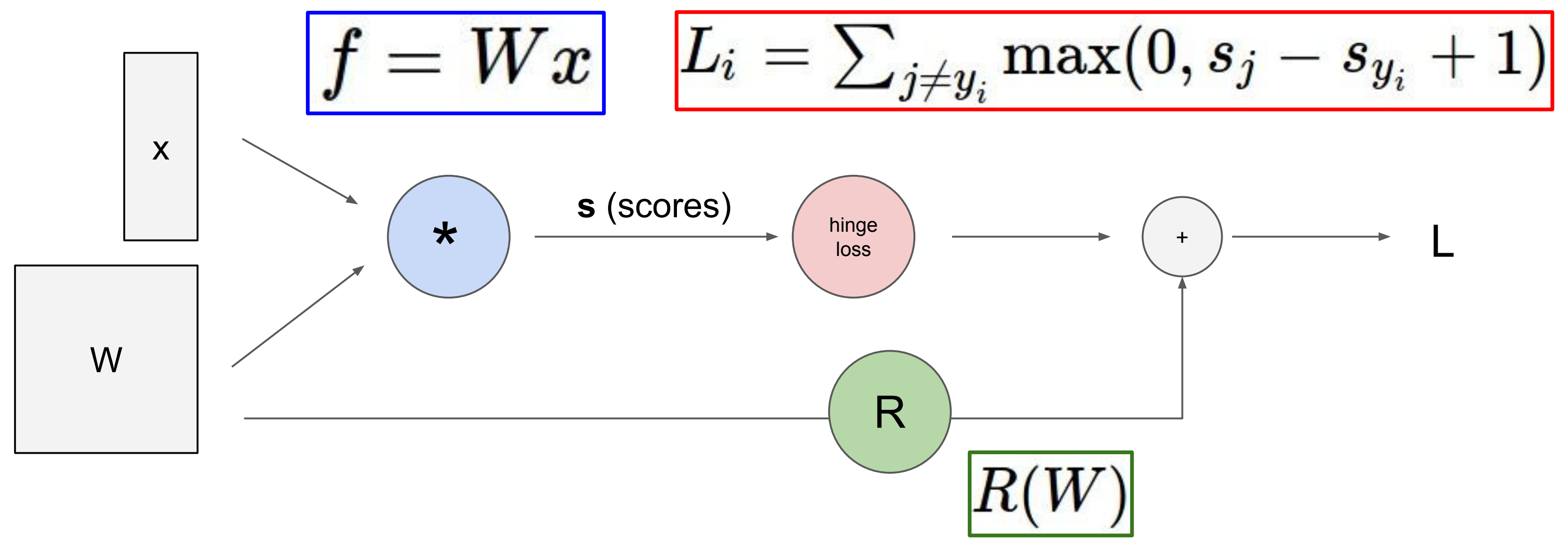
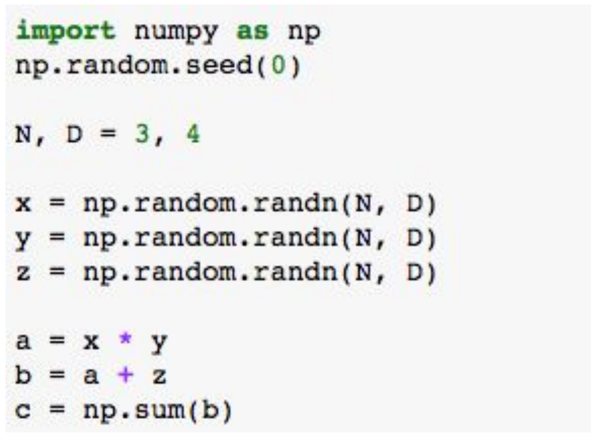
TensorFlow 0.0

Recall: Computational Graphs



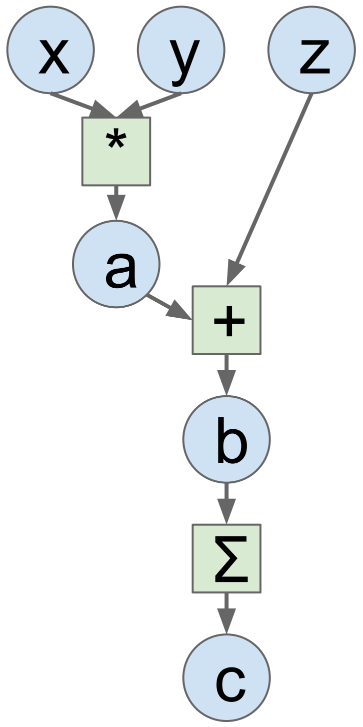
Computational Graphs

Numpy



```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```

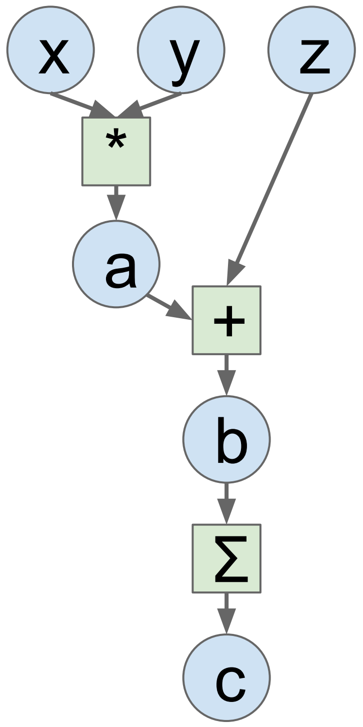
$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dl] & z \arrow[ddl] \\
& * \arrow[d] & & \\
& a \arrow[r] & + \arrow[d] & \\
& & b \arrow[d] & \\
& & \Sigma \arrow[d] & \\
& & c &
\end{tikzcd}$$



Computational Graphs

Numpy

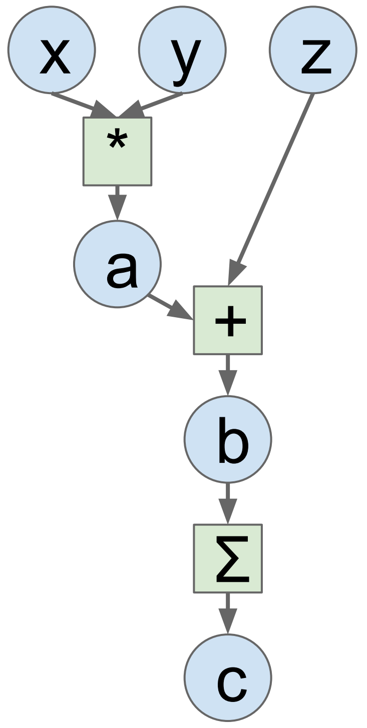
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Computational Graphs

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

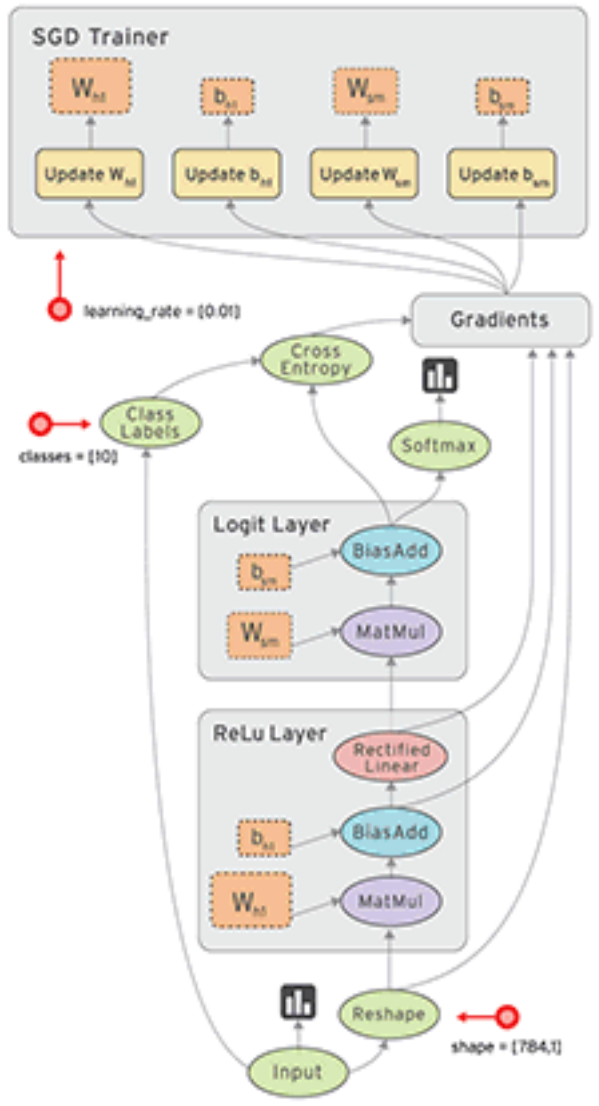


Computation graphs

Static Computation Graph

Step 1: Build computational graph describing our computation (including finding paths for backprop)

Step 2: Reuse the same graph on every iteration

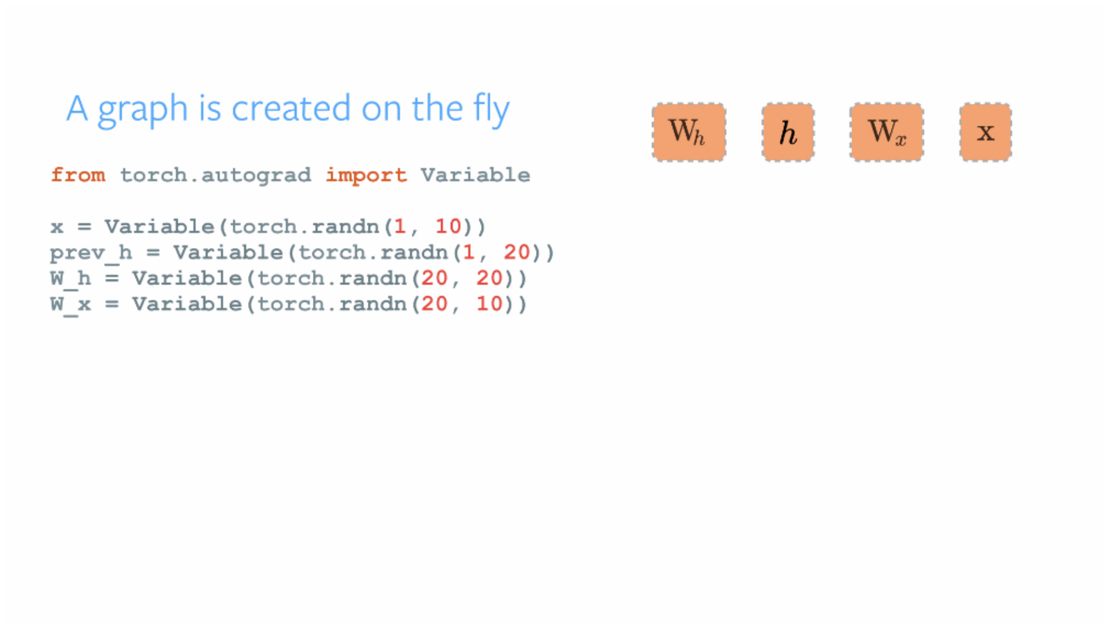


Dynamic Computation Graph

Building the graph and **computing** the graph happen at the same time.

Seems inefficient, especially if we are building the same graph over and over again...

For backprop: search for path between output and variables AND perform computation



```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

Flow (of data)

- **Parallelism.** By using explicit edges to represent dependencies between operations, it is easy for the system to identify operations that can execute in parallel.

- **Distributed execution.** By using explicit edges to represent the values that flow between operations, it is possible for TensorFlow to partition your program across multiple devices (CPUs, GPUs, and TPUs) attached to different machines. TensorFlow inserts the necessary communication and coordination between devices.

- **Compilation.** TensorFlow's XLA compiler can use the information in your dataflow graph to generate faster code, for example, by fusing together adjacent operations.

- **Portability.** The dataflow graph is a language-independent representation of the code in your model. You can build a dataflow graph in Python, store it in a SavedModel, and restore it in a C++ program for low-latency inference.

TensorFlow: Neural Net

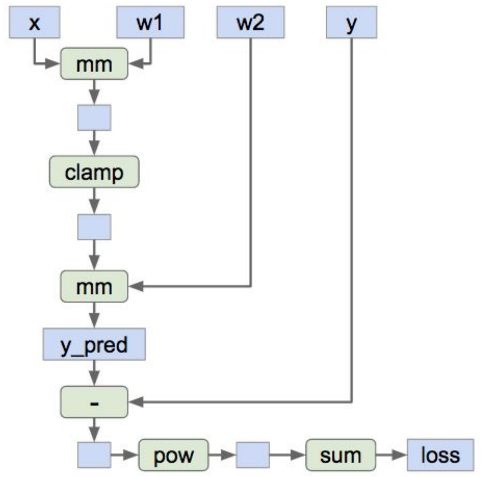
First **define** computational graph

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

Then **run** the graph many times

```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

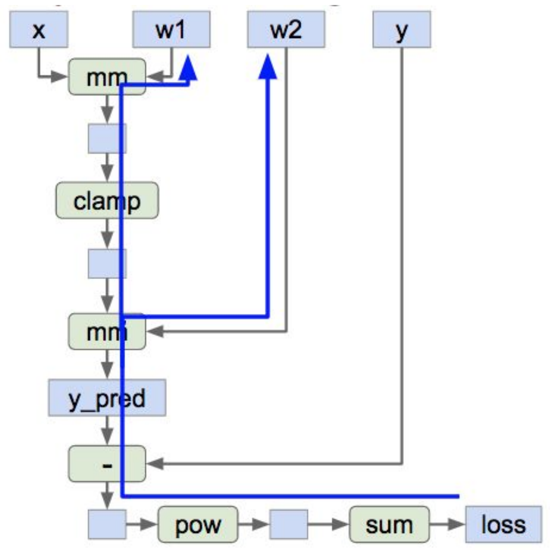
TensorFlow: Neural Net



Forward pass: compute prediction for y and loss. No computation - just building graph

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

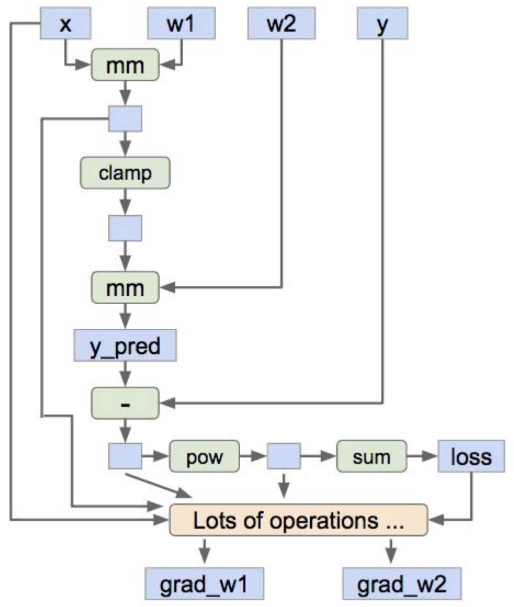
TensorFlow: Neural Net



Find paths between loss and w1, w2

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

TensorFlow: Neural Net



Add new operators to the graph which compute grad_w1 and grad_w2

```python
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Low Level TF API

- tf.Graph
  - Structure
    - Each operation is a node on the graph
  - Collections, e.g. tf.trainable_variables
- tf.Session
    - TensorFlow uses the `tf.Session` class to represent a connection between the client program---typically a Python program, although a similar interface is available in other languages---and the C++ runtime. A `tf.Session` object provides access to devices in the local machine, and remote devices using the distributed TensorFlow runtime. It also caches information about your `tf.Graph` so that you can efficiently run the same computation multiple times.
- tf.tensors
  - A tensor is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes.
  - When writing a TensorFlow program, the main object you manipulate and pass around is the tf.Tensor.
    - tf.Variable mutable tensor; these are used for internal parameters that will be updated during learning; A TensorFlow **variable** is the best way to represent shared, persistent state manipulated by your program.
    - Tf.placeholder immutable tensor: used for input, Inserts a placeholder for a tensor that will be always fed.
  - Shape can be defined at run-time
- The `tf.train.Saver` class provides methods to save and restore models. The `tf.saved_model.simple_save` function is an easy way to build

tf.data

- Reading data
    - Finally TF has one (nearly) API to read in data
- A `tf.data.Dataset` represents a sequence of elements, in which each element contains one or more `Tensor` objects. For example, in an image pipeline, an element might be a single training example, with a pair of tensors representing the image data and a label. There are two distinct ways to create a dataset:
    - Creating a **source** (e.g. `Dataset.from_tensor_slices()`) constructs a dataset from one or more `tf.Tensor` objects.
    - Applying a **transformation** (e.g. `Dataset.batch()`) constructs a dataset from one or more `tf.data.Dataset` objects.
- A `tf.data.Iterator` provides the main way to extract elements from a dataset. The operation returned by `Iterator.get_next()` yields the next element of a `Dataset` when executed, and typically acts as the interface between input pipeline code and your model. The simplest iterator is a "one-shot iterator", which is associated with a particular `Dataset` and iterates through it once. For more sophisticated uses, the `Iterator.initializer` operation enables you to reinitialize and parameterize an iterator with different datasets, so that you can, for example, iterate over training and validation data multiple times in the same program.