Large-Scale Distributed Data Processing
Dataflow Programming, TensorFlow

Silviu Maniu

April 14th, 2022

Table of contents

Dataflow Programming

Procedural/sequential programming – instructions, statements, tell the computer what steps to follow

- in distributed settings: hard to parallelize (how to split instructions between nodes?), how to deal with failures, programs for each machine
- message-passing architectures (MPI)

Dataflow Programming

Dataflow programming – old concept from processor design [Dennis and Misunas, 1974]

- programs are expressed at a higher level
- graphs (usually, DAGs) of operators and the dependency between them
- inputs and outputs connect operators
- operators can be black boxes (algorithms, learners, data transformation)

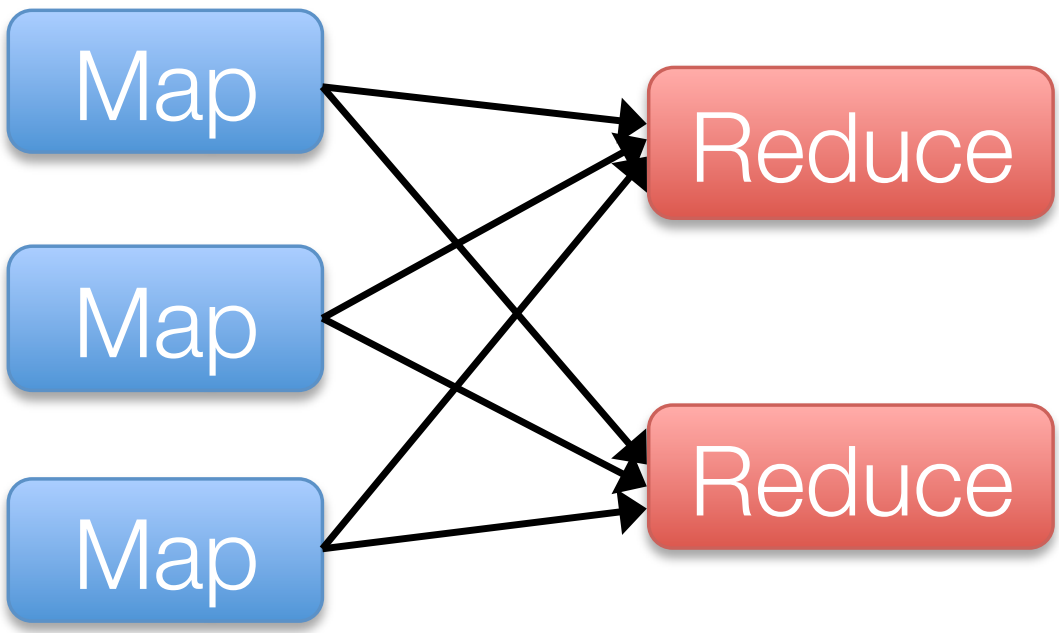
Why use dataflow?

- ease of programming (high-level)
- wide deployment (Spark, Pig, Hive)
- scalability to clusters

Dataflow Programming – MapReduce

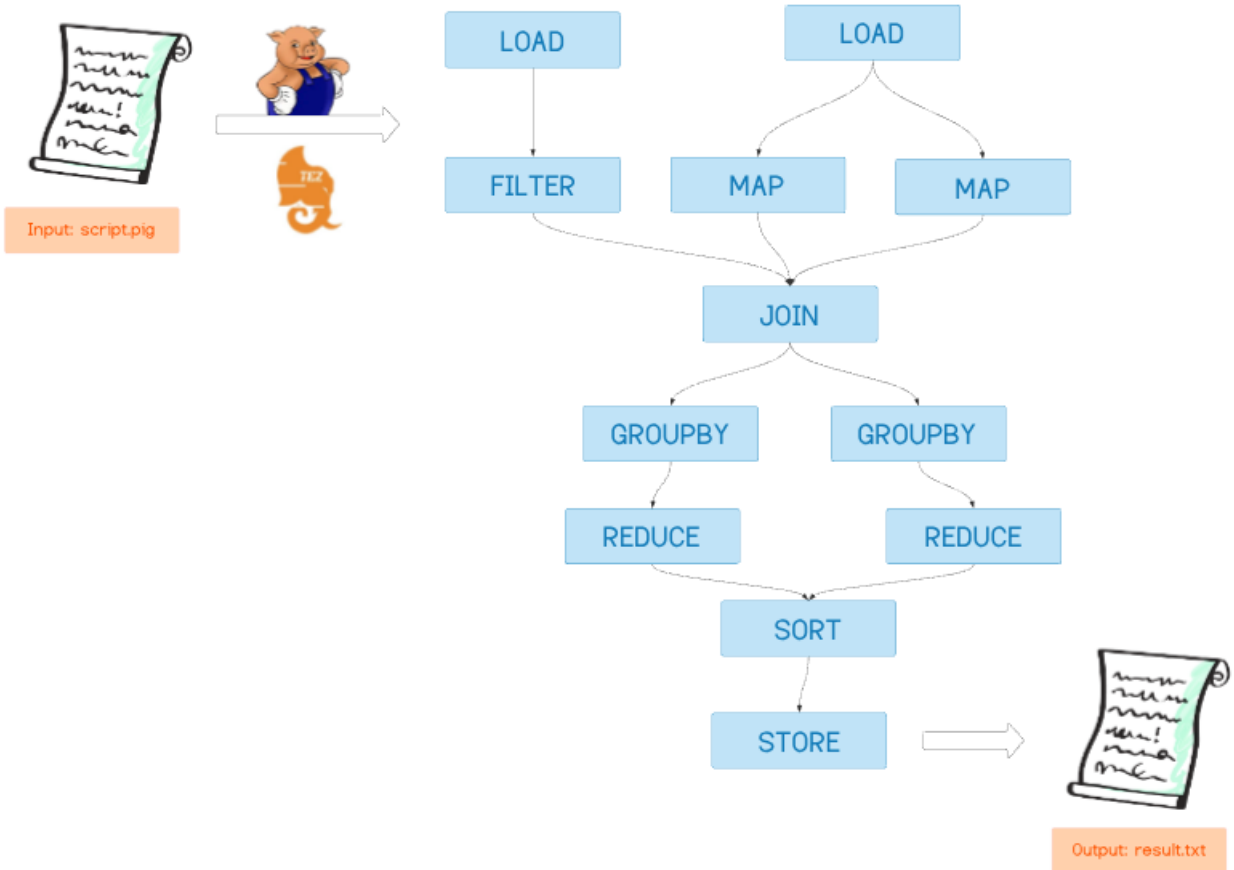
Example of DataFlow programming in distributed settings: MapReduce

- most systems rewrite algorithm in a MapReduce manner
- massive replication



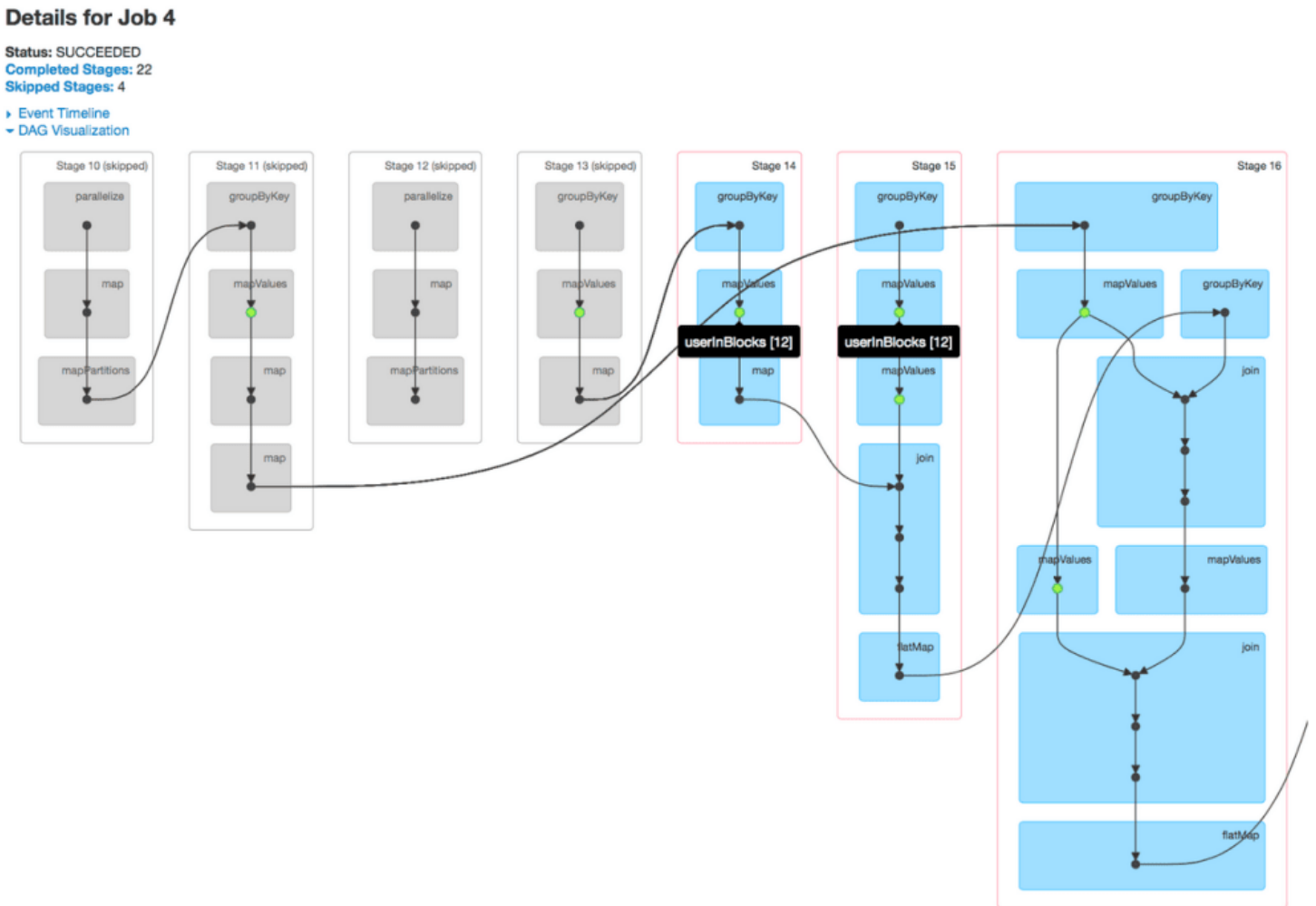
Dataflow Programming – Apache Pig

Transforms a SQL-like declarative command into a MapReduce DAG



Dataflow Programming – Spark

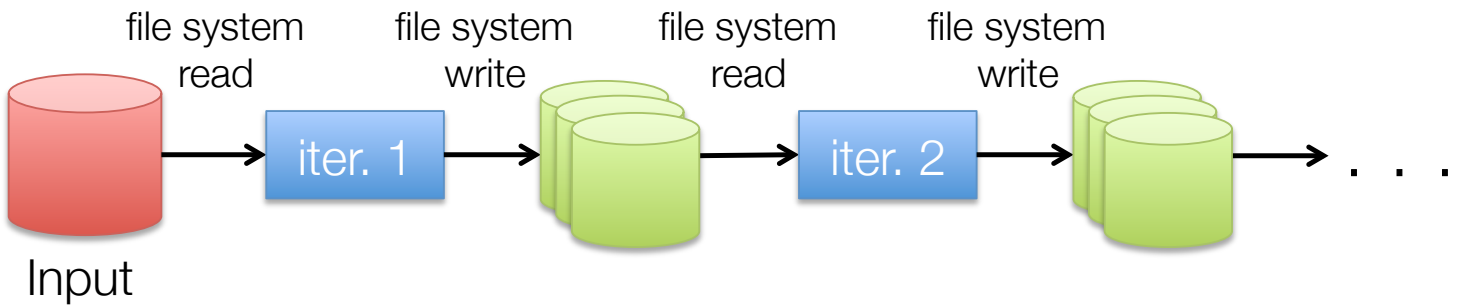
Operator on RDDs (via Scala, Python, ...) translated into a MapReduce DAG

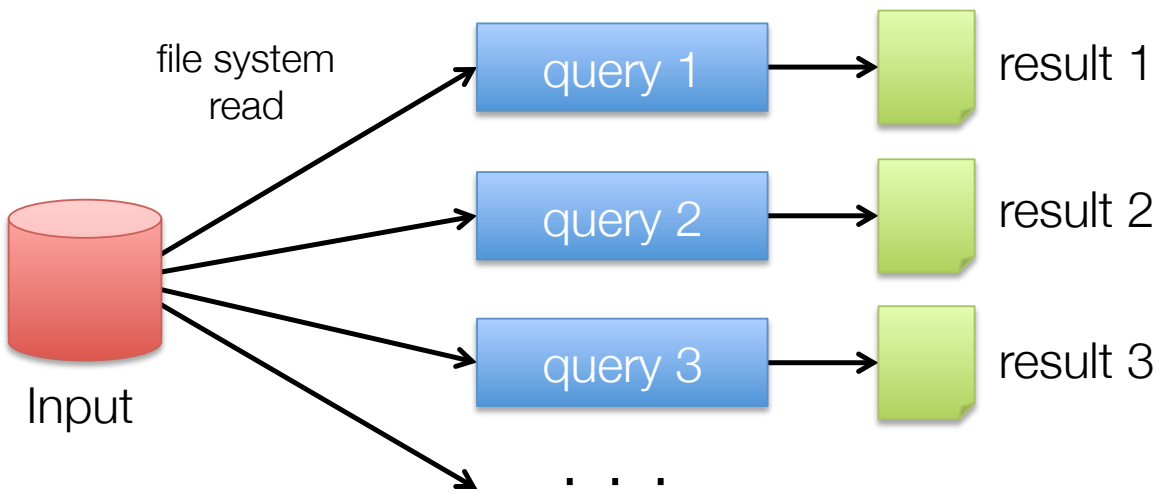


Dataflow and Iterative Programming

MapReduce: inefficient for multi-pass algorithms

- have to transfer state between stages
- no primitive for data sharing





Dataflow and Iterative Programming – PageRank

Repeatedly multiply sparse matrix and vector, most of computation (90%) is in I/O

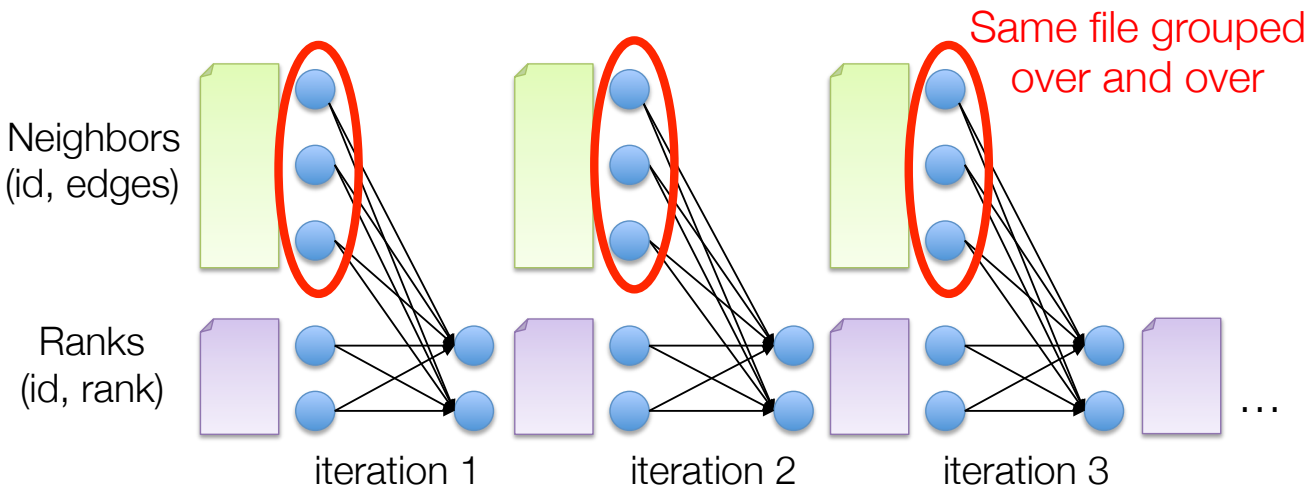


Table of contents

TensorFlow

TensorFlow – machine learning / deep learning library open-sourced from Google [Abadi et al., 2016]

- more precisely, interface for differential programming

Main functionality:

- primitives and functions which operate on tensors
- automatically computes derivatives

Dataflow-based conceptual model:

- specify relations between tensors, `.eval()` then creates the computation graph and executes

Tensors

Tensors: multi-linear maps from vector spaces to reals

$$f : V^* \times \cdots \times V^* \times V \times \cdots \times V \to \mathbb{R}$$

- scalars are tensors $f : \mathbb{R} \to \mathbb{R}, f(x) = c$
- vectors are tensors $f : \mathbb{R}^n \to \mathbb{R}, f(x_i) = v_i$
- matrices are tensors $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}, f(x_i, x_j) = A_{ij}$

Closest point of comparison: Python library Numpy

- tensors can be represented as a multidimensional array

The following 28 slides are taken from

`https://stanford.edu/~rezab/classes/cme323/S18/`

Simple Numpy Recap

```
In [23]: import numpy as np

In [24]: a = np.zeros((2,2)); b = np.ones((2,2))

In [25]: np.sum(b, axis=1)
Out[25]: array([ 2.,  2.])

In [26]: a.shape
Out[26]: (2, 2)

In [27]: np.reshape(a, (1,4))
Out[27]: array([[ 0.,  0.,  0.,  0.]])
```

Repeat in TensorFlow

More on `Session` soon

More on `.eval()` in a few slides

```
In [31]: import tensorflow as tf

In [32]: tf.InteractiveSession()

In [33]: a = tf.zeros((2,2)); b = tf.ones((2,2))

In [34]: tf.reduce_sum(b, reduction_indices=1).eval()
Out[34]: array([ 2.,  2.], dtype=float32)

In [35]: a.get_shape()
Out[35]: TensorShape([Dimension(2), Dimension(2)])

In [36]: tf.reshape(a, (1, 4)).eval()
Out[36]: array([[ 0.,  0.,  0.,  0.]], dtype=float32)
```

`TensorShape` behaves like a python tuple.

Numpy to TensorFlow Dictionary

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(a,reduction_indices=[1])`
`a.shape`	`a.get_shape()`
`np.reshape(a, (1,4))`	`tf.reshape(a, (1,4))`
`b * 5 + 1`	`b * 5 + 1`
`np.dot(a,b)`	`tf.matmul(a, b)`
`a[0,0], a[:,0], a[0,:]`	`a[0,0], a[:,0], a[0,:]`

TensorFlow requires explicit evaluation!

```
In [37]: a = np.zeros((2,2))

In [38]: ta = tf.zeros((2,2))

In [39]: print(a)
[[ 0.  0.]
 [ 0.  0.]]

In [40]: print(ta)
Tensor("zeros_1:0", shape=(2, 2), dtype=float32)

In [41]: print(ta.eval())
[[ 0.  0.]
 [ 0.  0.]]
```

*TensorFlow computations define a **computation graph** that has no numerical value until evaluated!*

TensorFlow Eager has begun to change this state of affairs...

TensorFlow Session Object (1)

- “A Session object encapsulates the environment in which Tensor objects are evaluated” - [TensorFlow Docs]

```
In [20]: a = tf.constant(5.0)

In [21]: b = tf.constant(6.0)

In [22]: c = a * b

In [23]: with tf.Session() as sess:
   ....:     print(sess.run(c))
   ....:     print(c.eval())
   ....:
30.0
30.0
```

`c.eval()` is just syntactic sugar for `sess.run(c)` in the currently active session!

TensorFlow Session Object (2)

- `tf.InteractiveSession()` is just convenient syntactic sugar for keeping a default session open in ipython.
- `sess.run(c)` is an example of a TensorFlow *Fetch*. Will say more on this soon.

Tensorflow Computation Graph

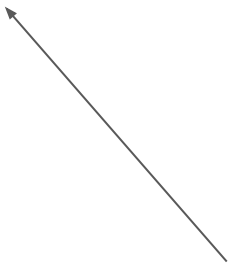
- “TensorFlow programs are usually structured into a construction phase, that assembles a graph, and an execution phase that uses a session to execute ops in the graph.” - TensorFlow docs
- All computations add nodes to global default graph (docs)

TensorFlow Variables (1)

- “When you train a model you use variables to hold and update parameters. Variables are in-memory buffers containing tensors” - TensorFlow Docs.
- All tensors we’ve used previously have been *constant* tensors, not variables.

TensorFlow Variables (2)

```
In [32]: W1 = tf.ones((2,2))

In [33]: W2 = tf.Variable(tf.zeros((2,2)), name="weights")

In [34]: with tf.Session() as sess:
             print(sess.run(W1))
             sess.run(tf.initialize_all_variables())
             print(sess.run(W2))
   ....:
[[ 1.  1.]
 [ 1.  1.]]
[[ 0.  0.]
 [ 0.  0.]]
```



Note the initialization step
`tf.initialize_all_variables()`

TensorFlow Variables (3)

- TensorFlow variables must be initialized before they have values! Contrast with constant tensors.

```
In [38]: W = tf.Variable(tf.zeros((2,2)), name="weights")

In [39]: R = tf.Variable(tf.random_normal((2,2)), name="random_weights")

In [40]: with tf.Session() as sess:
   ....:     sess.run(tf.initialize_all_variables())
   ....:     print(sess.run(W))
   ....:     print(sess.run(R))
   ....:
```

`Variable` objects can be initialized from constants or random values

Initializes all variables with specified values.

Updating Variable State

```
In [63]: state = tf.Variable(0, name="counter")

In [64]: new_value = tf.add(state, tf.constant(1))

In [65]: update = tf.assign(state, new_value)

In [66]: with tf.Session() as sess:
   ....:     sess.run(tf.initialize_all_variables())
   ....:     print(sess.run(state))
   ....:     for _ in range(3):
   ....:         sess.run(update)
   ....:         print(sess.run(state))
   ....:
0
1
2
3
```

Roughly `new_value = state + 1`

Roughly `state = new_value`

Roughly

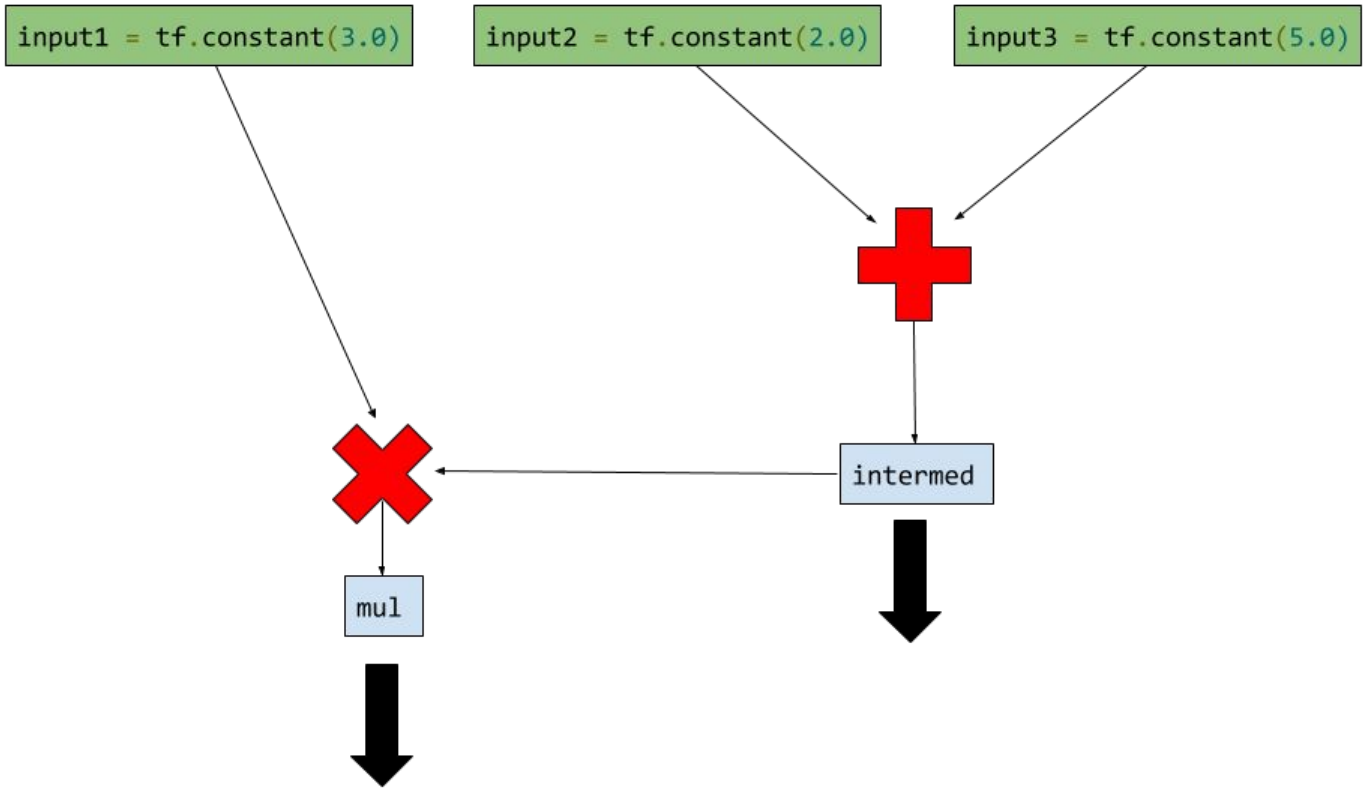
```
state = 0
print(state)
for _ in range(3):
  state = state + 1
  print(state)
```

Fetching Variable State (1)

```
In [82]: input1 = tf.constant(3.0)
In [83]: input2 = tf.constant(2.0)
In [84]: input3 = tf.constant(5.0)
In [85]: intermed = tf.add(input2, input3)
In [86]: mul = tf.mul(input1, intermed)
In [87]: with tf.Session() as sess:
   ....:       result = sess.run([mul, intermed])
   ....:       print(result)
   ....:
[21.0, 7.0]
```

Calling `sess.run(var)` on a `tf.Session()` object retrieves its value. Can retrieve multiple variables simultaneously with `sess.run([var1, var2])` (See *Fetches* in TF docs)

Fetching Variable State (2)



Inputting Data

- All previous examples have manually defined tensors. How can we input external data into TensorFlow?
- Simple solution: Import from Numpy:

```
In [93]: a = np.zeros((3,3))
In [94]: ta = tf.convert_to_tensor(a)
In [95]: with tf.Session() as sess:
   ....:     print(sess.run(ta))
   ....:
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Placeholders and Feed Dictionaries (1)

- Inputting data with `tf.convert_to_tensor()` is convenient, but doesn't scale.
- Use `tf.placeholder` variables (dummy nodes that provide entry points for data to computational graph).
- A `feed_dict` is a python dictionary mapping from `tf.placeholder` vars (or their names) to data (numpy arrays, lists, etc.).

Placeholders and Feed Dictionaries (2)

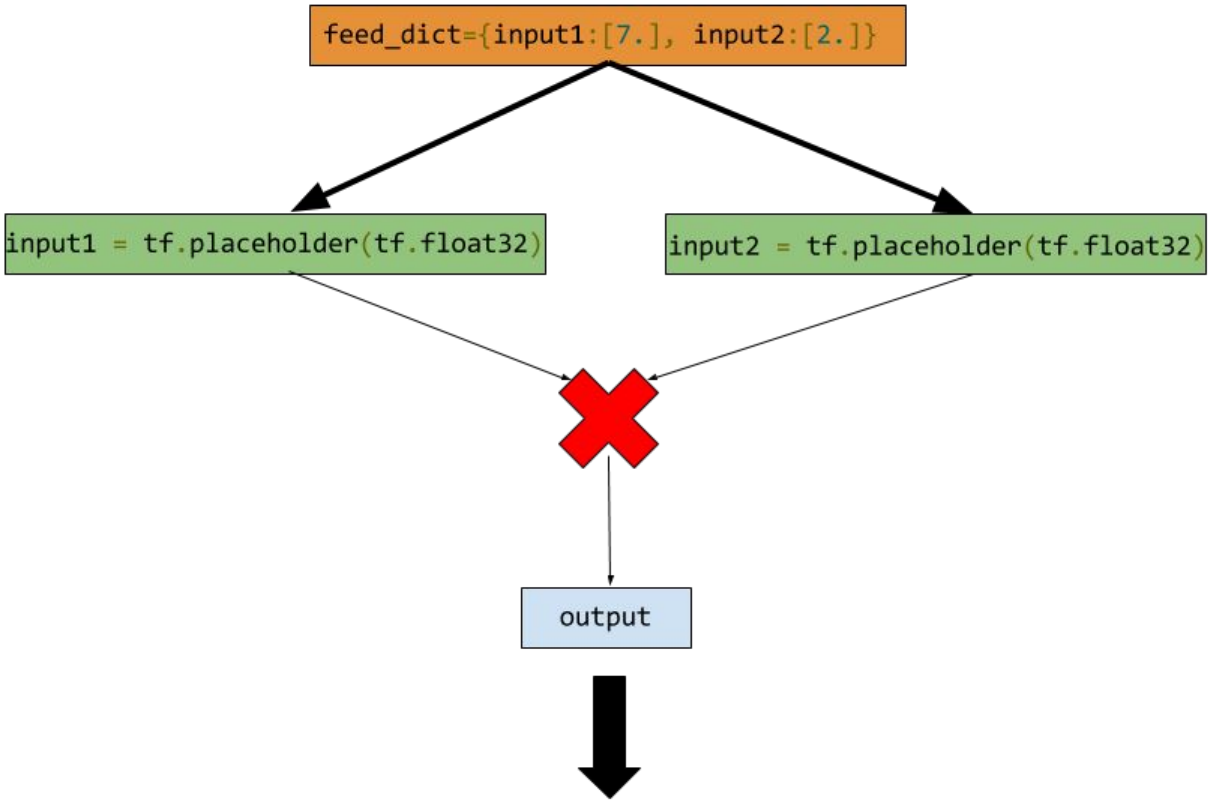
```
In [96]: input1 = tf.placeholder(tf.float32)

In [97]: input2 = tf.placeholder(tf.float32)

In [98]: output = tf.mul(input1, input2)

In [99]: with tf.Session() as sess:
   ....:       print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))
   ....:

[array([ 14.], dtype=float32)]
```

Define `tf.placeholder` *objects for data entry.*

Fetch value of output from computation graph.

Feed data into computation graph.

Placeholders and Feed Dictionaries (3)



Variable Scope (1)

- Complicated TensorFlow models can have hundreds of variables.
  - `tf.variable_scope()` provides simple name-spacing to avoid clashes.
  - `tf.get_variable()` creates/accesses variables from within a variable scope.

Variable Scope (2)

- Variable scope is a simple type of namespacing that adds prefixes to variable names within scope

```
with tf.variable_scope("foo"):
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])
assert v.name == "foo/bar/v:0"
```

Variable Scope (3)

- Variable scopes control variable (re)use

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
    tf.get_variable_scope().reuse_variables()
    v1 = tf.get_variable("v", [1])
assert v1 == v
```

Understanding get_variable (1)

- Behavior depends on whether variable reuse enabled
- **Case 1:** reuse set to false
  - Create and return new variable

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
assert v.name == "foo/v:0"
```

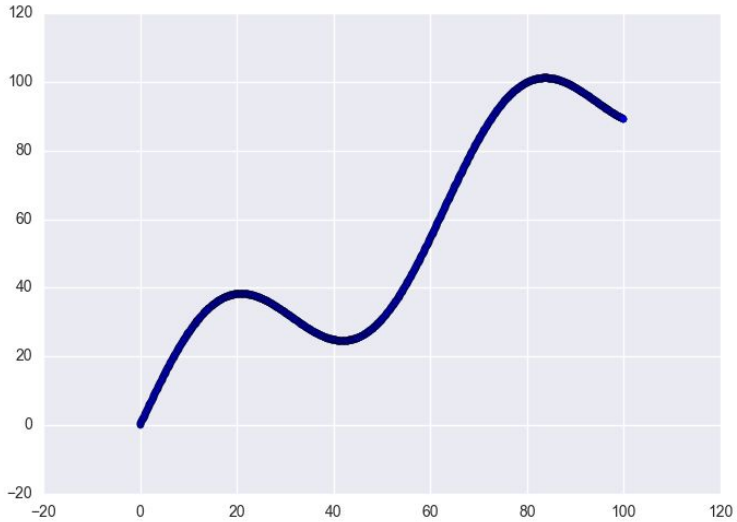
Understanding get_variable (2)

- **Case 2:** Variable reuse set to true
  - Search for existing variable with given name. Raise `ValueError` if none found.

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
with tf.variable_scope("foo", reuse=True):
    v1 = tf.get_variable("v", [1])
assert v1 == v
```

Ex: Linear Regression in TensorFlow (1)

```
import numpy as np
import seaborn

# Define input data
X_data = np.arange(100, step=.1)
y_data = X_data + 20 * np.sin(X_data/10)

# Plot input data
plt.scatter(X_data, y_data)
```



Ex: Linear Regression in TensorFlow (2)

```
# Define data size and batch size
n_samples = 1000
batch_size = 100

# Tensorflow is finicky about shapes, so resize
X_data = np.reshape(X_data, (n_samples,1))
y_data = np.reshape(y_data, (n_samples,1))

# Define placeholders for input
X = tf.placeholder(tf.float32, shape=(batch_size, 1))
y = tf.placeholder(tf.float32, shape=(batch_size, 1))
```

Ex: Linear Regression in TensorFlow (3)

```
# Define variables to be learned
with tf.variable_scope("linear-regression"):
  W = tf.get_variable("weights", (1, 1),
                      initializer=tf.random_normal_initializer())
  b = tf.get_variable("bias", (1,),
                      initializer=tf.constant_initializer(0.0))
  y_pred = tf.matmul(X, W) + b
  loss = tf.reduce_sum((y - y_pred)**2/n_samples)
```

Note `reuse=False` so these tensors are created anew

$$J(W,b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (Wx_i + b))^2$$

Ex: Linear Regression in TensorFlow (4)

```
# Sample code to run one step of gradient descent
In [136]: opt = tf.train.AdamOptimizer()

In [137]: opt_operation = opt.minimize(loss)

In [138]: with tf.Session() as sess:
   .....:     sess.run(tf.initialize_all_variables())
   .....:     sess.run([opt_operation], feed_dict={X: X_data, y: y_data})
   .....:
```

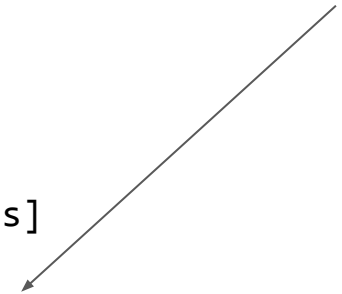
Note TensorFlow scope is not python scope! Python variable `loss` is still visible.

But how does this actually work under the hood? Will return to TensorFlow computation graphs and explain.

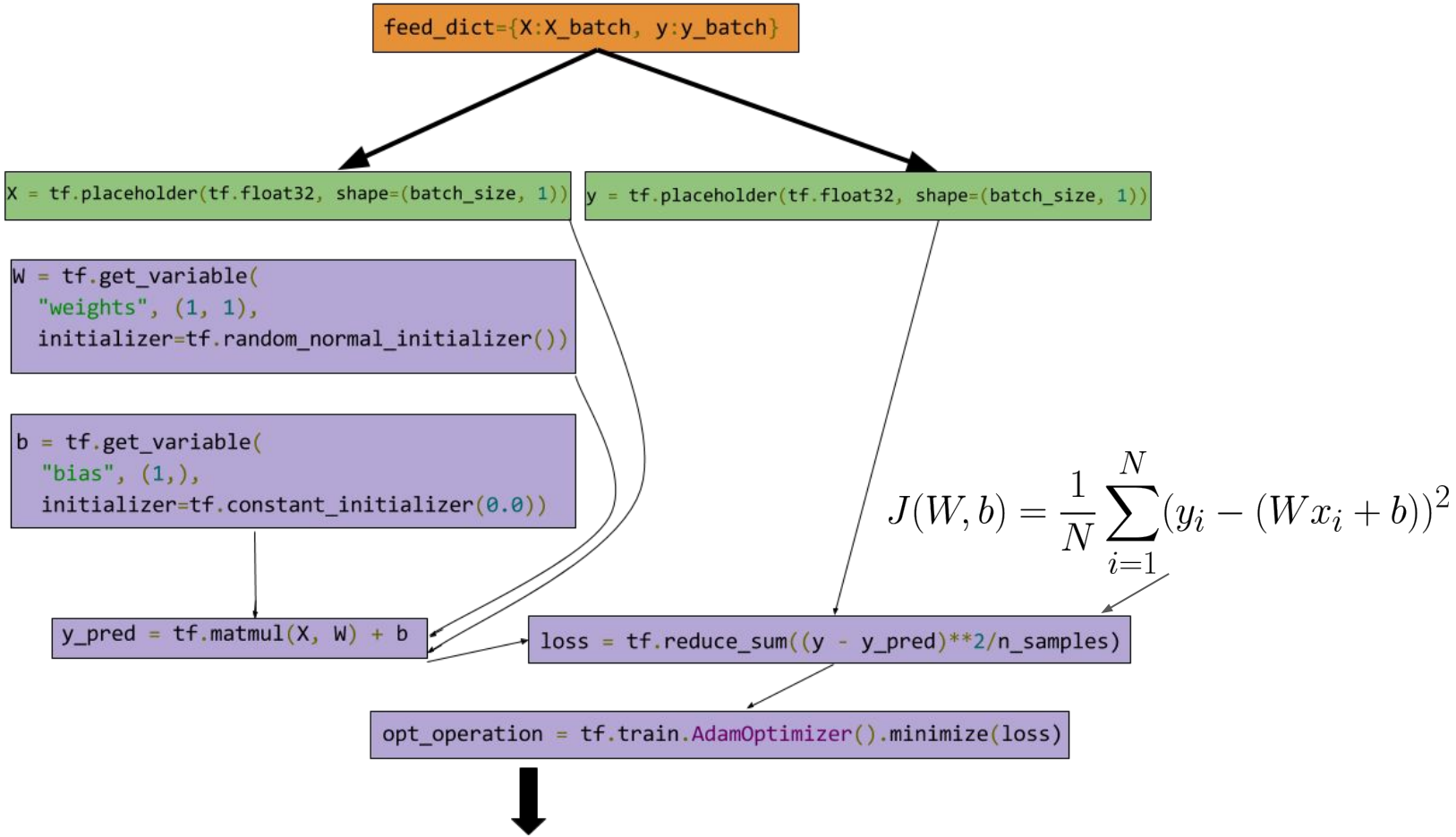
Ex: Linear Regression in TensorFlow (4)

```
# Sample code to run full gradient descent:
# Define optimizer operation
opt_operation = tf.train.AdamOptimizer().minimize(loss)

with tf.Session() as sess:
  # Initialize Variables in graph
  sess.run(tf.initialize_all_variables())
  # Gradient descent loop for 500 steps
  for _ in range(500):
    # Select random minibatch
    indices = np.random.choice(n_samples, batch_size)
    X_batch, y_batch = X_data[indices], y_data[indices]
    # Do gradient descent step
    _, loss_val = sess.run([opt_operation, loss], feed_dict={X: X_batch, y: y_batch})
```

Let's do a deeper. graphical dive into this operation



Ex: Linear Regression in TensorFlow (5)



Keras

Keras – high-level library to specify deep-learning models on top of TensorFlow

- can specify the layer of the neural networks, their relationship
- can scale to thousands of GPUs
- TensorFlow version 2 uses Keras as the main entry point (it was separate in TensorFlow 1)

Simplest model, `Sequential` stack of layers:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=10, activation='softmax'))

...
```

Keras

Provides methods for compiling the models

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd', metrics=['accuracy'])
```

Fitting and evaluating (can use Numpy arrays as inputs)

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
loss_and_metrics = model.evaluate(x_test, y_test,
                                  batch_size=128)
```

Predicting

```
classes = model.predict(x_test, batch_size=128)
```

For arbitrary models of more complex topology and input/output relationships, Keras provides the Functional API

Further Reading

For in-depth details the TensorFlow website is quite detailed and has several tutorials, both for basic and advanced uses:
`https://www.tensorflow.org/overview/`

Keras also has a quite detailed website:

- for engineers `https://keras.io/getting_started/intro_to_keras_for_engineers/`
- for researchers `https://keras.io/getting_started/intro_to_keras_for_researchers/`

References i

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016).
TensorFlow: A system for large-scale machine learning.
In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 265–283.

Dennis, J. B. and Misunas, D. P. (1974).
A preliminary architecture for a basic data-flow processor.
In *Proceedings of the Symposium on Computer Architecture (ISCA)*, page 126–132.