TensorFlow

Amir H. Payberah
payberah@kth.se
23/11/2018

The Course Web Page

https://id2223kth.github.io

Where Are We?

Deep Learning

CNN | RNN | Autoencoder

Deep Feedforward Network | Training Feedforward Network

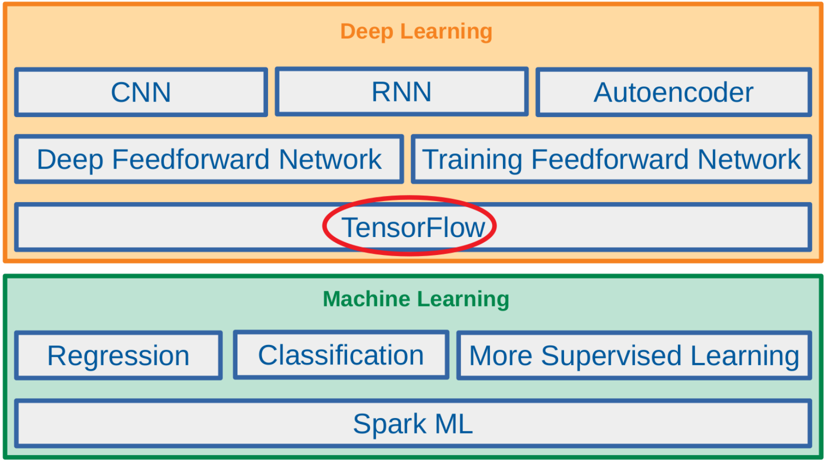
TensorFlow

Machine Learning

Regression | Classification | More Supervised Learning

Spark ML

Where Are We?



Deep Learning
CNN
RNN
Autoencoder
Deep Feedforward Network
Training Feedforward Network
TensorFlow
Machine Learning
Regression
Classification
More Supervised Learning
Spark ML

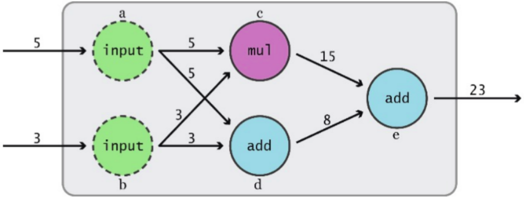
Introduction

- TensorFlow is an open source software library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning.
- Was developed by the Google Brain team.

Hello World

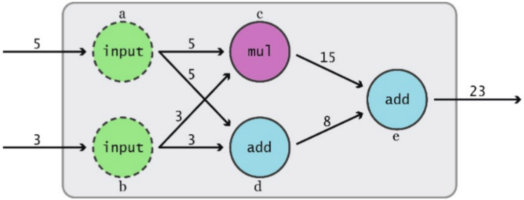
- Implement machine learning algorithms by creating and computing operations that interact with one another.

$$e = c + d$$
$$c = a \times b$$
$$d = a + b$$



Two Phases of Tensorflow

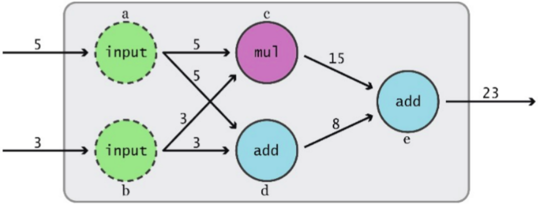
- Working with TensorFlow involves two main phases.
  1. Build a graph
  2. Execute it



Phase 1: Build a Graph

- `import tensorflow as tf`: it forms an empty default graph.

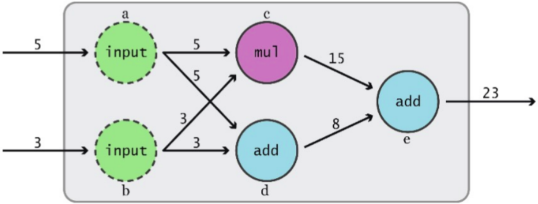
```
import tensorflow as tf

a = tf.constant(5)
b = tf.constant(3)

c = tf.multiply(a, b)
d = tf.add(a, b)
e = tf.add(c, d)
```



Phase 1: Build a Graph

- `import tensorflow as tf`: it forms an empty default graph.
- First, add two nodes to output a constant value

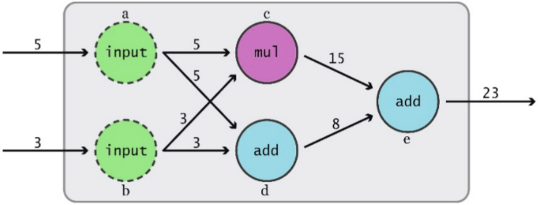
```
import tensorflow as tf

a = tf.constant(5)
b = tf.constant(3)

c = tf.multiply(a, b)
d = tf.add(a, b)
e = tf.add(c, d)
```



Phase 1: Build a Graph

- `import tensorflow as tf`: it forms an empty default graph.
- First, add two nodes to output a constant value
- Each of the next three nodes gets two existing variables as inputs, and performs simple arithmetic operations on them, and generates outputs.

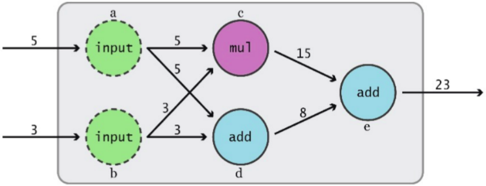
```
import tensorflow as tf

a = tf.constant(5)
b = tf.constant(3)

c = tf.multiply(a, b)
d = tf.add(a, b)
e = tf.add(c, d)
```



Phase 2: Execute a Graph

- Now, we are ready to run the computations.

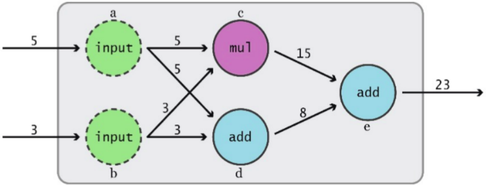
```
sess = tf.Session()
print(sess.run(e))
sess.close()

# Alternative way
with tf.Session() as sess:
  print(sess.run(e))
```



Phase 2: Execute a Graph

- Now, we are ready to run the computations.
- Create and run a session, by calling the `run()` method of the `Session` object.

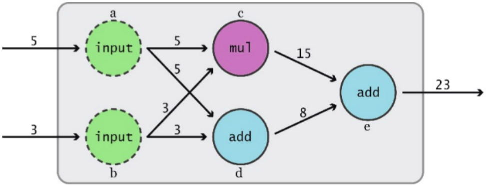
```
sess = tf.Session()
print(sess.run(e))
sess.close()

# Alternative way
with tf.Session() as sess:
  print(sess.run(e))
```



Phase 2: Execute a Graph

- Now, we are ready to run the computations.
- Create and run a session, by calling the `run()` method of the `Session` object.
- When `sess.run(e)` is called, it starts at the requested output `e`, and then works backward, computing nodes that must be executed.

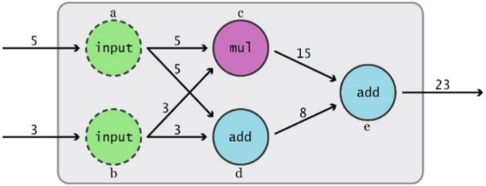
```
sess = tf.Session()
print(sess.run(e))
sess.close()

# Alternative way
with tf.Session() as sess:
  print(sess.run(e))
```



Phase 2: Execute a Graph

- Now, we are ready to run the computations.
- Create and run a session, by calling the `run()` method of the `Session` object.
- When `sess.run(e)` is called, it starts at the requested output `e`, and then works backward, computing nodes that must be executed.
- Close the session at the end of the computation, using the `sess.close()` command.

```python
sess = tf.Session()
print(sess.run(e))
sess.close()

# Alternative way
with tf.Session() as sess:
  print(sess.run(e))
```



The Complete Code

```python
import tensorflow as tf

# Building the Graph
a = tf.constant(5)
b = tf.constant(3)

c = tf.multiply(a, b)
d = tf.add(a, b)
e = tf.add(c, d)

# Executing the Graph
with tf.Session() as sess:
  print(sess.run(e))
```

Visualize the Code

```python
import tensorflow as tf

# Building the Graph
a = tf.constant(5)
b = tf.constant(3)

c = tf.multiply(a, b)
d = tf.add(a, b)
e = tf.add(c, d)

writer = tf.summary.FileWriter("./graphs", tf.get_default_graph())

# Executing the Graph
with tf.Session() as sess:
  print(sess.run(e))
```

Visualize the Code

```python
import tensorflow as tf

# Building the Graph
a = tf.constant(5)
b = tf.constant(3)

c = tf.multiply(a, b)
d = tf.add(a, b)
e = tf.add(c, d)

writer = tf.summary.FileWriter("./graphs", tf.get_default_graph())

# Executing the Graph
with tf.Session() as sess:
  print(sess.run(e))
```

```
tensorboard --logdir="./graphs" --port 6006
```

Let's Give Name to Variables

```python
import tensorflow as tf

# Building the Graph
a = tf.constant(5, name="a")
b = tf.constant(3, name="b")

c = tf.multiply(a, b, name="c_mul")
d = tf.add(a, b, name="d_add")
e = tf.add(c, d, name="e_add")

writer = tf.summary.FileWriter("./graphs", tf.get_default_graph())

# Executing the Graph
with tf.Session() as sess:
  print(sess.run(e))
```

```
tensorboard --logdir="./graphs" --port 6006
```

Tensor Objects

What is Tensor?

- The central unit of data in TensorFlow is the tensor.
- An n-dimensional array of primitive values.

Tensor Objects

- The main object you manipulate and pass around is the `tf.Tensor`.
- TensorFlow programs work by building a graph of `tf.Tensor` objects, and running parts of this graph.
- Each Tensor object is specified by:
  - Rank
  - Shape
  - Datatype

Tensor Objects - Rank

- The number of dimensions.
  - rank 0: scalar (number), e.g., 5
  - rank 1: vector, e.g., [2, 5, 7]
  - rank 2: matrix, e.g., [[1, 2], [3, 4], [5, 6]]
  - rank 3: 3-Tensor
  - rank n: n-Tensor

- The `tf.rank` method determines the rank of a `tf.Tensor` object.

```
c = tf.constant([[4], [9], [16], [25]])

r = tf.rank(c) # rank 2
```

Tensor Objects - Shape

- The number of elements in each dimension.
- The `get_shape()` returns the shape of the tensor as a tuple of integers.

```
c = tf.constant([[[1, 2, 3], [4, 5, 6]],
                 [[1, 1, 1], [2, 2, 2]]])

s = c.get_shape() # (2, 2, 3)
```

Tensor Objects - Data Types (1/2)

- We can explicitly choose the data type a Tensor object.
- Make sure the data types match throughout the graph.
- We can use the `tf.cast()` method to change the data type of a Tensor object.

```
c = tf.constant(4.0, dtype=tf.float64)

x = tf.constant([1, 2, 3], dtype=tf.float32)
y = tf.cast(x, tf.int64)
```

Tensor Objects - Data Types (2/2)

Data type	Python type	Description
DT_FLOAT	tf.float32	32-bit floating point.
DT_DOUBLE	tf.float64	64-bit floating point.
DT_INT8	tf.int8	8-bit signed integer.
DT_INT16	tf.int16	16-bit signed integer.
DT_INT32	tf.int32	32-bit signed integer.
DT_INT64	tf.int64	64-bit signed integer.
DT_UINT8	tf.uint8	8-bit unsigned integer.
DT_UINT16	tf.uint16	16-bit unsigned integer.
DT_STRING	tf.string	Variable-length byte array. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32-bit floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64-bit floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8-bit signed integer used in quantized ops.
DT_QINT32	tf.qint32	32-bit signed integer used in quantized ops.
DT_QUINT8	tf.quint8	8-bit unsigned integer used in quantized ops.

Tensor Objects - Name

- Each Tensor object has an identifying name.
- This name is an intrinsic string name, not to be confused with the name of the variable.

```
c = tf.constant(4.0, dtype=tf.float64, name="input")
```

Tensor Objects - Name Scopes

- To deal with large graphs, we can use node grouping to make it easier to manage.

Tensor Objects - Name Scopes

- To deal with large graphs, we can use node grouping to make it easier to manage.
- Hierarchically group nodes by their names, using `tf.name_scope()` together with the `with` clause.

```
with tf.name_scope("myprefix"):
  c1 = tf.constant(4.0, dtype=tf.int32, name="input1")
  c2 = tf.constant(4.0, dtype=tf.float64, name="input2")
```

Tensor Objects - Name Scopes

- To deal with large graphs, we can use node grouping to make it easier to manage.
- Hierarchically group nodes by their names, using `tf.name_scope()` together with the `with` clause.
- Below, the name of each operation within the scope is prefixed with `myprefix/`, e.g., `myprefix/input1`.

```
with tf.name_scope("myprefix"):
  c1 = tf.constant(4.0, dtype=tf.int32, name="input1")
  c2 = tf.constant(4.0, dtype=tf.float64, name="input2")
```

Main Types of Tensors

- Constants `tf.constant`
- Variables `tf.Variable`
- Placeholders `tf.placeholder`

Constants

Constants (1/3)

- The value of constant Tensor cannot be changed in the future.

```
tf.constant(value, dtype=None, shape=None, name="Const", verify_shape=False)

a = tf.constant([[0, 1], [2, 3]], name="b")
b = tf.constant([[4], [9], [16], [25]], name="c")
```

Constants (2/3)

- The initialization should be with a value, not with operation.

TensorFlow operation	Description
`tf.constant(value)`	Creates a tensor populated with the value or values specified by the argument *value*
`tf.fill(shape, value)`	Creates a tensor of shape *shape* and fills it with *value*
`tf.zeros(shape)`	Returns a tensor of shape *shape* with all elements set to 0
`tf.zeros_like(tensor)`	Returns a tensor of the same type and shape as *tensor* with all elements set to 0
`tf.ones(shape)`	Returns a tensor of shape *shape* with all elements set to 1
`tf.ones_like(tensor)`	Returns a tensor of the same type and shape as *tensor* with all elements set to 1
`tf.random_normal(shape, mean, stddev)`	Outputs random values from a normal distribution
`tf.truncated_normal(shape, mean, stddev)`	Outputs random values from a truncated normal distribution (values whose magnitude is more than two standard deviations from the mean are dropped and re-picked)
`tf.random_uniform(shape, minval, maxval)`	Generates values from a uniform distribution in the range [*minval*, *maxval*)
`tf.random_shuffle(tensor)`	Randomly shuffles a tensor along its first dimension

Constants (3/3)

- What's wrong with constants?

Constants (3/3)

- What's wrong with constants?
- Constants are stored in the graph definition.
- This makes loading graphs expensive when constants are big.

Constants (3/3)

- What's wrong with constants?
- Constants are stored in the graph definition.
- This makes loading graphs expensive when constants are big.
- Only use constants for primitive types.
- Use variables for more data that requires more memory.

Variables

Variables

- A variable represents a Tensor whose value can be changed by running ops on it.

Variables

- A variable represents a Tensor whose value can be changed by running ops on it.
- `tf.Variable` is a class with several ops.

Variables

- A variable represents a Tensor whose value can be changed by running ops on it.
- `tf.Variable` is a class with several ops.
- Create variables with `tf.get_variable`.
- `tf.get_variable` returns an existing variable with the given parameters if it is available.

```
# not recommended way to make a variable
tf.Variable(<initial-value>, name)

w = tf.Variable([[0, 1], [2, 3]], name="matrix")

# recommended
tf.get_variable(name, shape=None, dtype=tf.float32, initializer=None,
       regularizer=None, trainable=True, collections=None)

w = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
```

Initialize Variables

- Variables should be initialized before being used.

Initialize Variables

- Variables should be initialized before being used.
- Initialize all variables at once.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```

Initialize Variables

- Variables should be initialized before being used.
- Initialize all variables at once.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```

- Initialize only a subset of variables.

```
with tf.Session() as sess:
    sess.run(tf.variables_initializer([a, b]))
```

Initialize Variables

- Variables should be initialized before being used.
- Initialize all variables at once.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```

- Initialize only a subset of variables.

```
with tf.Session() as sess:
    sess.run(tf.variables_initializer([a, b]))
```

- Initialize a single variable.

```
w = tf.Variable(tf.zeros([784,10]))

with tf.Session() as sess:
    sess.run(w.initializer)
```

Assign Values to Variables (1/3)

- What does it print?

```
w = tf.get_variable("scalar", initializer=tf.constant(2))
w.assign(100)

with tf.Session() as sess:
    sess.run(w.initializer)
    print(sess.run(w))
```

Assign Values to Variables (1/3)

- What does it print?

```
w = tf.get_variable("scalar", initializer=tf.constant(2))
w.assign(100)

with tf.Session() as sess:
    sess.run(w.initializer)
    print(sess.run(w))
```

- Prints 2, because `w.assign(100)` creates an assign op.
- That op needs to be executed in a session to take effect.

```
w = tf.get_variable("scalar", initializer=tf.constant(2))
assign_op = w.assign(100)

with tf.Session() as sess:
    sess.run(w.initializer)
    sess.run(assign_op)
    print(sess.run(w))
```

Assign Values to Variables (2/3)

- What does it print?

```
w = tf.get_variable("scalar", initializer=tf.constant(2))
w_times_two = w.assign(2 * w)

with tf.Session() as sess:
    sess.run(w.initializer)
    print(sess.run(w_times_two))
    print(sess.run(w_times_two))
    print(sess.run(w_times_two))
```

Assign Values to Variables (3/3)

- `assign_add()` and `assign_sub()`

```
w = tf.get_variable("scalar", initializer=tf.constant(2))

with tf.Session() as sess:
    sess.run(w.initializer)

    # increment by 10
    print(sess.run(w.assign_add(10)))

    # decrement by 5
    print(sess.run(w.assign_sub(5)))
```

Placeholders

Placeholders

- Placeholders are built-in structures for feeding input values.
- Empty variables that will be filled with data later on.
- `shape=None` means that tensor of any shape will be accepted.
  - E.g., `shape=[None, 10]`: a matrix with 10 columns and any number of rows.

```
tf.placeholder(dtype, shape=None, name=None)

x = tf.placeholder(tf.float32, shape=[None, 10])
```

Feeding Placeholders (1/2)

- What's wrong with this code?

```
a = tf.placeholder(tf.float32, shape=[3])

b = tf.constant([5, 5, 5], tf.float32)

c = a + b

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

Feeding Placeholders (2/2)

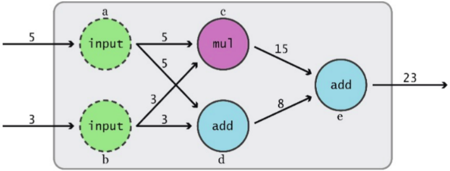
- Supplement the values to placeholders using a dictionary.

```
a = tf.placeholder(tf.float32, shape=[3])

b = tf.constant([5, 5, 5], tf.float32)

c = a + b

with tf.Session() as sess:
  print(sess.run(c, feed_dict={a: [1, 2, 3]}))
```

Dataflow Graphs

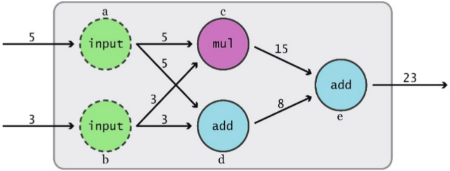
Graph

- A computational graph is a series of TensorFlow operations arranged into a graph.



Graph

- A computational graph is a series of TensorFlow operations arranged into a graph.
- The graph is composed of two types of objects:
  - Operations: the nodes of the graph that that consume and produce tensors.
  - Tensors: the edges in the graph that represent the flowing values through the graph.



Common Tensorflow Operations

TensorFlow operator	Shortcut	Description
`tf.add()`	`a + b`	Adds a and b, element-wise.
`tf.multiply()`	`a * b`	Multiplies a and b, element-wise.
`tf.subtract()`	`a - b`	Subtracts a from b, element-wise.
`tf.divide()`	`a / b`	Computes Python-style division of a by b.
`tf.pow()`	`a ** b`	Returns the result of raising each element in a to its corresponding element b, element-wise.
`tf.mod()`	`a % b`	Returns the element-wise modulo.
`tf.logical_and()`	`a & b`	Returns the truth table of `a & b`, element-wise. `dtype` must be `tf.bool`.
`tf.greater()`	`a > b`	Returns the truth table of `a > b`, element-wise.
`tf.greater_equal()`	`a >= b`	Returns the truth table of `a >= b`, element-wise.
`tf.less_equal()`	`a <= b`	Returns the truth table of `a <= b`, element-wise.
`tf.less()`	`a < b`	Returns the truth table of `a < b`, element-wise.
`tf.negative()`	`-a`	Returns the negative value of each element in a.
`tf.logical_not()`	`~a`	Returns the logical NOT of each element in a. Only compatible with Tensor objects with `dtype` of `tf.bool`.
`tf.abs()`	`abs(a)`	Returns the absolute value of each element in a.
`tf.logical_or()`	`a | b`	Returns the truth table of `a | b`, element-wise. `dtype` must be `tf.bool`.

Managing Multiple Graphs (1/2)

- When we call `import tensorflow`, a default graph is automatically created.

Managing Multiple Graphs (1/2)

- When we call `import tensorflow`, a default graph is automatically created.
- We can also create additional graphs, by calling `tf.Graph()`.

Managing Multiple Graphs (1/2)

- When we call `import tensorflow`, a default graph is automatically created.
- We can also create additional graphs, by calling `tf.Graph()`.
- `tf.get_default_graph()` tells which graph is currently set as the default graph.

```
import tensorflow as tf

g = tf.Graph()
a = tf.constant(5)

print(a.graph is g)
# Out: False

print(a.graph is tf.get_default_graph())
# Out: True
```

Managing Multiple Graphs (2/2)

- Use `with` together with `as_default()` to associate your constructed nodes the a right graph.

```
import tensorflow as tf

g1 = tf.get_default_graph()
g2 = tf.Graph()

print(g1 is tf.get_default_graph())
# Out: True

with g2.as_default():
  print(g1 is tf.get_default_graph())
# Out: False
  print(g2 is tf.get_default_graph())
# Out: True
```

Session

Session

- A `Session` object encapsulates the environment.
- Operation objects are executed, and Tensor objects are evaluated.
- Session will also allocate memory to store the current values of variables.

Session

- A `Session` object encapsulates the environment.
- Operation objects are executed, and Tensor objects are evaluated.
- Session will also allocate memory to store the current values of variables.

```
sess = tf.Session()
outs = sess.run(e)
print("outs = {}".format(outs))
sess.close()
```

Session

- A `Session` object encapsulates the environment.
- Operation objects are executed, and Tensor objects are evaluated.
- Session will also allocate memory to store the current values of variables.

```python
sess = tf.Session()
outs = sess.run(e)
print("outs = {}".format(outs))
sess.close()
```

```python
# can be written as follows
with tf.Session() as sess:
    outs = sess.run(e)

print("outs = {}".format(outs))
```

Feeding

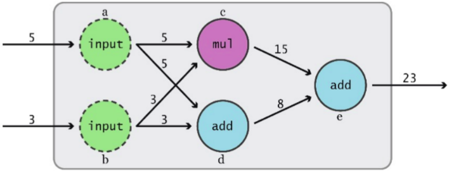
- A graph can be parameterized to accept external inputs via placeholders.
- To feed a placeholder, the input data is passed to the `session.run()`.
- Each key corresponds to a placeholder variable name, and the matching values are the data values.

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

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

Fetches

- To fetche a list of outputs of nodes.

```
with tf.Session() as sess:
    fetches = [a, b, c, d, e]
    outs = sess.run(fetches)

print("outs = {}".format(outs))
```



`Session.run()` vs. `Tensor.eval()`

- Two ways to evaluate part of graph: `Session.run` and `Tensor.eval`.

Session.run() vs. Tensor.eval()

- Two ways to evaluate part of graph: `Session.run` and `Tensor.eval`.
- The most important difference is that you can use `sess.run()` to fetch the values of many tensors in the same step.

```
t = tf.constant(42.0)
u = tf.constant(37.0)
tu = tf.mul(t, u)
ut = tf.mul(u, t)

with sess.as_default():
   tu.eval()  # runs one step
   ut.eval()  # runs one step
   sess.run([tu, ut])  # evaluates both tensors in a single step
```

Linear Regression in TensorFlow

Linear Regression

- We want to find weights $\mathbf{w}$ and a bias term $\mathtt{b}$.
- Assume our target value is a linear combination of some input vector $\mathbf{x}$: $\hat{y} = \mathbf{w}^{\mathrm{T}}\mathbf{x} + \mathtt{b}$.

Linear Regression

- We want to find weights $\mathbf{w}$ and a bias term b.
- Assume our target value is a linear combination of some input vector $\mathbf{x}$: $\hat{y} = \mathbf{w}^T\mathbf{x} + b$.
- Let's generate synthetic data.

```
import numpy as np
import tensorflow as tf

x_data = np.random.randn(2000, 3)
w_real = [0.3, 0.5, 0.1]
b_real = -0.2
y_data = np.matmul(w_real, x_data.T) + b_real
```

Linear Regression - Placeholders and Variables

- Create placeholders for our input and output data.
- Create variables for our weights and intercept.

```
# placeholders
x = tf.placeholder(tf.float32, shape=[None, 3])
y_true = tf.placeholder(tf.float32, shape=None)

# variables
w = tf.get_variable("weights", dtype=tf.float32, initializer=tf.constant([[0., 0., 0.]]))
b = tf.get_variable("bias", dtype=tf.float32, initializer=tf.constant(0.))
```

Linear Regression - Defining a Cost Function

- We need a good measure to evaluate the model's performance.
- Let's define MSE (mean squared error).

```
# the cost function
y_hat = tf.matmul(w, tf.transpose(x)) + b

cost = tf.reduce_mean(tf.square(y_true - y_hat))
```

Linear Regression - The Gradient Descent Optimizer

- Next, we need to minimize the cost function.
- Let's use the gradient descent.
- First create an optimizer by using the `GradientDescentOptimizer()` function.
- Then, create a train operation by calling the `optimizer.minimize()` to update our variables.

```
# optimizer
learning_rate = 0.5

optimizer = tf.train.GradientDescentOptimizer(learning_rate)
train = optimizer.minimize(cost)
```

Linear Regression - Execute It

- At the end, we need to initialize the variables and execute the train operation.

```
num_steps = 10
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for step in range(num_steps):
        sess.run(train, {x: x_data, y_true: y_data})
    print(sess.run([w, b, cost], {x: x_data, y_true: y_data}))
```

Logistic Regression in TensorFlow

Logistic Regression

- We want to find weights $\mathbf{w}$ and a bias term $\mathtt{b}$ in a logisitc regression model:

$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x}+\mathtt{b})}}$$

- Let's generate synthetic data.

```
import numpy as np
import tensorflow as tf

x_data = np.random.randn(2000, 3)
w_real = [0.3, 0.5, 0.1]
b_real = -0.2
y_data = np.matmul(w_real, x_data.T) + b_real
```

Logistic Regression - Placeholders and Variables

- Create placeholders for our input and output data.
- Create variables for our weights and intercept.

```
# placeholders
x = tf.placeholder(tf.float32, shape=[None, 3])
y_true = tf.placeholder(tf.float32, shape=None)

# variables
w = tf.get_variable("weights", dtype=tf.float32, initializer=tf.constant([[0., 0., 0.]]))
b = tf.get_variable("bias", dtype=tf.float32, initializer=tf.constant(0.))
```

Logistic Regression - Defining a Loss Function

- For the cost function, we use the cross-entropy model.

```
z = tf.matmul(w, tf.transpose(x)) + b
y_hat = tf.sigmoid(z)

cost = -y_true * tf.log(y_hat) - (1 - y_true) * tf.log(1 - y_hat)
cost = tf.reduce_mean(cost)
```

Logistic Regression - Defining a Loss Function

- For the cost function, we use the cross-entropy model.

```
z = tf.matmul(w, tf.transpose(x)) + b
y_hat = tf.sigmoid(z)

cost = -y_true * tf.log(y_hat) - (1 - y_true) * tf.log(1 - y_hat)
cost = tf.reduce_mean(cost)
```

- Alternatively, we can use a designated function by TensorFlow.

```
cost = tf.nn.sigmoid_cross_entropy_with_logits(labels=y_true, logits=y_hat)
cost = tf.reduce_mean(cost)
```

Logistic Regression - The Gradient Descent Optimizer

- Similar to linear regression.

```
learning_rate = 0.5

optimizer = tf.train.GradientDescentOptimizer(learning_rate)
train = optimizer.minimize(cost)
```

Logistic Regression - Execute It

- Similar to linear regression.

```
num_steps = 10
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for step in range(num_steps):
        sess.run(train, {x: x_data, y_true: y_data})
    print(sess.run([w, b, cost], {x: x_data, y_true: y_data}))
```

Saving and Restoring Models

Saving Models

- Save a model's parameters in disk.
- Create a `Saver` node at the end of the construction phase.
- Then, in the execution phase, call its `save()` method whenever you want to save the model.

Saving Models

- Save a model's parameters in disk.
- Create a `Saver` node at the end of the construction phase.
- Then, in the execution phase, call its `save()` method whenever you want to save the model.

```
w = tf.Variable([[0, 0, 0]], dtype=tf.float32, name="weights")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
  sess.run(init)
  for step in range(num_steps):
    if step % 100 == 0: # checkpoint every 100 epochs
      save_path = saver.save(sess, "/tmp/my_model.ckpt")
    sess.run(train, {x: x_data, y_true: y_data})
  best_w = sess.run(w)
  save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

Restoring Models

- Create a `Saver` node at the end of the construction phase.
- Then, at the beginning of the execution phase call the `restore()` method of the `Saver` node.
  - Instead of initializing the variables using the `init` node.

```
with tf.Session() as sess:
  saver.restore(sess, "/tmp/my_model_final.ckpt")
  [...]
```

TensorBoard

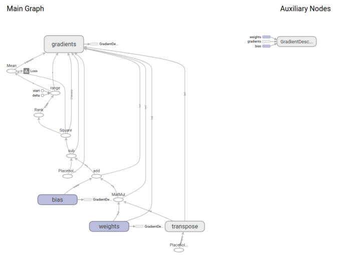
TensorBoard (1/3)

- TensorFlow provides a utility called TensorBoard.
- To visualize your model, you need to write the graph definition and some training stats to a log directory that TensorBoard will read from.
- Use a different log directory every time you run your program, or else TensorBoard will merge them.

TensorBoard (2/3)

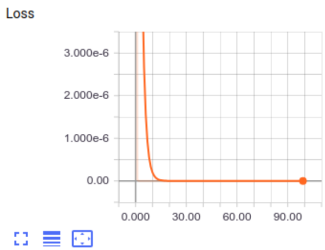
- Add the following code at the very end of the construction phase.
- The first line writes the `cost`.
- The second line creates a `FileWriter` that writes summaries of the graph.
- Start the TensorBoard web server (port 6006): `tensorboard --logdir .`

```
logdir = "."
mse_summary = tf.summary.scalar("MSE", cost)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
file_writer.close()
```



TensorBoard (3/3)

```
cost_summary = tf.summary.scalar("Loss", cost)
file_writer = tf.summary.FileWriter('.', tf.get_default_graph())
[...]
for step in range(num_steps):
  sess.run(train, {x: x_data, y_true: y_data})
  summary_str = cost_summary.eval(feed_dict={x: x_data, y_true: y_data})
  file_writer.add_summary(summary_str, step)
```



Keras

Keras

- Keras is a high-level API to build and train deep learning models.
- To get started, import `tf.keras` to your program.

```
import tensorflow as tf
from tensorflow.keras import layers
```

Keras Layers (1/2)

- In Keras, you assemble layers `tf.keras.layers` to build models.
- A model is (usually) a graph of layers.
- There are many types of layers, e.g., `Dense`, `Conv2D`, `RNN`, ...

Keras Layers (2/2)

- Common constructor parameters:

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
             bias_initializer=tf.keras.initializers.constant(2.0))
```

Keras Layers (2/2)

- Common constructor parameters:
  - `activation`: the activation function for the layer.

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
             bias_initializer=tf.keras.initializers.constant(2.0))
```

Keras Layers (2/2)

- Common constructor parameters:
  - `activation`: the activation function for the layer.
  - `kernel_initializer` and `bias_initializer`: the initialization schemes of the layer's weights.

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
             bias_initializer=tf.keras.initializers.constant(2.0))
```

Keras Layers (2/2)

- Common constructor parameters:
  - `activation`: the activation function for the layer.
  - `kernel_initializer` and `bias_initializer`: the initialization schemes of the layer's weights.
  - `kernel_regularizer` and `bias_regularizer`: the regularization schemes of the layer's weights, e.g., L1 or L2.

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
             bias_initializer=tf.keras.initializers.constant(2.0))
```

Keras Models

- There are two ways to build Keras models: sequential and functional.

Keras Models

- There are two ways to build Keras models: sequential and functional.
- The sequential API allows you to create models layer-by-layer.

Keras Models

- There are two ways to build Keras models: sequential and functional.
- The sequential API allows you to create models layer-by-layer.
- The functional API allows you to create models that have a lot more flexibility.
  - You can define models where layers connect to more than just the previous and next layers.

Keras Models - Sequential Models

- You can use `tf.keras.Sequential` to build a sequential model.

```
from tensorflow.keras import layers

model = tf.keras.Sequential()

model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```

Keras Models - Functional Models

- You can use `tf.keras.Model` to build a functional model.

```
from tensorflow.keras import layers

inputs = tf.keras.Input(shape=(32,))
x = layers.Dense(64, activation="relu")(inputs)
x = layers.Dense(64, activation="relu")(x)
predictions = layers.Dense(10, activation="softmax")(x)

model = tf.keras.Model(inputs=inputs, outputs=predictions)
```

Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(trainig_data, training_labels, epochs=10, batch_size=32)
```

Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.
  - `optimizer`: specifies the training procedure.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(trainig_data, training_labels, epochs=10, batch_size=32)
```

Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.
  - `optimizer`: specifies the training procedure.
  - `loss`: the cost function to minimize during optimization, e.g., mean square error (`mse`), `categorical_crossentropy`, and `binary_crossentropy`.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(trainig_data, training_labels, epochs=10, batch_size=32)
```

Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.
  - `optimizer`: specifies the training procedure.
  - `loss`: the cost function to minimize during optimization, e.g., mean square error (`mse`), `categorical_crossentropy`, and `binary_crossentropy`.
  - `metrics`: used to monitor training.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(trainig_data, training_labels, epochs=10, batch_size=32)
```

Training Keras Models

- Call the `compile` method to configure the learning process.
- `tf.keras.Model.compile` takes three important arguments.
  - `optimizer`: specifies the training procedure.
  - `loss`: the cost function to minimize during optimization, e.g., mean square error (`mse`), `categorical_crossentropy`, and `binary_crossentropy`.
  - `metrics`: used to monitor training.
- Call the `fit` method to fit the model the training data.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(trainig_data, training_labels, epochs=10, batch_size=32)
```

Evaluate and Predict

- `tf.keras.Model.evaluate`: evaluate the cost and metrics for the data provided.
- `tf.keras.Model.predict`: predict the output of the last layer for the data provided.

```
model.evaluate(test_data, test_labels, batch_size=32)

model.predict(test_data, batch_size=32)
```

Linear Regression in Keras

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers

x_data = np.random.randn(2000, 3)
w_real = [0.3, 0.5, 0.1]
b_real = -0.2
y_data = np.matmul(w_real, x_data.T) + b_real

model = tf.keras.Sequential([layers.Dense(1, activation="linear")])

model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001),
              loss="mse", metrics=["mae"])

model.fit(x_data, y_data, epochs=100, batch_size=32)

print(model.get_weights())
```

Logistic Regression in Keras

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers

x_data = ...
y_data = ...

model = tf.keras.Sequential([layers.Dense(1, activation="sigmoid")])

model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001),
              loss="binary_crossentropy", metrics=["accuracy"])

model.fit(x_data, y_data, epochs=100, batch_size=32)

print(model.get_weights())
```

Summary

Summary

- Dataflow graph
- Tensors: constants, variables, placeholders
- Session
- Save and restore models
- TensorBoard
- Keras

Reference

- Aurélien Géron, Hands-On Machine Learning (Ch. 9, 12)
- Some slides were derived from Chip Huyen's slides: `http://web.stanford.edu/class/cs20si`

Questions?