iris Classification

Goal: classify each Iris flower you find.

We will classify Iris flowers based only on the *length and width of their sepals and petals*.

We'll focus on the following 3 species:

Iris setosa, Iris virginica, Iris versicolor



Shape of a Learning Problem: Iris Classification

Starting point: We have a data set of 120 Iris flowers with their sepal and petal measurements.

The first 5 elements:

Sepal length	sepal width	petal length	petal width	species
6.4	2.8	5.6	2.2	2
5.0	2.3	3.3	1.0	1
4.9	2.5	4.5	1.7	2
4.9	3.1	1.5	0.1	0
5.7	3.8	1.7	0.3	0

Shape of a Learning Problem: Iris Classification

Given a problem and some data, the shape of the problem we must solve looks like:

1. *Import and parse the data sets*
2. *Create feature columns to describe the data*
3. *Select the type of model (or create one yourself)*
4. *Train the model*
5. *Evaluate the model's effectiveness*
6. *Let the trained model make its own predictions*

Shape of a Learning Problem: Iris Classification

1. Import and parse the data sets

```
TRAIN_URL = "http://download.tensorflow.org/data/iris_training.csv"
TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"

CSV_COLUMN_NAMES = ['SepalLength', 'SepalWidth',
                    'PetalLength', 'PetalWidth', 'Species']

def load_data(label_name='Species'):
    """Parses the csv file in TRAIN_URL and TEST_URL."""
    # ...
    return (train_features, train_label), (test_features, test_label)
```

Shape of a Learning Problem: Iris Classification

2. Create feature columns to describe the data

```
# VERSION 1: Create feature columns for all features.
my_feature_columns = []
for key in train_x.keys():
     my_feature_columns.append(tf.feature_column.numeric_column(key=key))
```

```
# VERSION 2: Create feature columns for all features.
my_feature_columns = [
    tf.feature_column.numeric_column(key='SepalLength'),
    tf.feature_column.numeric_column(key='SepalWidth'),
    tf.feature_column.numeric_column(key='PetalLength'),
    tf.feature_column.numeric_column(key='PetalWidth')
]
```

A **feature column** is a data structure that tells your model how to interpret the data in each feature.

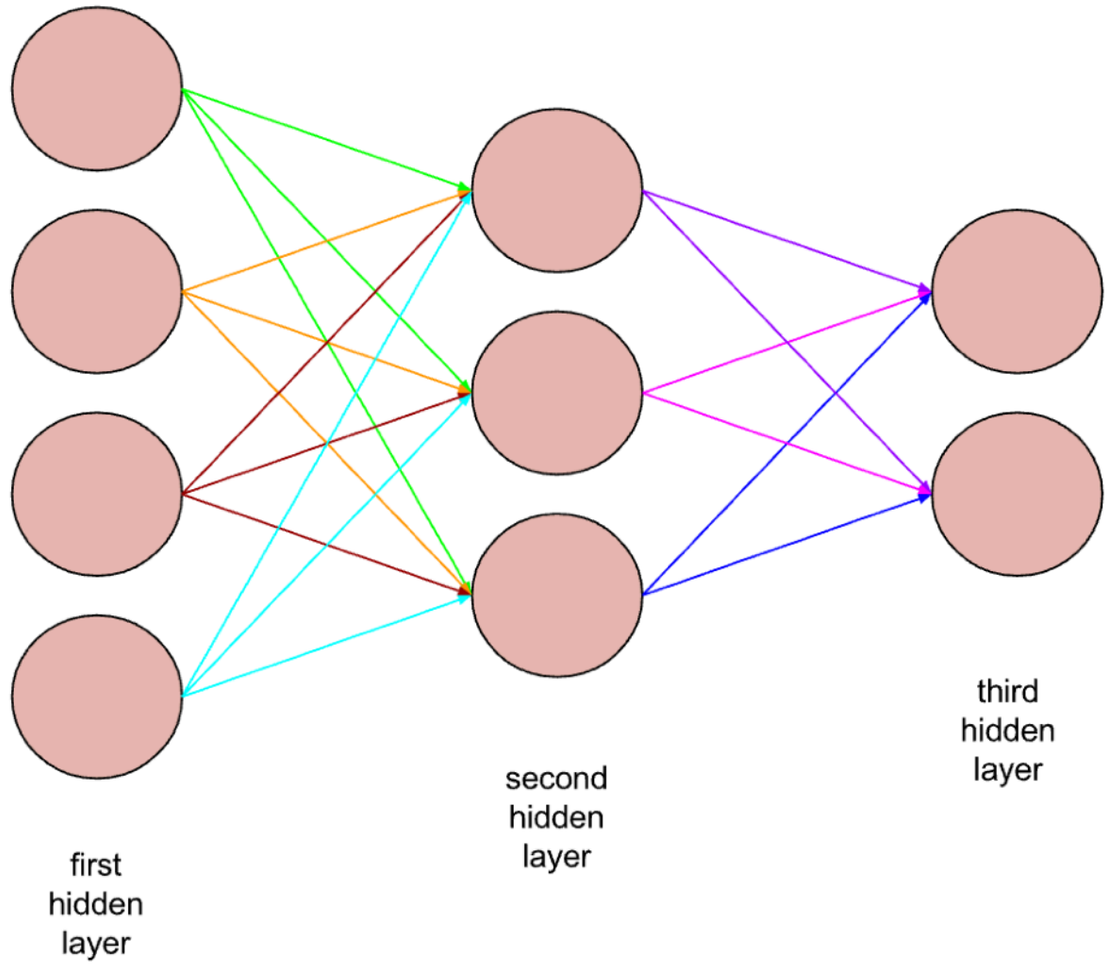
In this case, we're interpreting the features quite literally, but in many other cases, feature selection is less obvious.

Shape of a Learning Problem: Iris Classification

3. Select a model to use to solve your problem

In this case, we'll choose a ***neural network***.

Neural networks can find complex relationships between features and labels. It's normally a very structured graph organized into one or more *hidden layers*.



Shape of a Learning Problem: Iris Classification

3. **Select a model to use to solve your problem**

   To specify a model type, first create an instance of `Estimator`.

   We can use a pre-made Estimator, provided by TensorFlow called `tf.estimator.DNNClassifier`. This Estimator builds a neural network that classifies examples.

```
classifier = tf.estimator.DNNClassifier(
    feature_columns=my_feature_columns,
    hidden_units=[10, 10],
    n_classes=3)
```

The length of the list assigned to `hidden_units` identifies the number of hidden layers (2, in this case).

Shape of a Learning Problem: Iris Classification

4. Train the model

Instantiating a `tf.Estimator.DNNClassifier` creates a framework for learning the model. Basically, we've wired a network but haven't yet let data flow through it.

To train the neural network, call the `Estimator` object's train method.

```
classifier.train(
    input_fn=lambda:train_input_fn(train_feature, train_label, args.batch_size),
    steps=args.train_steps)
```

Shape of a Learning Problem: Iris Classification

4. Train the model

```
classifier.train(
    input_fn=lambda:train_input_fn(train_feature, train_label, args.batch_size),
    steps=args.train_steps)
```

`train` takes:

- `train_feature`: a Python dictionary in which (1) each key is the name of a feature, (2) each value is an array containing the value for each example in the training set.
- `train_label`: an array containing the values of the label of every example in the data set.
- `args.batch_size`: an integer representing the batch size, or the number of examples used in one iteration (that is, one gradient update) of model training.

Shape of a Learning Problem: Iris Classification

5. **Evaluate the model's effectiveness**

   Evaluating means determining how effectively the model makes predictions.

   To determine the Iris classification model's effectiveness, pass some sepal and petal measurements to the model and ask the model to predict what Iris species they represent. Then compare the model's prediction against the actual label.

Test Set					
Features				Label	Prediction
5.9	3.0	4.3	1.5	1	1
6.9	3.1	5.4	2.1	2	2
5.1	3.3	1.7	0.5	0	0
6.0	3.4	4.5	1.6	1	2
5.5	2.5	4.0	1.3	1	1

A model that is 80% accurate.

Shape of a Learning Problem: Iris Classification

5. **Evaluate the model's effectiveness**

   `Estimators` provide an estimate method to do this for you.

```
# Evaluate the model.
eval_result = classifier.evaluate(
    input_fn=lambda:eval_input_fn(test_x, test_y, args.batch_size))
print('\nTest set accuracy: {accuracy:0.3f}\n'.format(**eval_result))

# Output:
Test set accuracy: 0.967
```

The call to `classifier.evaluate` is similar to the call to `classifier.train`.

The biggest difference is that `classifier.evaluate` must get its examples from the test set rather than the training set. In other words, to fairly assess a model's effectiveness, the examples used to *evaluate* a model must be different from the examples used to *train* the model.

Shape of a Learning Problem: Iris Classification

6. **Let the trained model make its own predictions**

   Now we have a trained model we'd like to actually use to make predictions on unlabeled examples.

   Assuming we have the following three unlabeled examples:

```
predict_x = {
    'SepalLength': [5.1, 5.9, 6.9],
    'SepalWidth': [3.3, 3.0, 3.1],
    'PetalLength': [1.7, 4.2, 5.4],
    'PetalWidth': [0.5, 1.5, 2.1],
}
```

Shape of a Learning Problem: Iris Classification

6. **Let the trained model make its own predictions**

   We can use the `predict` method on `Estimator`.

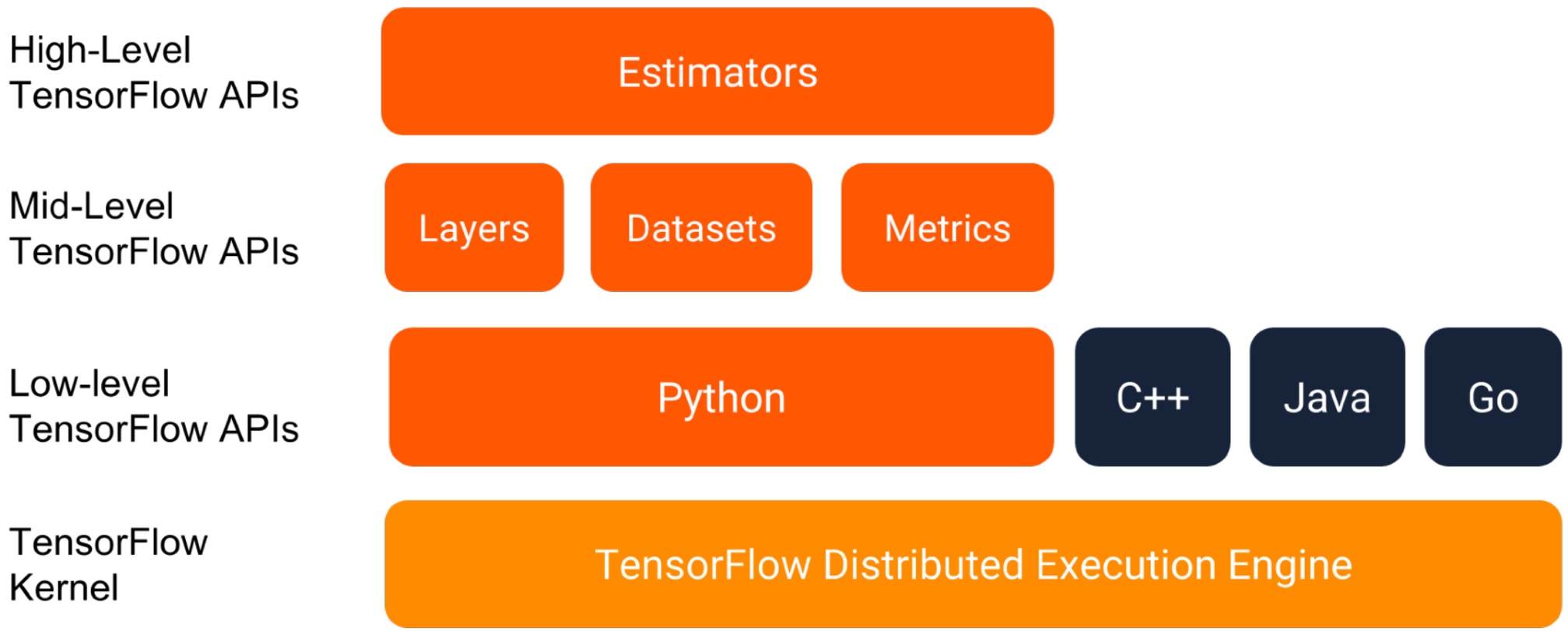
```
predictions = classifier.predict(
    input_fn=lambda:eval_input_fn(predict_x,
                                  labels=None,
                                  batch_size=args.batch_size))
```

The `predict` method returns a python iterable, yielding a dictionary of prediction results for each example.

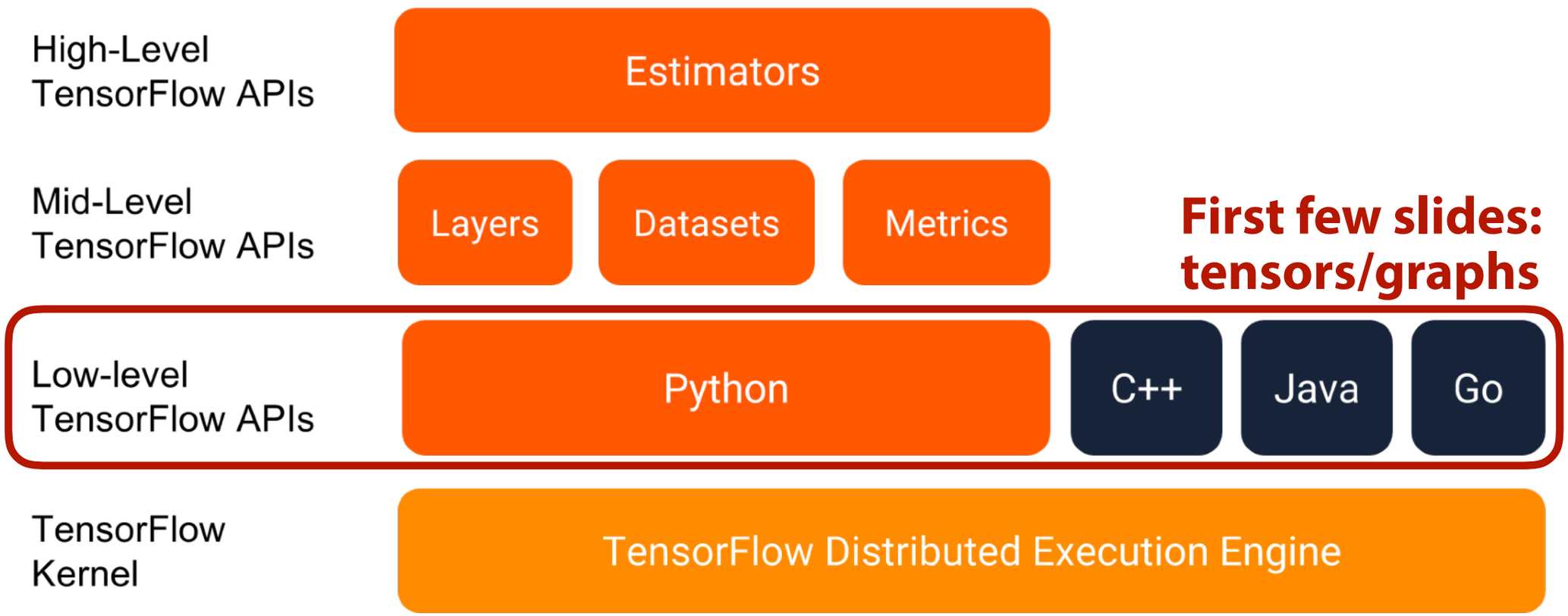
```
'probabilities': array([  1.19127117e-08,   3.97069454e-02,   9.60292995e-01])

        Prediction is "Setosa" (99.6%), expected "Setosa"
        Prediction is "Versicolor" (99.8%), expected "Versicolor"
        Prediction is "Virginica" (97.9%), expected "Virginica"
```

TensorFlow: High-Level Picture

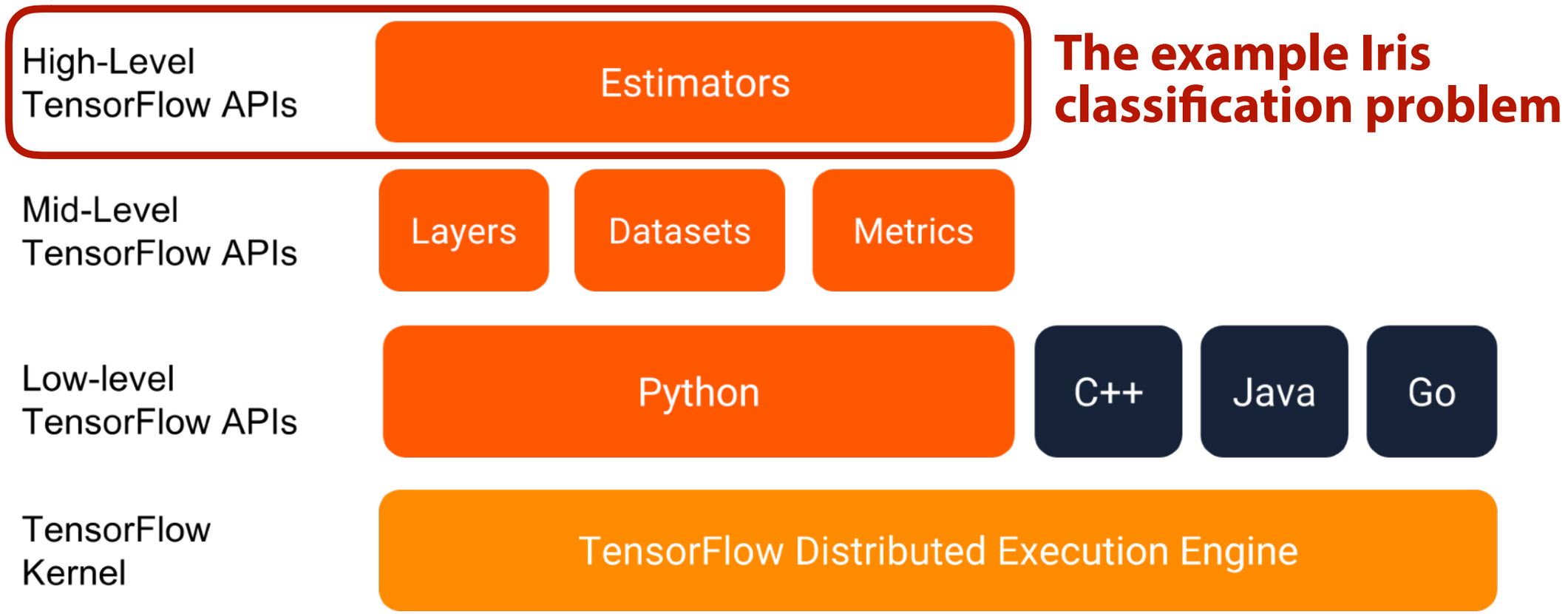


TensorFlow: High-Level Picture



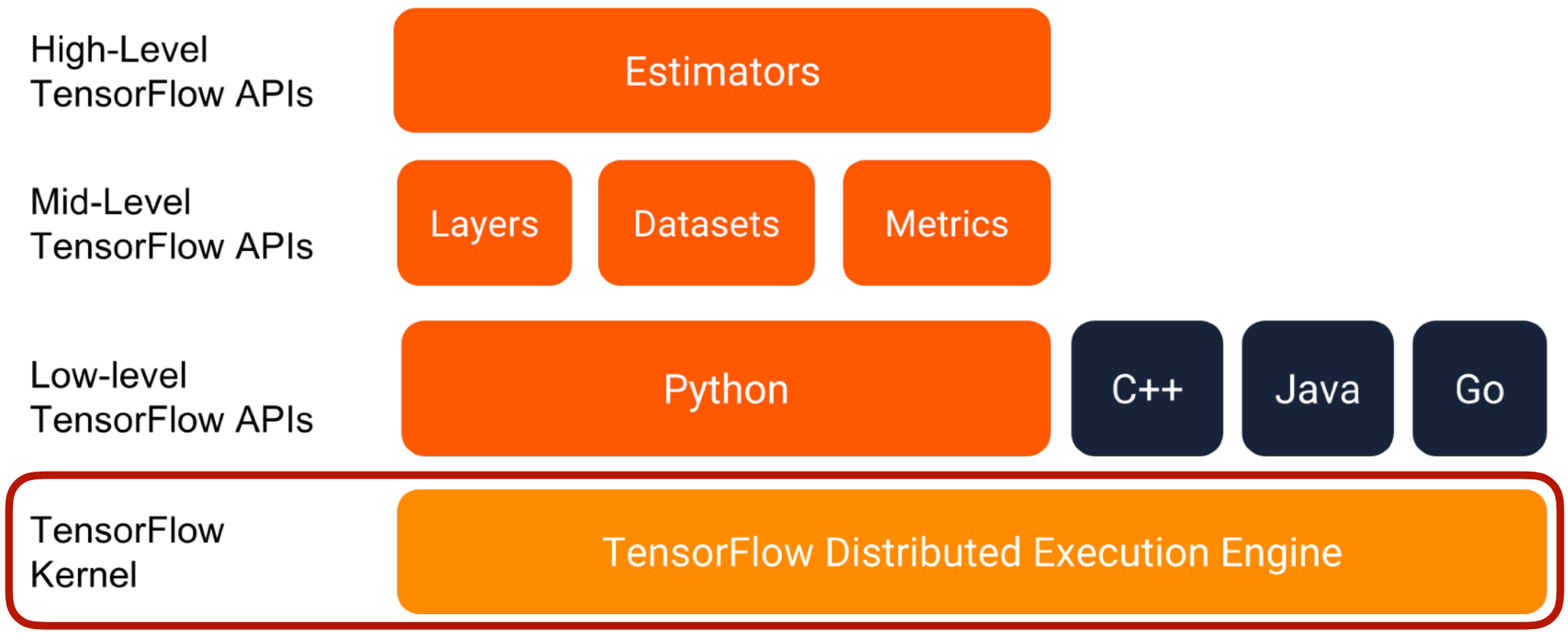
First few slides: tensors/graphs

TensorFlow: High-Level Picture



The example Iris classification problem

TensorFlow: High-Level Picture



The kernel operations are particular implementations of operations that can be run on a particular type of device (e.g., CPU, GPU)

This is a major strength of TensorFlow! It means it can run on heterogeneous hardware!

TensorFlow: High-Level Picture

> *"TensorFlow takes computations described using a dataflow-like model and maps them onto a wide variety of different hardware platforms."*
>
> *"…ranging from running inference on mobile device platforms like Android and iOS to modest-sized training and inference systems using single machines containing one or many GPU cards to large-scale training systems running on hundreds of specialized machines with thousands of GPUs."*
>
> *"Having a single system that can span such a broad range of platforms significantly simplifies the real-world use of a machine learning system. (As we have found that having separate systems for large-scale training and small-scale deployment leads to significant maintenance burdens and leaky abstractions."*

- From Google's 2015 TensorFlow whitepaper

TensorFlow: High-Level Picture

At Google, used in:

- Google Search
- advertising products
- speech recognition systems
- Google Photos
- Google Maps
- StreetView
- Google Translate
- YouTube
- *and many others…*

Used at Google as a scalable distributed training and inference system by over 50 teams!

TensorFlow: High-Level Picture

At Google, used in:

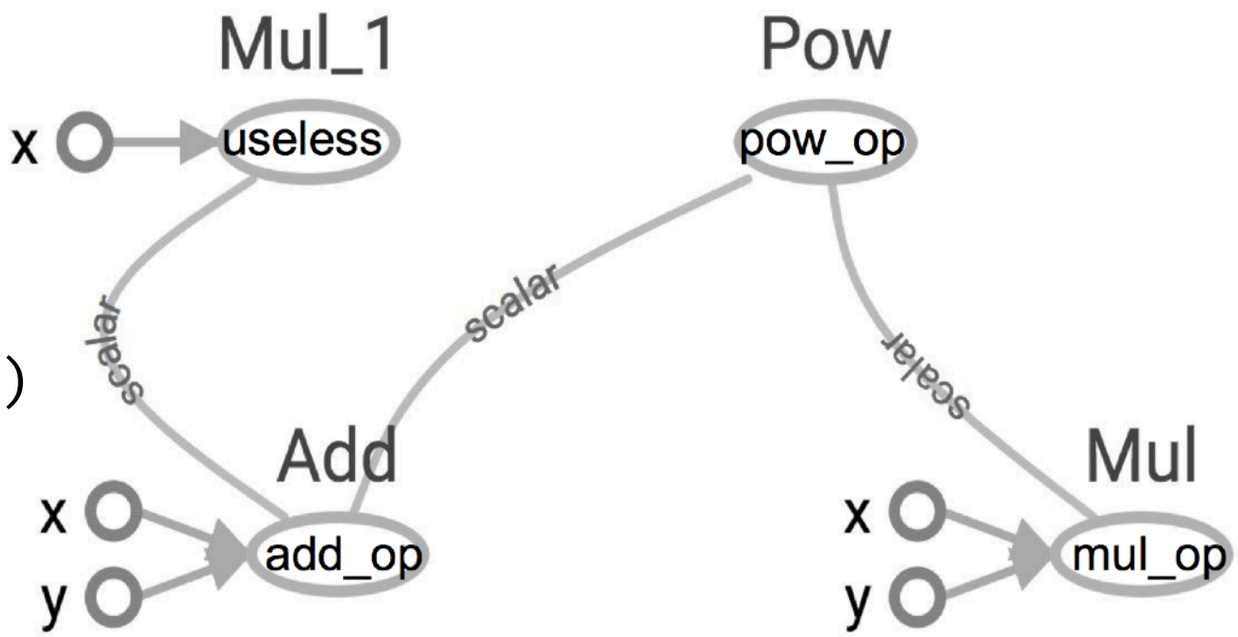
- Google Search
- advertising products
- speech recognition systems
- Google Photos
- Google Maps
- StreetView
- Google Translate
- YouTube
- *and many others…*

Used at Google as a scalable distributed training and inference system by over 50 teams!

Big idea:
TF programming model can be run distributed, single-node, or on various different hardware.

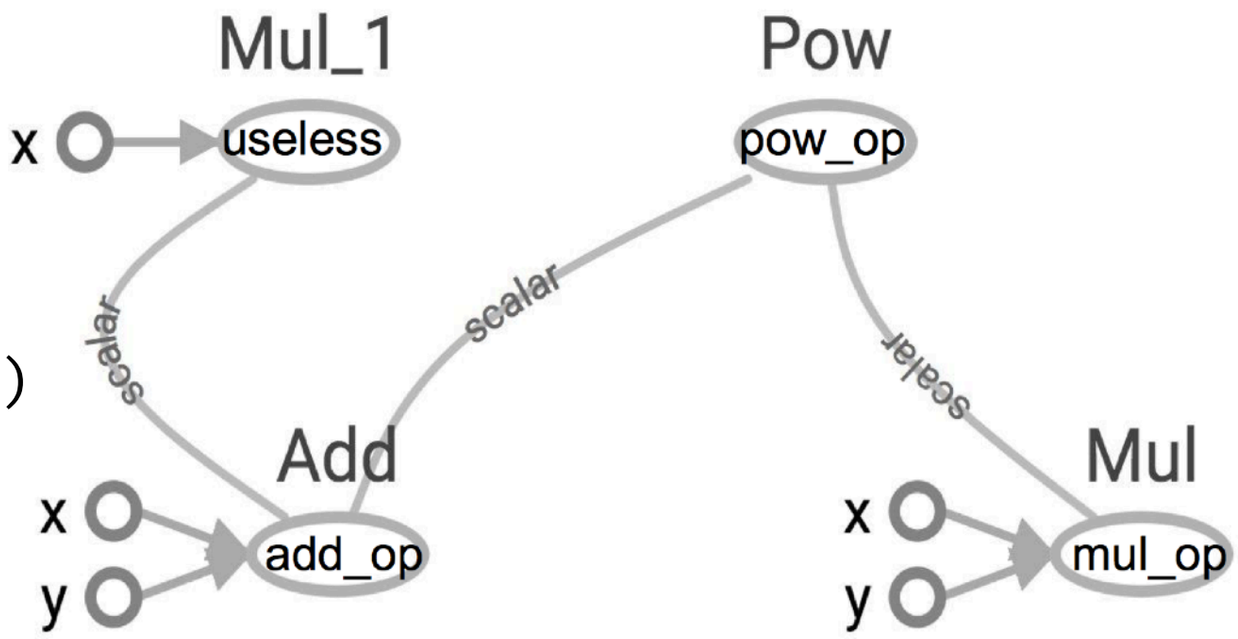
Back to Our Graphs

```
x=2
y=3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
      z = sess.run(pow_op)
```



Back to Our Graphs

```
x=2
y=3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
      z = sess.run(pow_op)
```



As it stands, this graph is not especially interesting because it always produces a constant result.

A graph can be parameterized to accept external inputs, known as **placeholders**.

A placeholder is a promise to provide a value later, like a future, or promise in other languages.

Parameterized Graphs: Placeholders

A graph can be parameterized to accept external inputs, known as **placeholders**.

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = x + y
```

This can be thought of as a function in which we define two input parameters (x and y) and then an operation on them.

Parameterized Graphs: Placeholders

A graph can be parameterized to accept external inputs, known as **placeholders**.

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = x + y
```

This can be thought of as a function in which we define two input parameters (x and y) and then an operation on them.

We can evaluate this graph with multiple inputs by using the `feed_dict` argument of the run method to feed concrete values to the placeholders.

```
print(sess.run(z, feed_dict={x: 3, y: 4.5}))
print(sess.run(z, feed_dict={x: [1, 3], y: [2, 4]}))

# Output:
7.5
[ 3.  7.]
```

Parameterized Graphs: Placeholders

A graph can be parameterized to accept external inputs, known as **placeholders**.

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = x + y
```

This can
paramet
t

Placeholders are a kind of Tensor!

We can
ed_dict
argument of the run method to feed concrete values to the placeholders.

```
print(sess.run(z, feed_dict={x: 3, y: 4.5}))
print(sess.run(z, feed_dict={x: [1, 3], y: [2, 4]}))

# Output:
7.5
[ 3.  7.]
```

Parameterized Models: Datasets

Placeholders work for simple experiments, but **Datasets** are the preferred method of streaming data into a model.

Parameterized Models: Datasets

Placeholders work for simple experiments, but **Datasets** are the preferred method of streaming data into a model.

But Datasets by themselves aren't Tensors. To get a runnable `tf.Tensor` from a Dataset you must first convert it to a `tf.data.Iterator`, and then call the Iterator's `get_next` method.

Given,

```
my_data = [[0, 1,],[2, 3,],[4, 5,],[6, 7,],]
```

You can create an Iterator using the the `make_one_shot_iterator` method:

```
slices = tf.data.Dataset.from_tensor_slices(my_data)
next_item = slices.make_one_shot_iterator().get_next()
```

Parameterized Models: Datasets

Placeholders work for simple experiments, but **Datasets** are the preferred method of streaming data into a model.

But Datasets by themselves aren't Tensors. To get a runnable `tf.Tensor` from a Dataset you must first convert it to a `tf.dat` `next` method.

Given,

Data obtained from a Dataset via an Iterator is another kind of Tensor!

You can create an Iterator using the the `make_one_shot_iterator` method:

```
slices = tf.data.Dataset.from_tensor_slices(my_data)
next_item = slices.make_one_shot_iterator().get_next()
```

But what about nodes/operations?

We have seen many different kinds of datatypes as tensors. But what about operations on that data?

1. **Edges**.→ Tensors. *Or, n-dimensional array.*
   - *0-d tensor: scalar (number)*
   - *1-d tensor: vector*
   - *2-d tensor: matrix*

2. **Nodes**. → Operators, variables, constants.

But what about nodes/operations?

TensorFlow makes scores of specialized operations available, given some kind of Tensor.

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ….
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle…
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, …
Stateful operations	Variable, Assign, AssignAdd, …
Neural-net building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, …
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease,…
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

What gets updated when training happens?

What gets updated when training happens?

Variables do.

A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.

What gets updated when training happens?

Variables do.

A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.

A `tf.Variable` represents a tensor whose value can be changed by running ops on it.

Unlike `tf.Tensor` objects, a `tf.Variable` exists outside the context of a `singlesession.run` call.

Internally, a `tf.Variable` stores a persistent tensor. Specific ops allow you to read and modify the values of this tensor. These modifications are visible across multiple `tf.Sessions`, so multiple workers can see the same values for a `tf.Variable`.

What gets updated when training happens?

Variables do.

A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.

A `tf.Variable` represents a tensor whose value can be changed by running ops on it.

Unlike `tf.Tensor` objects, a `tf.Variable` exists outside the context of a `singlesession.run` call.

Internally, a `tf.Variable` stores a persistent tensor. Specific ops allow you to read and modify the values of this tensor. These modifications are visible across multiple `tf.Sessions`, so multiple workers can see the same values for a `tf.Variable`.

What gets updated when training happens?

Variables do.

A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.

```
my_variable = tf.get_variable("my_variable", [1, 2, 3])
```

What gets updated when training happens?

Variables do.

A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.

```
my_variable = tf.get_variable("my_variable", [1, 2, 3])
```

Variables also must be initialized. One way to do that:

```
my_int_variable = tf.get_variable("my_int_variable", [1, 2, 3],
  dtype=tf.int32,initializer=tf.zeros_initializer)
```

What gets updated when training happens?

Variables do.

A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.

```
my_variable = tf.get_variable("my_variable", [1, 2, 3])
```

Variables also must be initialized. One way to do that:

```
my_int_variable = tf.get_variable("my_int_variable", [1, 2, 3],
  dtype=tf.int32,initializer=tf.zeros_initializer)
```

To use the value of a `tf.Variable` in a TensorFlow graph, simply treat it like a normal `tf.Tensor`:

```
v = tf.get_variable("v", shape=(), initializer=tf.zeros_initializer())
w = v + 1  # w is a tf.Tensor which is computed based on the value of v.
           # Any time a variable is used in an expression it gets automatically
           # converted to a tf.Tensor representing its value.
```

Distributed TensorFlow

Single-node to Distributed Execution

The main components in a TensorFlow system are:

- **the client**, which uses the Session interface to communicate with the *master*,
- **one or more *worker processes***, with each *worker process* responsible for arbitrating access to one or more computational devices (such as CPU cores or GPU cards) and for executing graph nodes on those devices as instructed by the master.

Single-node to Distributed Execution

The main components in a TensorFlow system are:

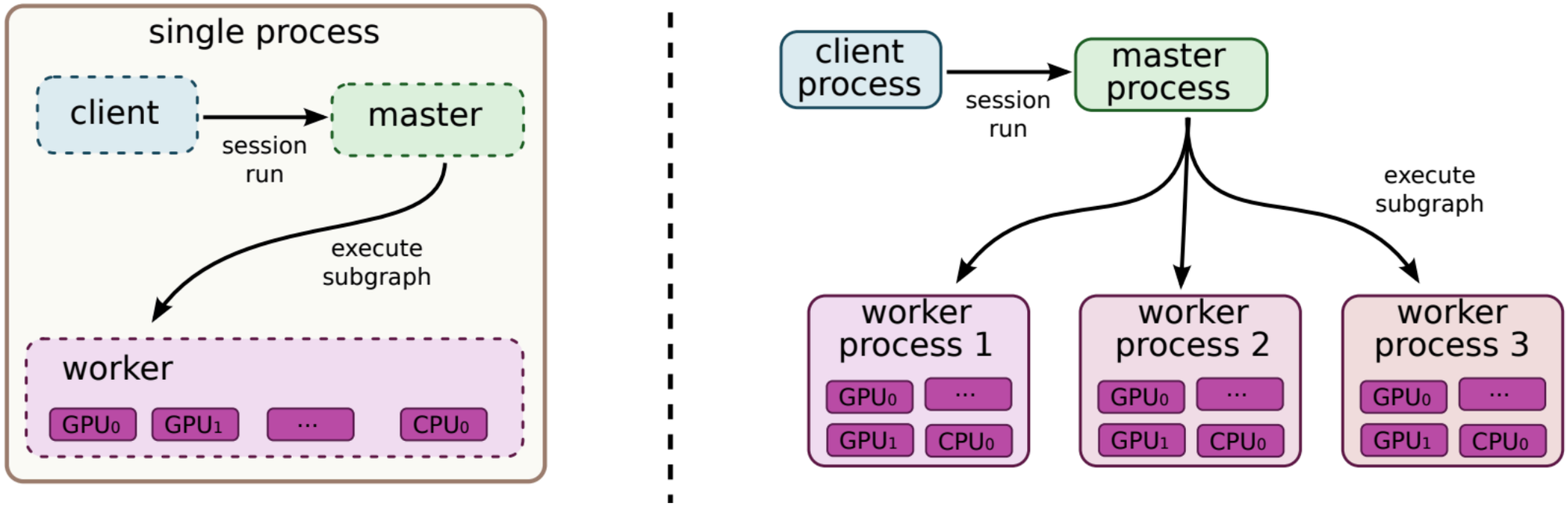
- **the client**, which uses the Session interface to communicate with the *master*,
- **one or more *worker processes***, with each *worker process* responsible for arbitrating access to one or more computational devices (such as CPU cores or GPU cards) and for executing graph nodes on those devices as instructed by the master.

Two implementations

- **local implementation** is used when the client, the master, and the worker all run on a single machine
- **distributed implementation** shares most of the code with the local implementation, but extends it with support for an environment where the client, master, and workers can all be in different processes on different machines.

Single-node to Distributed Execution

Single machine vs distributed system structure:



How is it Distributed?

Model parallelism?

When the model is too big to fit into memory on one machine, one can assign different parts of the graph to different machines. The parameters will live on their part of the cluster, and their training and update operations will happen locally on those nodes.

How is it Distributed?

Model parallelism?

When the model is too big to fit into memory on one machine, one can assign different parts of the graph to different machines. The parameters will live on their part of the cluster, and their training and update operations will happen locally on those nodes.

Not such a great idea!

A basic way to do it is to have the first layers on a machine, the next layers on another machine, etc.

If we do this, the deeper layers have to wait for the first layers during the forward pass, and the first layers need to wait for the deeper layers during the backpropagation.

You don’t get a lot of parallelism this way!

Data Parallelism is Better

Entire graph will live on one potentially replicated machine called the *parameter server*.

This is good because in the case of lots of I/O, the entire graph can live on several parameter servers to reduce the cost of I/O.

Data Parallelism is Better

Entire graph will live on one potentially replicated machine called the *parameter server*.

This is good because in the case of lots of I/O, the entire graph can live on several parameter servers to reduce the cost of I/O.

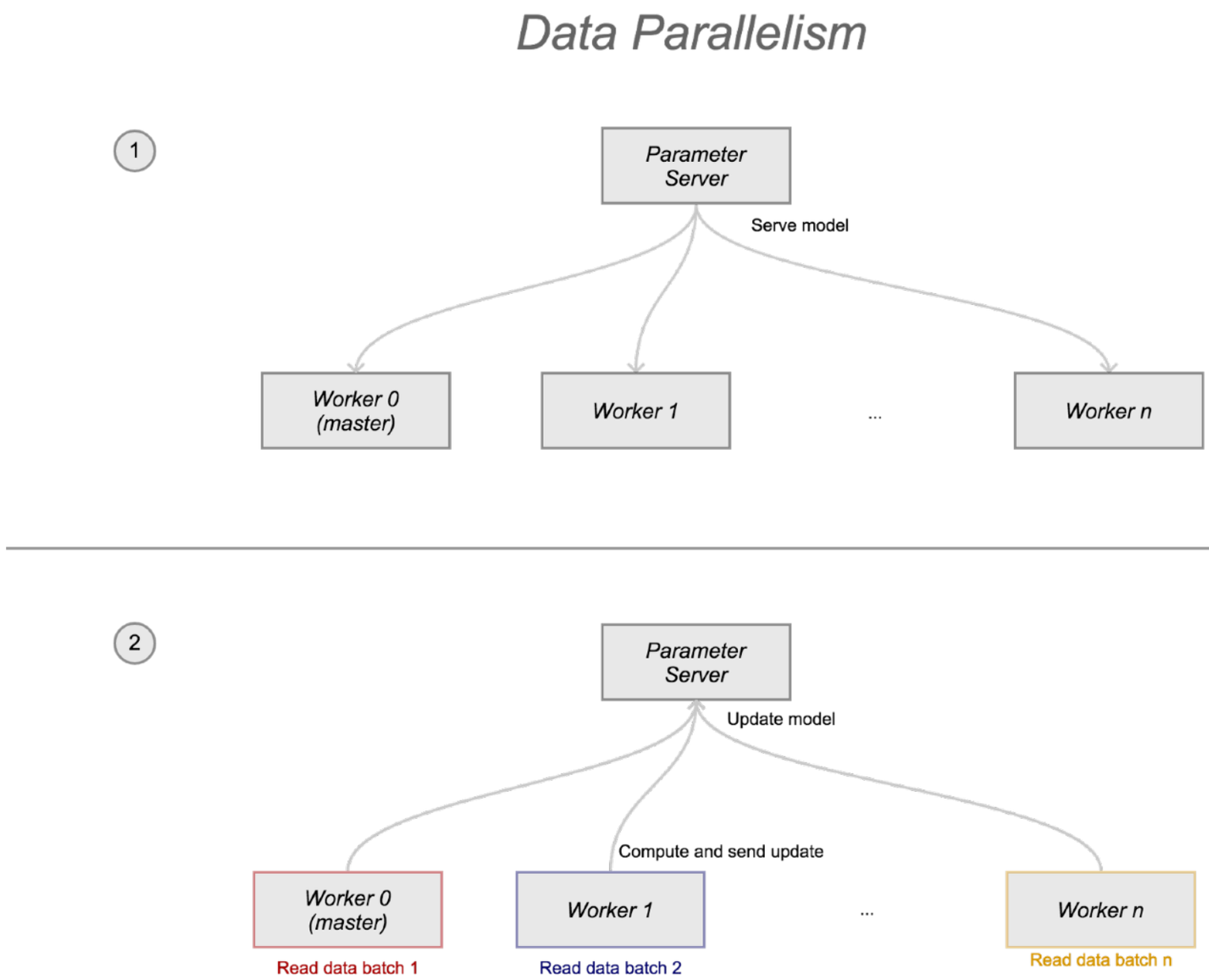
Tasks like training are then executed on *multiple workers*.

Workers:

- read different data batches
- compute gradients
- send update operations to parameter servers

Data Parallelism is Better

Entire graph will live on one potentially replicated machine called the *parameter server*.



Synchrony vs Asynchrony

Two ways to go about reading parameters, gradient computation, and sending updated parameters back to parameter server:

- **Synchronous training:** all the workers will read the parameters at the same time, compute a training operation and wait for all the others to be done. Then the gradients will be averaged and a single update will be sent to the parameter server. So at any point in time, the workers will all be aware of the same values for the graph parameters

- **Asynchronous training:** the workers will read from the parameter server(s) asynchronously, compute their training operation, and send asynchronous updates. At any point in time, two different workers might be aware of different values for the graph parameters

Other Ways to Distribute…

It's also possible to place specific operations on a particular device, like a CPU or GPU.

```
with tf.device("/job:ps/task:0"):
  weights_1 = tf.Variable( ... )
  biases_1 = tf.Variable( ... )

with tf.device("/job:ps/task:1"):
  weights_2 = tf.Variable( ... )
  biases_2 = tf.Variable( ... )

with tf.device("/job:worker/task:7"):
  input, labels =  ...
  layer_1 = tf.nn.relu(tf.matmul(input, weights_1) + biases_1)
  logits = tf.nn.relu(tf.matmul(layer_1, weights_2) + biases_2)
  # ...
  train_op =  ...

with tf.Session("grpc://worker7.example.com:2222") as sess:
  for _ in range(10000):
    sess.run(train_op)
```

Other Ways to Distribute…

It's also possible to place specific operations on a particular device, like a CPU or GPU.

```
with tf.device("/job:ps/task:0"):
  weights_1 = tf.Variable( ... )
  biases_1 = tf.Variable( ... )

with tf.device("/job:ps/task:1"):
  weights_2 = tf.Variable( ... )
  biases_2 = tf.Variable( ... )

with tf.device("/job:worker/task:7"):
  input, labels =  ...
  layer_1 = tf.nn.relu(tf.matmul(input, weights_1) + biases_1)
  logits = tf.nn.relu(tf.matmul(layer_1, weights_2) + biases_2)
  # ...
  train_op =  ...

with tf.Session("grpc://worker7.example.com:2222") as sess:
  for _ in range(10000):
    sess.run(train_op)
```

Do this on the parameter server

Do this on the worker machine