TensorFlow Tutorial

Chia-Po Wei

Department of Electrical Engineering
National Sun Yat-sen University

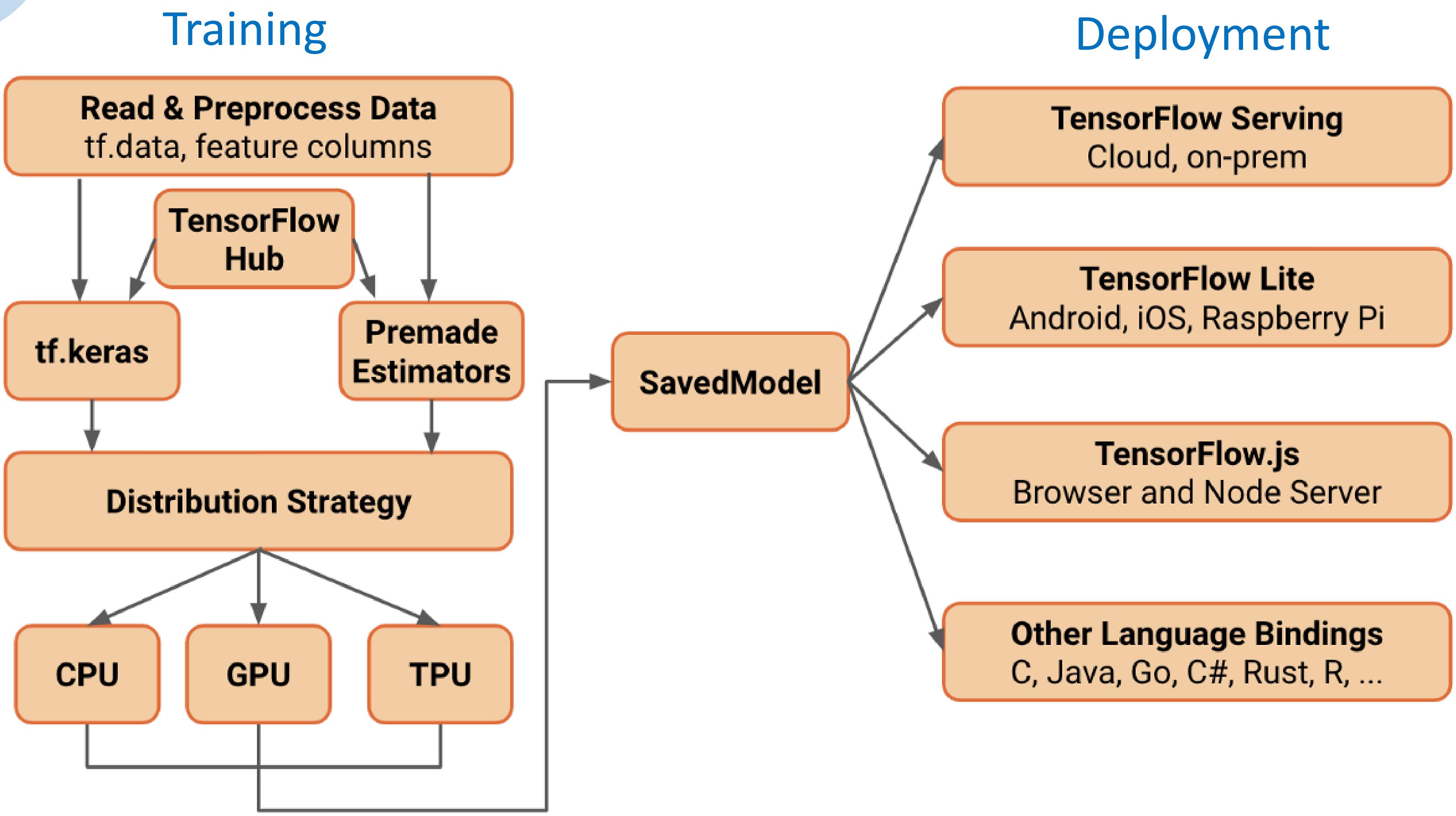
Outline

- Deep Learning Framework
- TensorFlow Core Walkthrough
  - Tensor, Graph, Session
  - tf.data.Dataset
  - Symbolic Execution vs. Eager Execution
- Fitting a Linear Model

What is TensorFlow?

- TensorFlow is an end-to-end open source machine learning platform.
- Tensorflow helps you develop and train Machine Learning models.

Tensorflow 2.0 (Conceptual Diagram)



Deep Learning Frameworks

- Tensorflow (Google), Pytorch (Facebook), Caffe2 (Facebook), mxnet (Apache), CNTK (Microsoft)

ONNX (Open Neural Network Exchange)

- ONNX is a open format to represent deep learning models.
- ONNX allows you to switch between deep learning frameworks.
- Facebook, Microsoft, Amazon announced support for ONNX, but Google hasn’t.

ONNX

TensorFlow vs. Numpy

- Both TensorFlow and Numpy provide support for large multi-dimensional (N-d) arrays with a collection of mathematical functions.
- But Numpy cannot automatically compute derivatives and cannot take advantage of GPU.

vs.

Numpy and TensorFlow Comparison

Numpy	TensorFlow
a = np.zeros((2,2)); b = np.ones((2,2))	a = tf.zeros((2,2)); b = tf.ones((2,2))
np.sum(b, axis=1)	tf.reduce_sum(b, axis=1)
a.shape	a.get_shape() or tf.shape(a)
np.reshape(a, (1,5))	tf.reshape(a, (1,5))
a * 7 + 1	a * 7 + 1
np.dot(a, b)	tf.matmul(a, b)
a[0, 1], a[:, 1], a[0,:]	a[0, 1], a[:, 1], a[0,:]

How to install TensorFlow?

- After installing **anaconda (Python 3.7)**
  - CPU version: `conda install tensorflow`
  - GPU version: `conda install tensorflow-gpu`
  - Prefer `conda install` to `pip install` if you want to install the GPU version.
    - Because you don’t have to struggle to find the compatible NVIDIA GPU drivers and CUDA Toolkits. conda will do the job for you.
- It is recommended that you learn to setup a virtual environment using conda.
- This allows you to switch between DL frameworks easily.

Anaconda

Setup your Python environment

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf

print(tf.__version__) # print the version of TensorFlow
```

Tensor: rank, shape

- The central unit of data in TensorFlow is called the tensor.
- A tensor's rank is its number of dimensions.
- A tensor's shape is a tuple of integers specifying the array's length along each dimension.

```
3. # rank 0 tensor; a scalar with shape []
[1., 2., 3.] # a rank 1 tensor; a vector with shape [3]
[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]
[[[1., 2., 3.]], [[2., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]
```

Quiz 1

Determine the rank and shape of the following Tensors.

1. [[[1, 2, 3], [4, 5, 6]]]
2. [[[1], [3], [5]], [[2], [4], [6]]]

Tensor Examples

- A tensor has a **data type** (float32, int32, string, etc.) and a **shape**.
- Each element in a tensor has **the same** data type, and the data type is known.

Four types of tensors:

- tf.Variable
- tf.constant
- tf.placeholder
- tf.SparseTensor

Rank 0 examples:

- mammal = tf.Variable("Elephant", tf.string)
- ignition = tf.Variable(451, tf.int16)
- floating = tf.Variable(3.14159265359, tf.float64)
- complicated = tf.Variable(12.3 - 4.85j, tf.complex64)

Rank 1 examples:

- mammal = tf.Variable(["Elephant"], tf.string)
- ignition = tf.Variable([123, 456], tf.int16)

TensorFlow Core Walkthrough

- A typical TensorFlow program consists of two discrete sections:
  - Building the computational graph (tf.Graph)
  - Running the computational graph (using tf.Session)
- A graph is composed of
  - tf.Operation (the nodes): Operations that consume and produce tensors.
  - tf.Tensor (the edges): Represent the values that will flow through the graph.

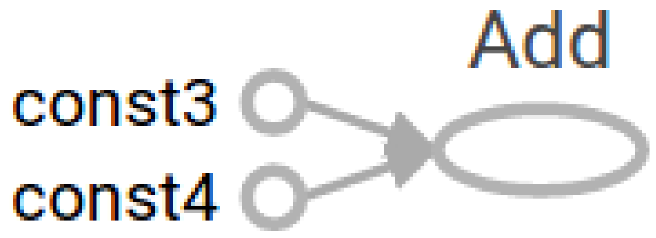
★ tf.Tensors do not have values, they are just handles to elements in the graph.

Simple Graph

```
a = tf.constant(3.0, dtype=tf.float32)
b = tf.constant(4.0) # also tf.float32 implicitly
total = a + b
print(a)
print(b)
print(total)
```

Output:

```
Tensor("Const:0", shape=(), dtype=float32)
Tensor("Const_1:0", shape=(), dtype=float32)
Tensor("add:0", shape=(), dtype=float32)
```



Session

- To evaluate tensors, a tf.Session object is instantiated.
- If a tf.Graph is like a .py file, a tf.Session is like the python executable.

```
a = tf.constant(3.0, dtype=tf.float32)
b = tf.constant(4.0)
total = a + b
sess = tf.Session()
print(sess.run(total))
print(sess.run({'ab': (a, b), 'total': total}))
```

Output:

```
7.0
{'total': 7.0, 'ab': (3.0, 4.0)}
```

Placeholder for Feeding Data

- A graph can be parameterized to accept external inputs, known as placeholders.
- A placeholder is a promise to provide a value later.

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

tf.data.Dataset

- Placeholders work for simple experiments, but tf.data are the **preferred** method of streaming data into a model.
- 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 tf.data.Iterator.get_next method.

```
my_data = [[0, 1,], [2, 3,], [4, 5,], [6, 7,]]
slices = tf.data.Dataset.from_tensor_slices(my_data)
next_item =
slices.make_one_shot_iterator().get_next()
```

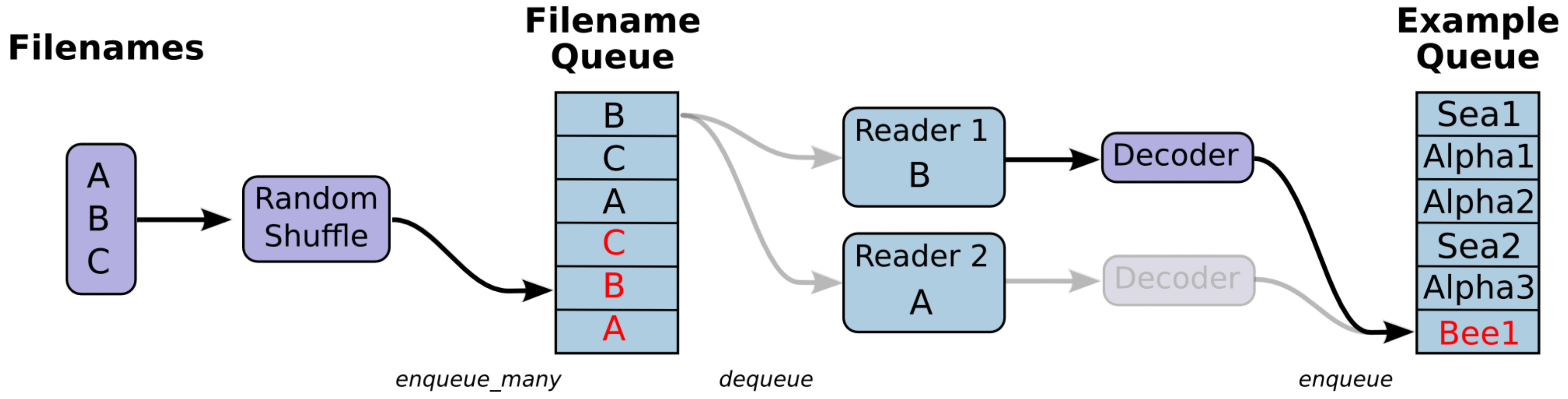
```
sess = tf.Session()
while True:
  try:
    print(sess.run(next_item))
  except tf.errors.OutOfRangeError:
    break
```

tf.data.Dataset (cont.)

```
# Read records from a list of files.
dataset = TFRecordDataset(["file1.tfrecord", "file2.tfrecord", …])
# Parse string values into tensors.
dataset = dataset.map(lambda record: tf.parse_single_example(record, …))
# Randomly shuffle using a buffer of 10000 examples.
dataset = dataset.shuffle(10000)
# Repeat for 100 epochs.
dataset = dataset.repeat(100)
# Combine 128 consecutive elements into a batch.
dataset = dataset.batch(128)
```

- tf.data.Dataset has been introduced since the release of TensorFlow 1.4.

Queue-based Pipelines (DEPRECATED)



- tf.train.string_input_producer outputs strings to a queue for an input pipeline.
- Queue-based input pipelines have been replaced by tf.data.

How are Machine Learning Models represented?

Model is a **Data Structure**
(e.g. a Graph)

aka

- Symbolic
- Deferred Execution
- Defined-and-run

Model is a **Program**
(e.g. Python Code)

aka

- Imperative
- Eager Execution
- Defined-by-run

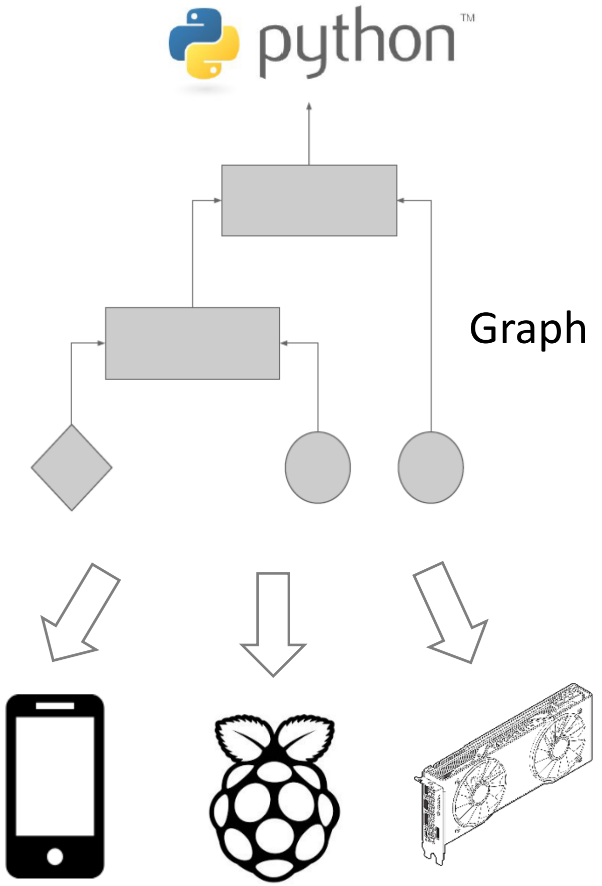
Why Symbolic Execution?

Pros (Makes (de)serialization easier)

- Parallelism
- Distributed execution (CPUs, GPUs, TPUs)
- Compilation (XLA compiler)
- Portability (language independent)

Cons (Less intuitive)

- Harder to debug
- Harder to write control flow structures
- Harder to write dynamic models



Why Eager Execution?

- Eager execution evaluates operations **immediately, without building graphs**: operations return concrete values instead of constructing a graph to run later.
- **Advantages**
  - An intuitive interface—Structure your code naturally and use Python data structures. Quickly iterate on small models and small data.
  - Easier debugging—Call ops directly to inspect running models and test changes. Use standard Python debugging tools for immediate error reporting.
  - Natural control flow—Use Python control flow instead of graph control flow, simplifying the specification of dynamic models.

Eager Execution (cont.)

```
a = tf.constant(3.0, dtype=tf.float32)
b = tf.constant(4.0)
total = a + b
print(total)
```

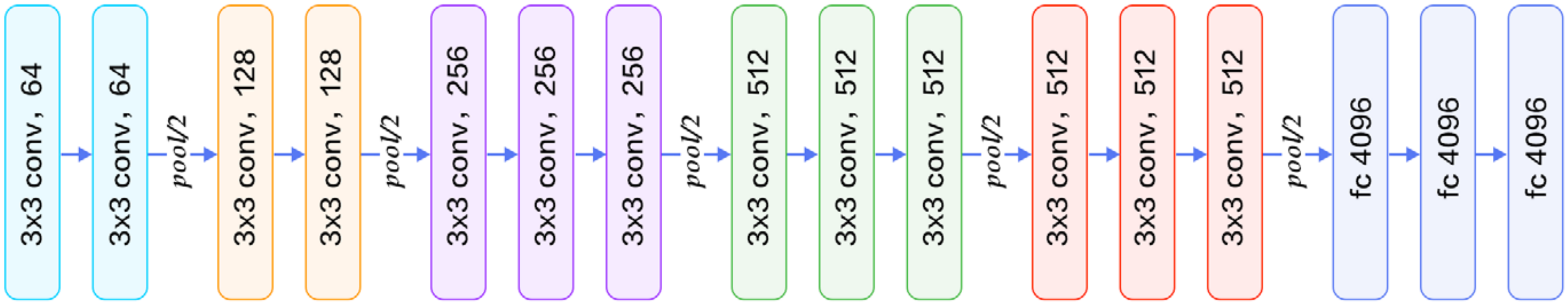
Output:
Tensor("add:0", shape=(), dtype=float32)

```
tf.enable_eager_execution()
a = tf.constant(3.0, dtype=tf.float32)
b = tf.constant(4.0)
total = a + b
print(total)
```

Output:
tf.Tensor(7.0, shape=(), dtype=float32)

```
# try print(total.numpy())
```

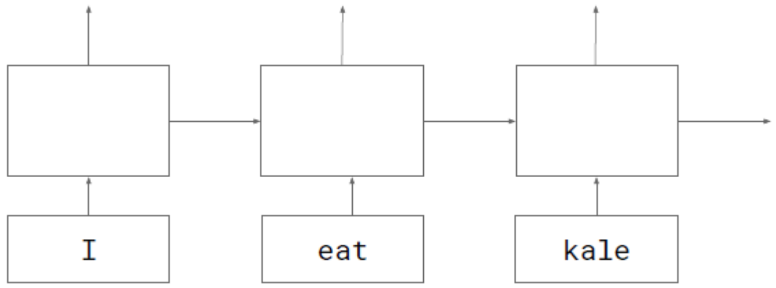
Static Model Structures



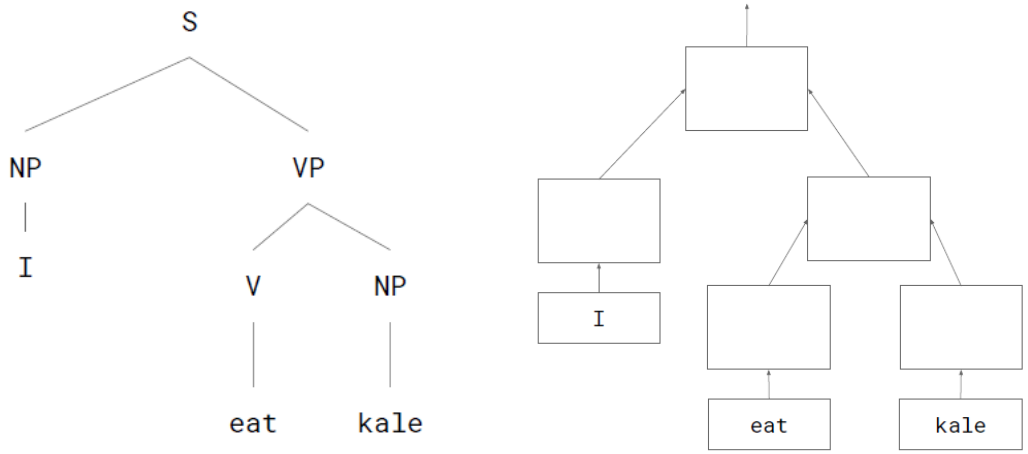
- Model structure is fixed regardless of input data
- The majority of DL models for image, audio and numerical data

Dynamic Model Structures

Traditional RNN



Tree RNN (Dynamic Models)



- Structures that change a lot with input data
- Can deal with hierarchical structures in natural language
- Difficult to write in the symbolic way (using tf.cond and tf.while_loop)
- Straightforward with Eager (Using the native Python control flow)

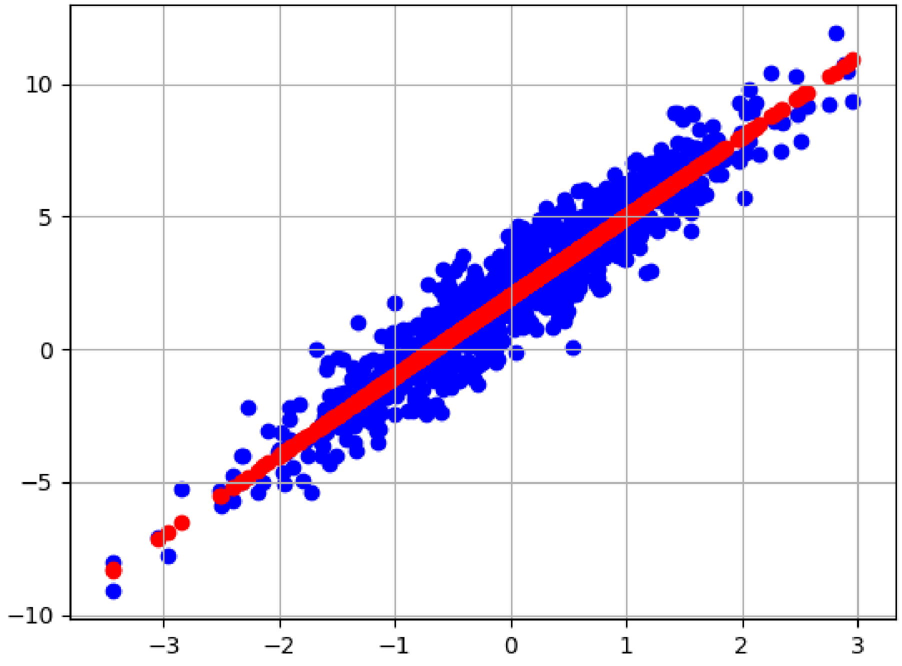
Fitting a Linear Model

- We will walk through the example of a simple linear model $f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$, which has two variables $\mathbf{W}$ and $\mathbf{b}$.

1. Obtain training data
2. Define the model
3. Define a loss function
4. Run through the training data

Step 1. Obtaining Training Data

- Synthesize data such that a well trained model would have W = 3.0 and b = 2.0




```
TRUE_W = 3.0
TRUE_b = 2.0
NUM_EXAMPLES = 1000
inputs  = tf.random_normal(shape=[NUM_EXAMPLES])
noise   = tf.random_normal(shape=[NUM_EXAMPLES])
outputs = inputs * TRUE_W + TRUE_b + noise
```

- Given inputs and outputs, our goal is to learn W and b such that f(inputs) = outputs.

Step 2. Define the Model

```
class Model(object):
    def __init__(self):
        self.W = tf.Variable(-1.0)
        self.b = tf.Variable(1.0)
    def __call__(self, x):
        return self.W * x + self.b

model = Model()
assert model(3.0).numpy() == 15.0
```

$$f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Step 3. Define the Loss Function

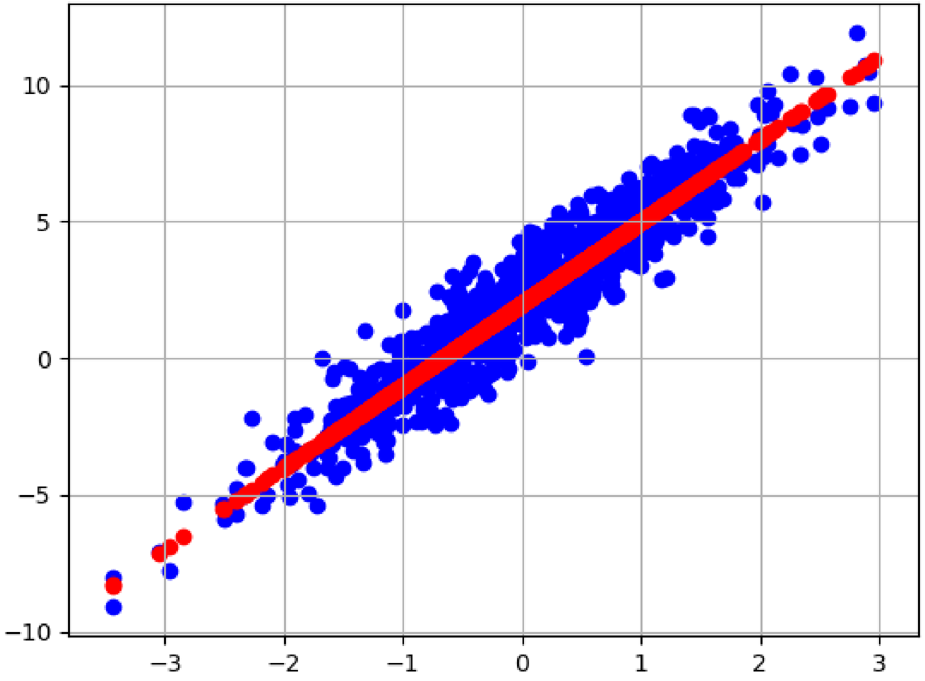
```
def loss(predicted_y, desired_y):
    return tf.reduce_mean(tf.square(predicted_y - desired_y))

def train(model, inputs, outputs, learning_rate):
    with tf.GradientTape() as tape:
        current_loss = loss(model(inputs), outputs)
    dW, db = tape.gradient(current_loss, [model.W, model.b])
    model.W.assign_sub(learning_rate * dW) # W -= learning_rate * dW
    model.b.assign_sub(learning_rate * db) # b -= learning_rate *db
    return current_loss
```

Step 4. Run Through the Training Data

```
model = Model()
Ws, bs = [], []
epochs = range(10)
for epoch in epochs:
    current_loss =  train(model, inputs, outputs, learning_rate=0.1)
    Ws.append(model.W.numpy())
    bs.append(model.b.numpy())
    print(epoch, Ws[-1], bs[-1], current_loss)
```

Simulation Results

```
Epoch  0: W=-1.00 b=1.00, loss=18.29831
Epoch  1: W=-0.19 b=1.18, loss=12.15019
Epoch  2: W=0.45 b=1.33, loss=8.18830
Epoch  3: W=0.97 b=1.45, loss=5.63520
Epoch  4: W=1.39 b=1.55, loss=3.98991
Epoch  5: W=1.72 b=1.63, loss=2.92962
Epoch  6: W=1.99 b=1.69, loss=2.24632
Epoch  7: W=2.20 b=1.74, loss=1.80596
Epoch  8: W=2.37 b=1.79, loss=1.52217
Epoch  9: W=2.51 b=1.82, loss=1.33927
                       ⋮
Epoch 29: W=3.07 b=1.96, loss=1.00772
```



- Blue dots are training data
- Red dots are prediction results

References

- TensorFlow Introduction (https://www.tensorflow.org/guide/low_level_intro)
- Tensors (https://www.tensorflow.org/guide/tensors)
- Graphs and Sessions (https://www.tensorflow.org/guide/graphs)
- Eager Execution (https://www.tensorflow.org/guide/eager)
- Fitting a Linear Model (https://www.tensorflow.org/tutorials/eager/custom_training)