

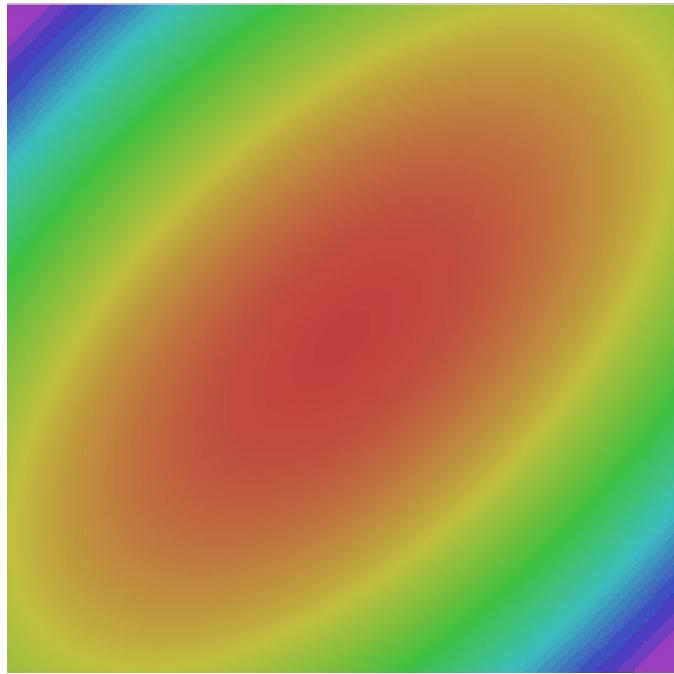
Lecture 8: Deep Learning Software

Administrative

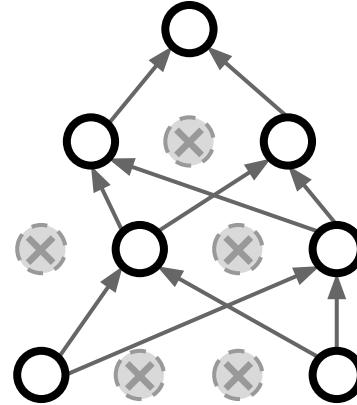
- Project proposals were due Tuesday
 - We are assigning TAs to projects, stay tuned
- We are grading A1
- A2 is due Thursday 5/4
 - Remember to **stop your instances** when not in use
 - Only use GPU instances for the **last notebook**

Last time

Optimization: SGD+Momentum,
Nesterov, RMSProp, Adam



Regularization: Dropout

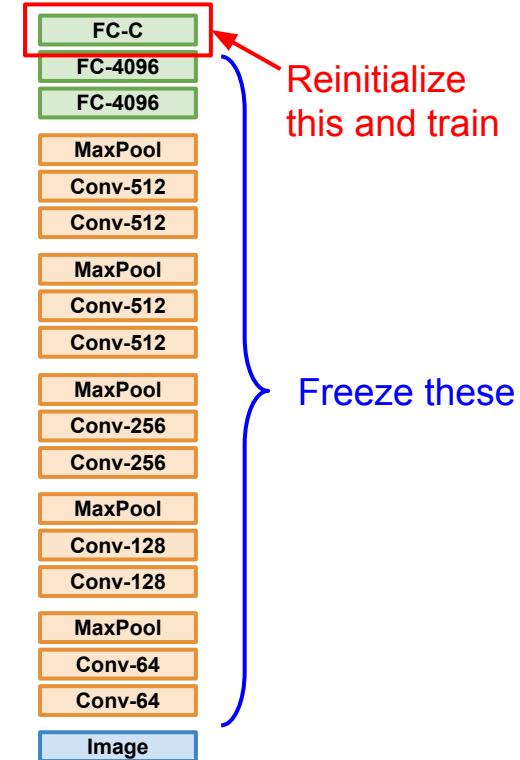


Regularization: Add noise, then
marginalize out

$$\text{Train} \quad y = f_W(x, z)$$

$$\text{Test} \quad y = f(x) = E_z[f(x, z)]$$

**Transfer
Learning**



Today

- CPU vs GPU
- Deep Learning Frameworks
 - Caffe / Caffe2
 - Theano / TensorFlow
 - Torch / PyTorch

CPU vs GPU

My computer

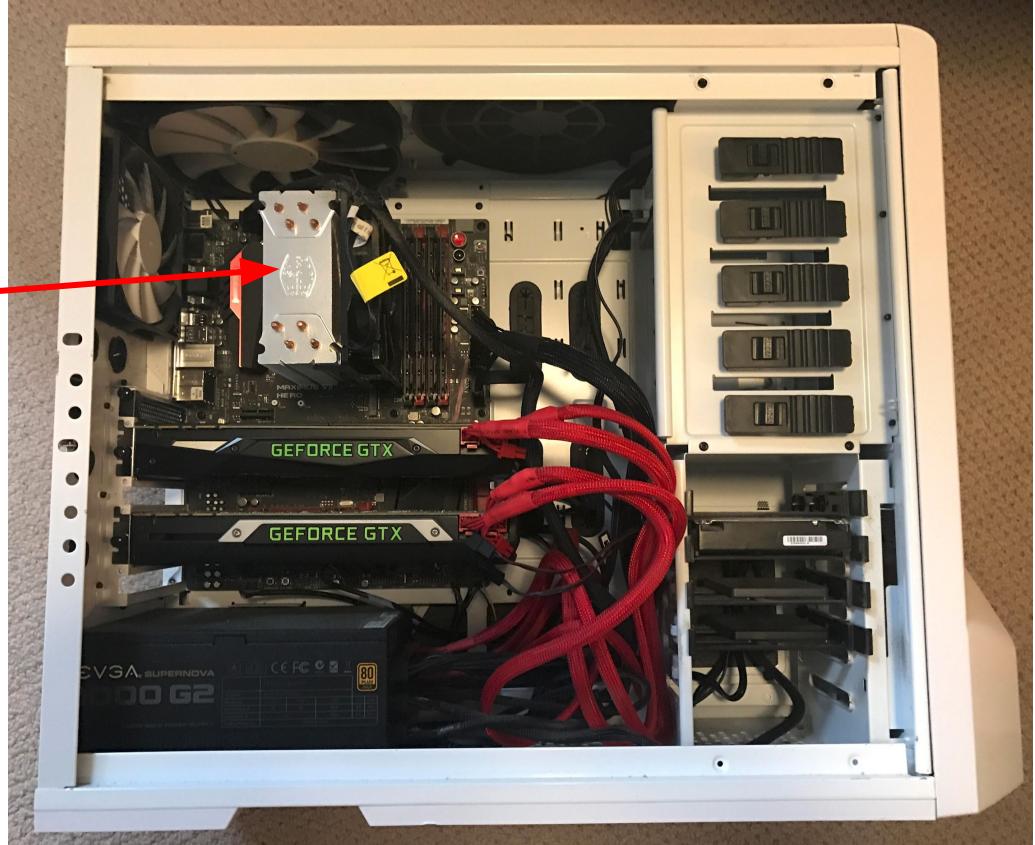


Spot the CPU!

(central processing unit)



[This image is licensed under CC-BY 2.0](#)



Spot the GPUs!

(graphics processing unit)



[This image is in the public domain](#)



NVIDIA vs AMD

NVIDIA

vs

AMD

CPU vs GPU

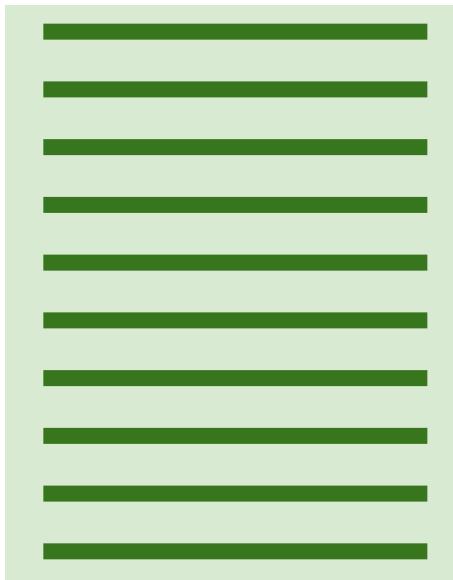
	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	\$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	\$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	\$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	\$399

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

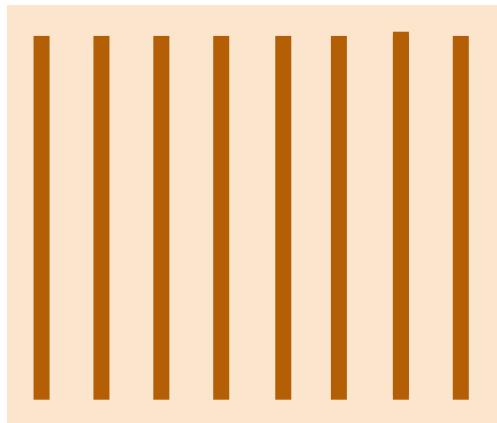
GPU: More cores, but each core is much slower and “dumber”; great for parallel tasks

Example: Matrix Multiplication

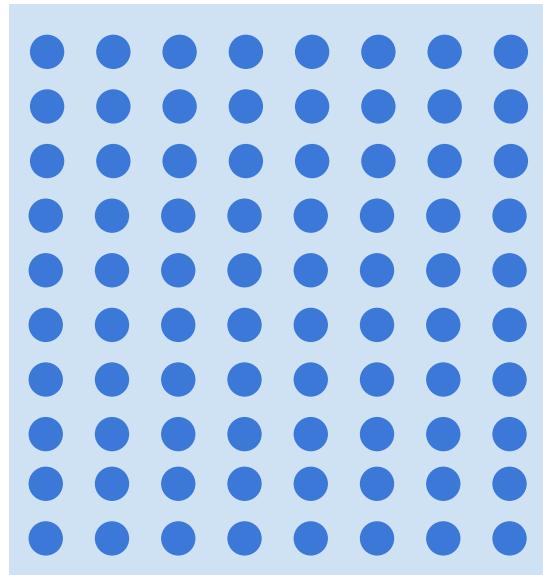
$A \times B$



$B \times C$



$A \times C$



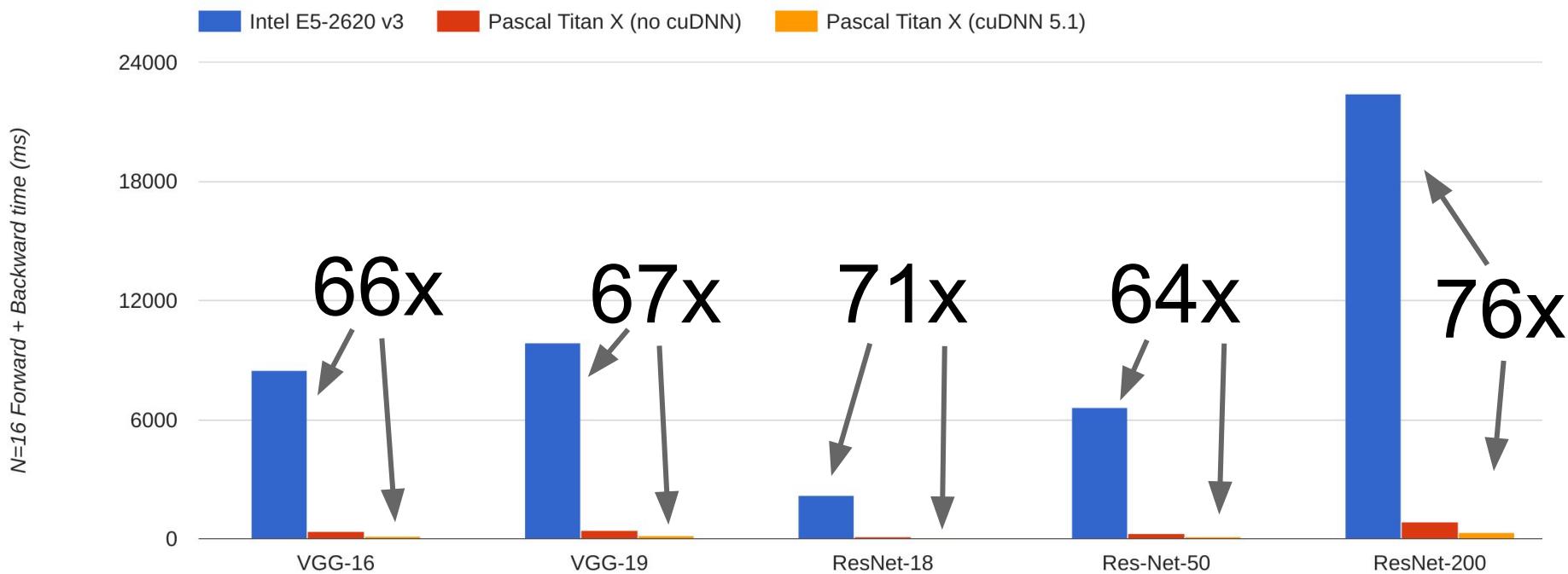
=

Programming GPUs

- CUDA (NVIDIA only)
 - Write C-like code that runs directly on the GPU
 - Higher-level APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
 - Similar to CUDA, but runs on anything
 - Usually slower :(
- Udacity: Intro to Parallel Programming
<https://www.udacity.com/course/cs344>
 - For deep learning just use existing libraries

CPU vs GPU in practice

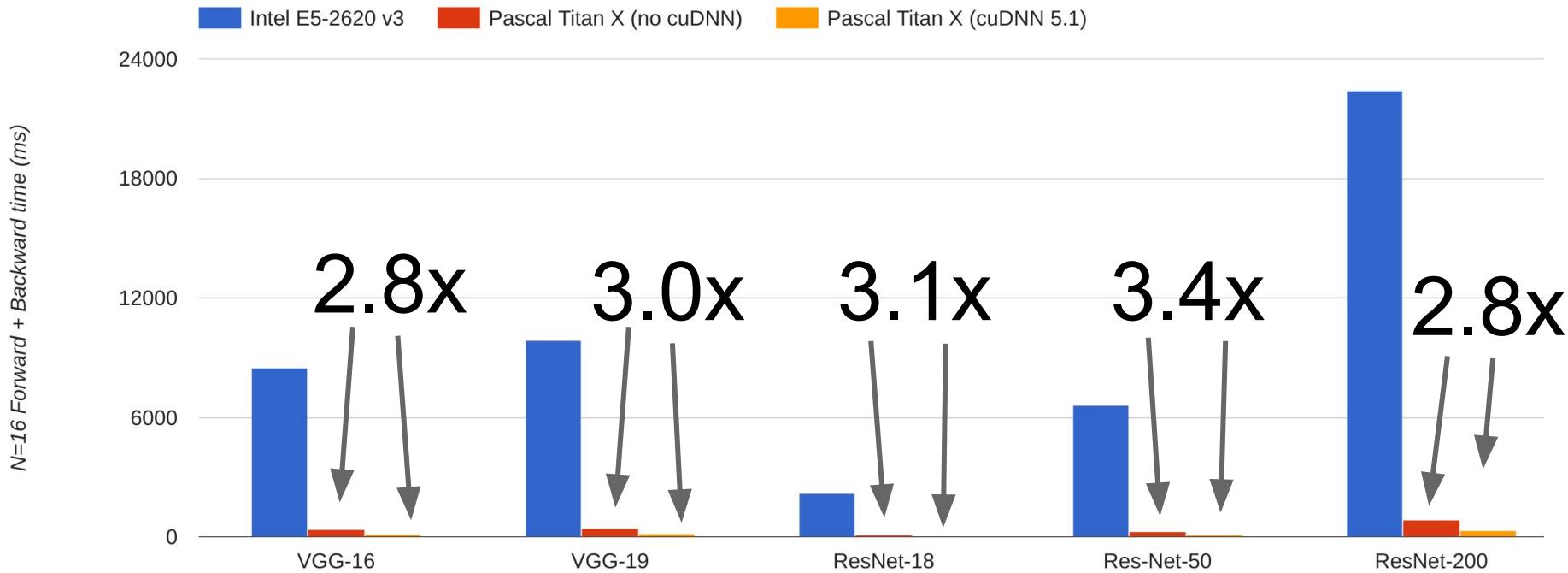
(CPU performance not well-optimized, a little unfair)



Data from <https://github.com/jcjohnson/cnn-benchmarks>

CPU vs GPU in practice

cuDNN much faster than
“unoptimized” CUDA

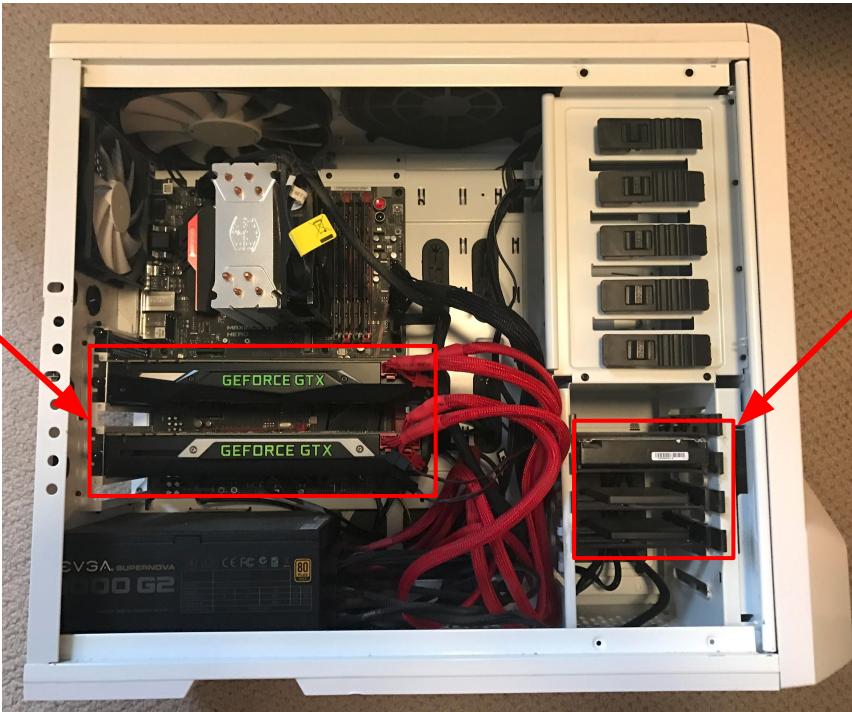


Data from <https://github.com/jcjohnson/cnn-benchmarks>

CPU / GPU Communication

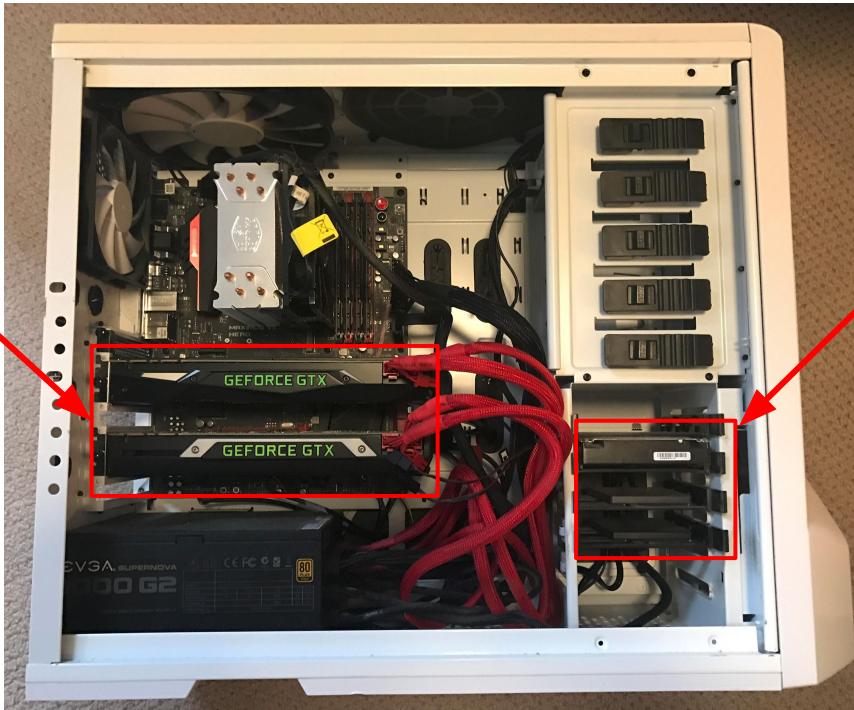
Model
is here

Data is here



CPU / GPU Communication

Model
is here



Data is here

If you aren't careful, training can bottleneck on reading data and transferring to GPU!

Solutions:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data

Deep Learning Frameworks

Last year ...

Caffe
(UC Berkeley)

Torch
(NYU / Facebook)

Theano  TensorFlow
(U Montreal) (Google)

This year ...

Caffe
(UC Berkeley)



Caffe2
(Facebook)

Torch
(NYU / Facebook)



PyTorch
(Facebook)

Theano
(U Montreal)



TensorFlow
(Google)

Paddle
(Baidu)

CNTK
(Microsoft)

MXNet
(Amazon)

Developed by U Washington, CMU, MIT,
Hong Kong U, etc but main framework of
choice at AWS

And others...

Today

A bit about these

Caffe
(UC Berkeley)

Caffe2
(Facebook)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Theano
(U Montreal)

TensorFlow
(Google)

Mostly these

Paddle
(Baidu)

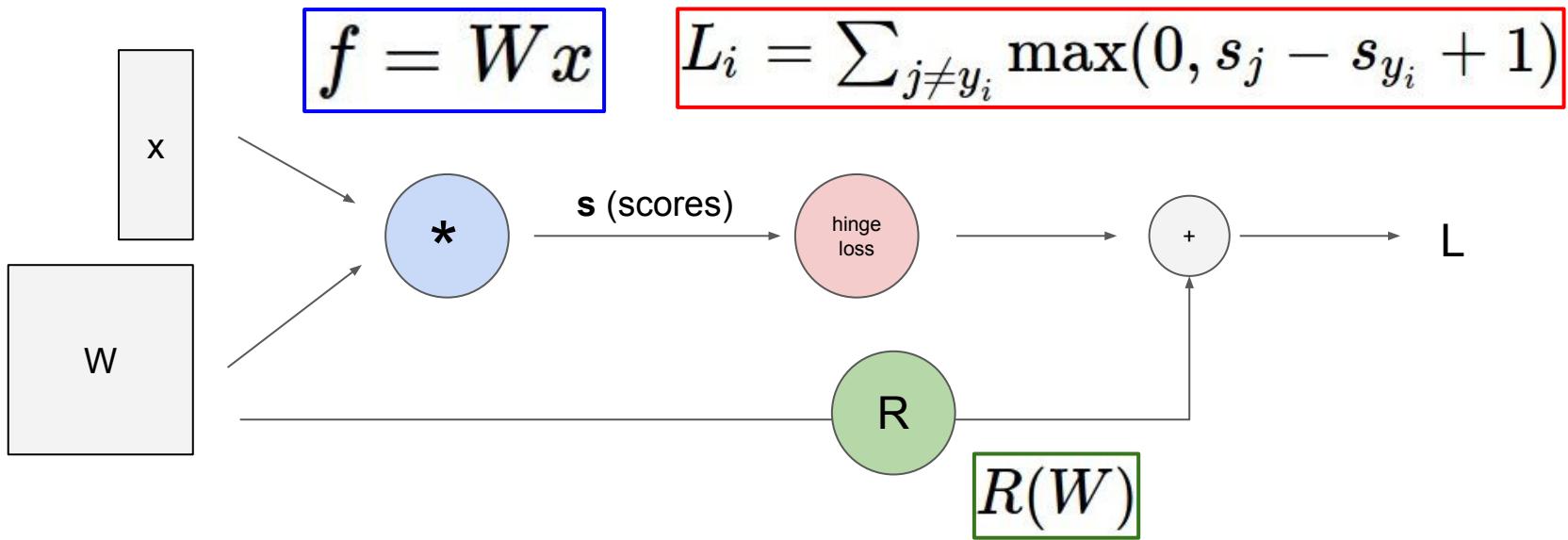
CNTK
(Microsoft)

MXNet
(Amazon)

Developed by U Washington, CMU, MIT,
Hong Kong U, etc but main framework of
choice at AWS

And others...

Recall: Computational Graphs



Recall: Computational Graphs

input image

weights

loss

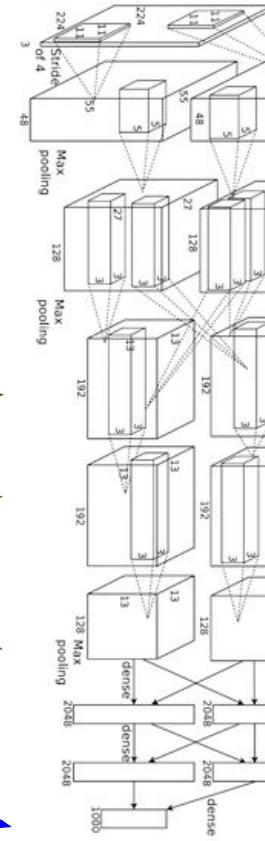


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Recall: Computational Graphs

input image

loss

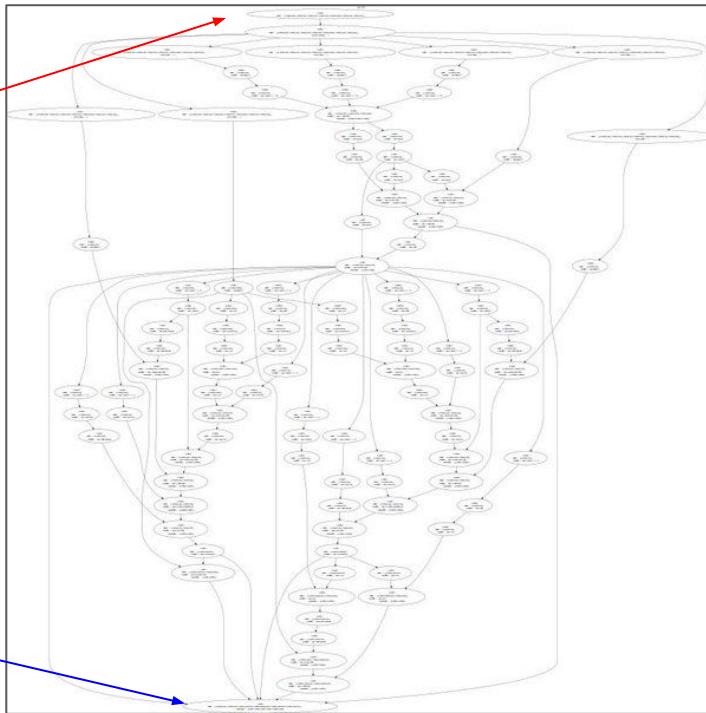


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

The point of deep learning frameworks

- (1) Easily build big computational graphs
- (2) Easily compute gradients in computational graphs
- (3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

Computational Graphs

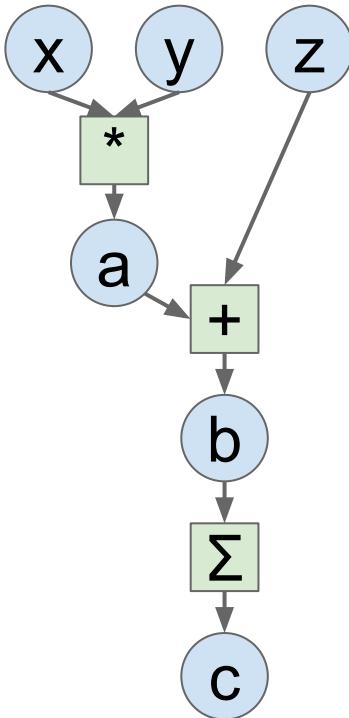
Numpy

```
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)
```



Computational Graphs

Numpy

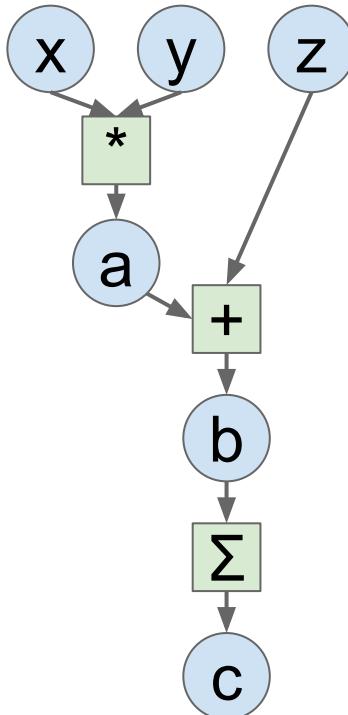
```
import numpy as np
np.random.seed(0)

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

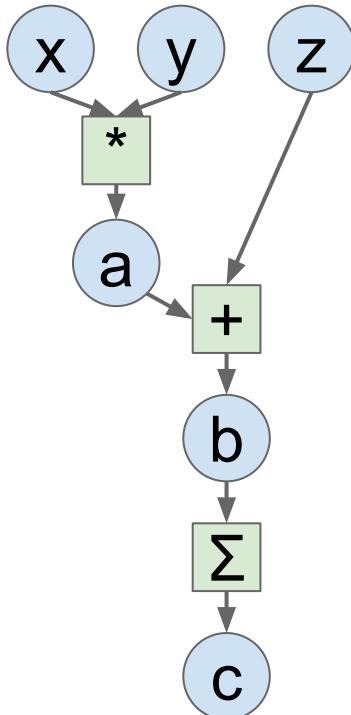
```
import numpy as np
np.random.seed(0)

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
```



Problems:

- Can't run on GPU
- Have to compute our own gradients

Computational Graphs

Numpy

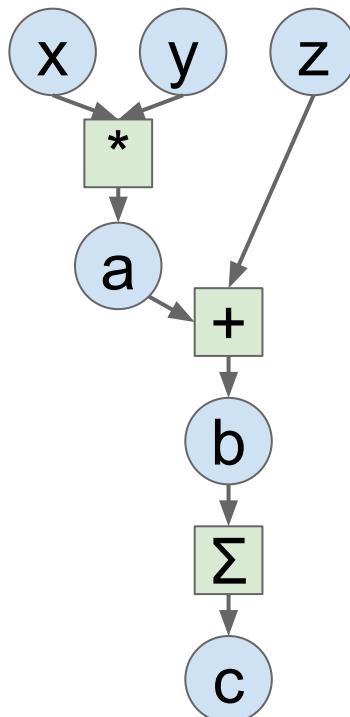
```
import numpy as np
np.random.seed(0)

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
```



TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

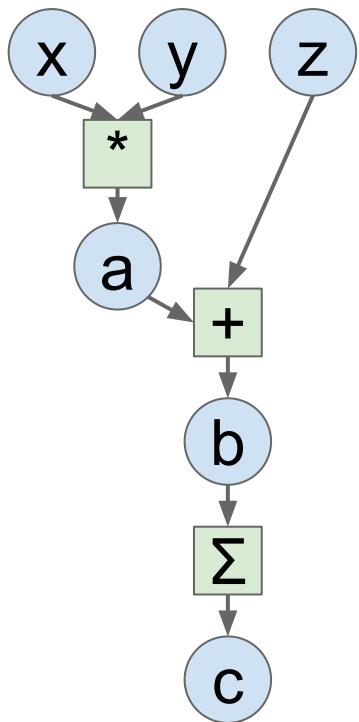
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                  feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Computational Graphs



Create forward
computational graph



TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

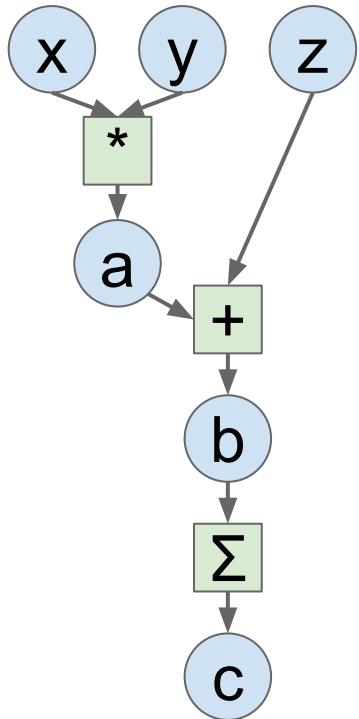
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                  feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Computational Graphs



Ask TensorFlow to
compute gradients

TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

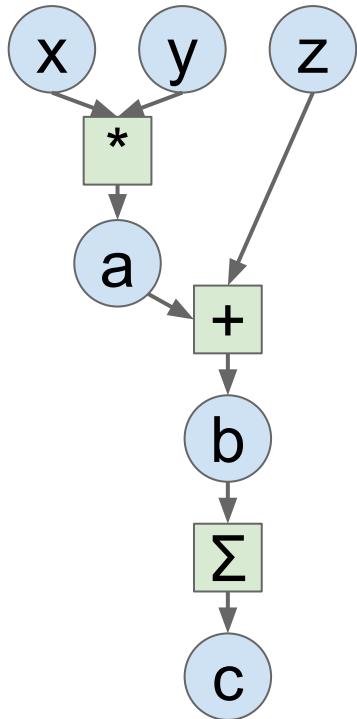
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                  feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Computational Graphs



Tell
TensorFlow
to run on **CPU**

TensorFlow

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

N, D = 3000, 4000

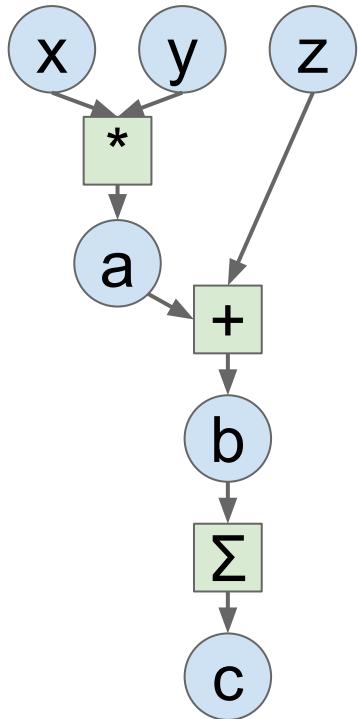
with tf.device('/cpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                  feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Computational Graphs



Tell
TensorFlow
to run on **GPU**

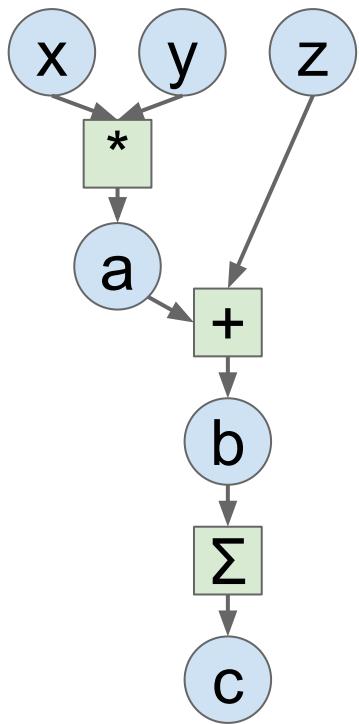
```
import numpy as np  
np.random.seed(0)  
import tensorflow as tf
```

```
N, D = 3000, 4000
```

```
with tf.device('/gpu:0'):  
    x = tf.placeholder(tf.float32)  
    y = tf.placeholder(tf.float32)  
    z = tf.placeholder(tf.float32)  
  
    a = x * y  
    b = a + z  
    c = tf.reduce_sum(b)  
  
grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])  
  
with tf.Session() as sess:  
    values = {  
        x: np.random.randn(N, D),  
        y: np.random.randn(N, D),  
        z: np.random.randn(N, D),  
    }  
    out = sess.run([c, grad_x, grad_y, grad_z],  
                 feed_dict=values)  
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

TensorFlow

Computational Graphs



PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

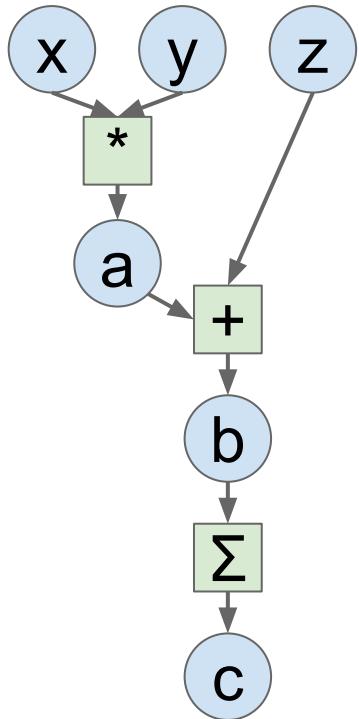
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

Computational Graphs



Define **Variables** to start building a computational graph

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

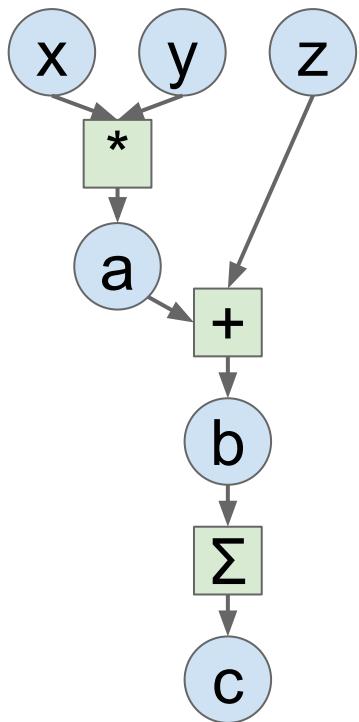
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

Computational Graphs



Forward pass
looks just like
numpy

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

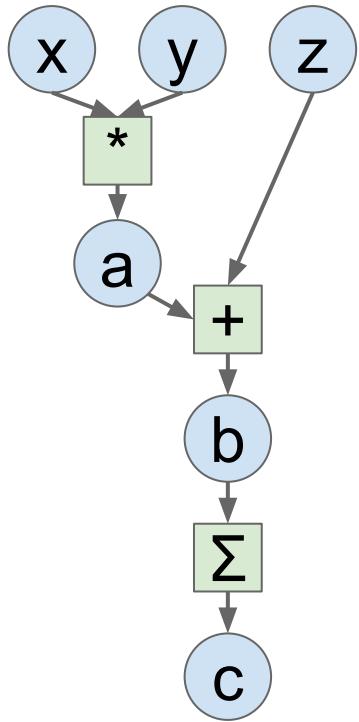
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

Computational Graphs



Calling `c.backward()` computes all gradients

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

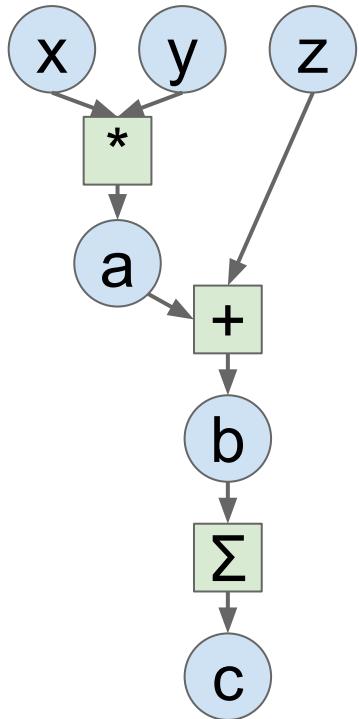
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

Computational Graphs



Run on GPU by casting to .cuda()

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

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
```

TensorFlow

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

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                  feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

TensorFlow

(more detail)

TensorFlow: Neural Net

Running example: Train
a two-layer ReLU
network on random data
with L2 loss

```
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

```
import numpy as np  
import tensorflow as tf
```

(Assume imports at the top of each snippet)

```
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

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

Create **placeholders** for
input x, weights w1 and
w2, and targets y

```
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

Forward pass: compute prediction for y and loss
(L2 distance between y and y_{pred})

No computation happens here - just building the 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

Tell TensorFlow to
compute loss of gradient
with respect to w1 and
w2.

Again no computation
here - just building the
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

Now done building our graph, so we enter a **session** so we can actually run the 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

Create numpy arrays
that will fill in the
placeholders above

```
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

Run the graph: feed in
the numpy arrays for x,
y, w1, and w2; get
numpy arrays for loss,
grad_w1, and grad_w2

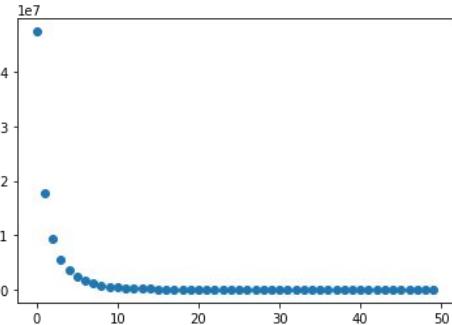
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

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



Train the network: Run the graph over and over, use gradient to update weights

```
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),}
    learning_rate = 1e-5
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2],
                      feed_dict=values)
        loss_val, grad_w1_val, grad_w2_val = out
        values[w1] -= learning_rate * grad_w1_val
        values[w2] -= learning_rate * grad_w2_val
```

TensorFlow: Neural Net

Problem: copying
weights between CPU /
GPU each step

Train the network: Run
the graph over and over,
use gradient to update
weights

```
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),}
    learning_rate = 1e-5
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2],
                      feed_dict=values)
        loss_val, grad_w1_val, grad_w2_val = out
        values[w1] -= learning_rate * grad_w1_val
        values[w2] -= learning_rate * grad_w2_val
```

TensorFlow: Neural Net

Change w1 and w2 from
placeholder (fed on
each call) to **Variable**
(persists in the graph
between calls)

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((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(tf.square(diff), axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

TensorFlow: Neural Net

Add **assign** operations
to update w1 and w2 as
part of the 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.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((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])
```

```
learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

TensorFlow: Neural Net

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

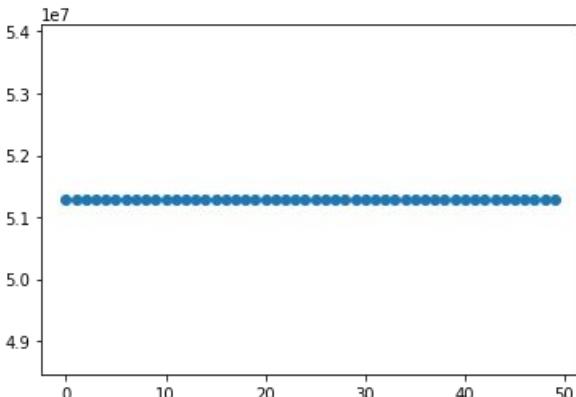
learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

Run graph once to
initialize w1 and w2

Run many times to train

TensorFlow: Neural Net



Problem: loss not going down! Assign calls not actually being executed!

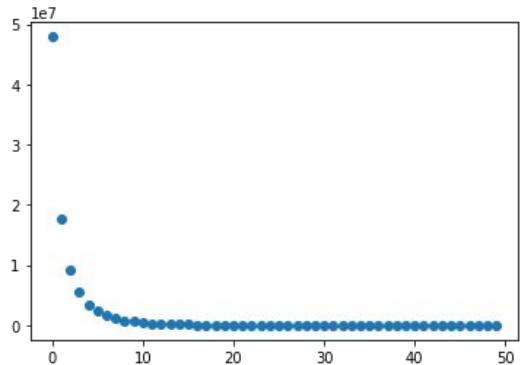
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

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])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
```

TensorFlow: Neural Net



Add dummy graph node
that depends on updates

Tell graph to compute
dummy node

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

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

TensorFlow: Optimizer

Can use an **optimizer** to
compute gradients and
update weights

Remember to execute the
output of the optimizer!

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((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 * diff, axis=1))
```

```
optimizer = tf.train.GradientDescentOptimizer(1e-5)
updates = optimizer.minimize(loss)
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

TensorFlow: LOSS

Use predefined
common losseses

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

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-3)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

TensorFlow: Layers

Use Xavier
initializer

tf.layers automatically
sets up weight and
(and bias) for us!

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

init = tf.contrib.layers.xavier_initializer()
h = tf.layers.dense(inputs=x, units=H,
                     activation=tf.nn.relu, kernel_initializer=init)
y_pred = tf.layers.dense(inputs=h, units=D,
                         kernel_initializer=init)

loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                             feed_dict=values)
```

Keras: High-Level Wrapper

Keras is a layer on top of
TensorFlow, makes common
things easy to do

(Also supports Theano
backend)

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))

optimizer = SGD(lr=1e-0)
model.compile(loss='mean_squared_error',
              optimizer=optimizer)

x = np.random.randn(N, D)
y = np.random.randn(N, D)
history = model.fit(x, y, nb_epoch=50,
                     batch_size=N, verbose=0)
```

Keras: High-Level Wrapper

Define model object as
a sequence of layers



```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))

optimizer = SGD(lr=1e-0)
model.compile(loss='mean_squared_error',
              optimizer=optimizer)

x = np.random.randn(N, D)
y = np.random.randn(N, D)
history = model.fit(x, y, nb_epoch=50,
                     batch_size=N, verbose=0)
```

Keras: High-Level Wrapper

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))

optimizer = SGD(lr=1e-0)
model.compile(loss='mean_squared_error',
              optimizer=optimizer)

x = np.random.randn(N, D)
y = np.random.randn(N, D)
history = model.fit(x, y, nb_epoch=50,
                     batch_size=N, verbose=0)
```

Define optimizer object



Keras: High-Level Wrapper

Build the model,
specify loss function



```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))

optimizer = SGD(lr=1e-0)
model.compile(loss='mean_squared_error',
              optimizer=optimizer)

x = np.random.randn(N, D)
y = np.random.randn(N, D)
history = model.fit(x, y, nb_epoch=50,
                     batch_size=N, verbose=0)
```

Keras: High-Level Wrapper

Train the model
with a single line!

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))

optimizer = SGD(lr=1e-0)
model.compile(loss='mean_squared_error',
              optimizer=optimizer)

x = np.random.randn(N, D)
y = np.random.randn(N, D)
history = model.fit(x, y, nb_epoch=50,
                     batch_size=N, verbose=0)
```

TensorFlow: Other High-Level Wrappers

Keras (<https://keras.io/>)

TFLearn (<http://tflearn.org/>)

TensorLayer (<http://tensorlayer.readthedocs.io/en/latest/>)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

TF-Slim (<https://github.com/tensorflow/models/tree/master/inception/inception/slim>)

tf.contrib.learn (https://www.tensorflow.org/get_started/tflearn)

Pretty Tensor (<https://github.com/google/prettytensor>)

TensorFlow: Other High-Level Wrappers

Keras (<https://keras.io/>)

TFLearn (<http://tflearn.org/>)

TensorLayer (<http://tensorlayer.readthedocs.io/en/latest/>)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

TF-Slim (<https://github.com/tensorflow/models/tree/master/inception/inception/slim>)

tf.contrib.learn (https://www.tensorflow.org/get_started/tflearn)

Pretty Tensor (<https://github.com/google/prettytensor>)

Ships with TensorFlow



TensorFlow: Other High-Level Wrappers

Keras (<https://keras.io/>)

TFLearn (<http://tflearn.org/>)

TensorLayer (<http://tensorlayer.readthedocs.io/en/latest/>)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

TF-Slim (<https://github.com/tensorflow/models/tree/master/inception/inception/slim>)

tf.contrib.learn (https://www.tensorflow.org/get_started/tflearn)

Pretty Tensor (<https://github.com/google/prettytensor>)

Ships with TensorFlow



From Google



TensorFlow: Other High-Level Wrappers

Keras (<https://keras.io/>)

TFLearn (<http://tflearn.org/>)

TensorLayer (<http://tensorlayer.readthedocs.io/en/latest/>)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

TF-Slim (<https://github.com/tensorflow/models/tree/master/inception/inception/slim>)

tf.contrib.learn (https://www.tensorflow.org/get_started/tflearn)

Pretty Tensor (<https://github.com/google/prettytensor>)

Sonnet (<https://github.com/deepmind/sonnet>)

Ships with TensorFlow



From Google

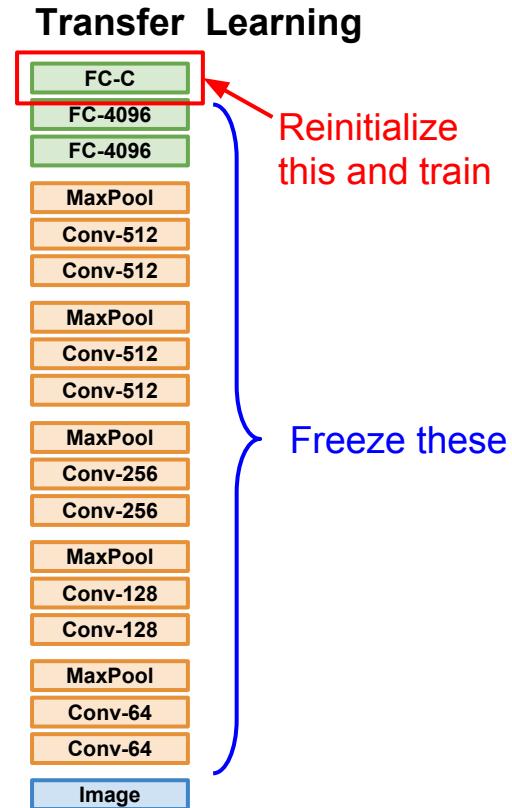
From DeepMind



TensorFlow: Pretrained Models

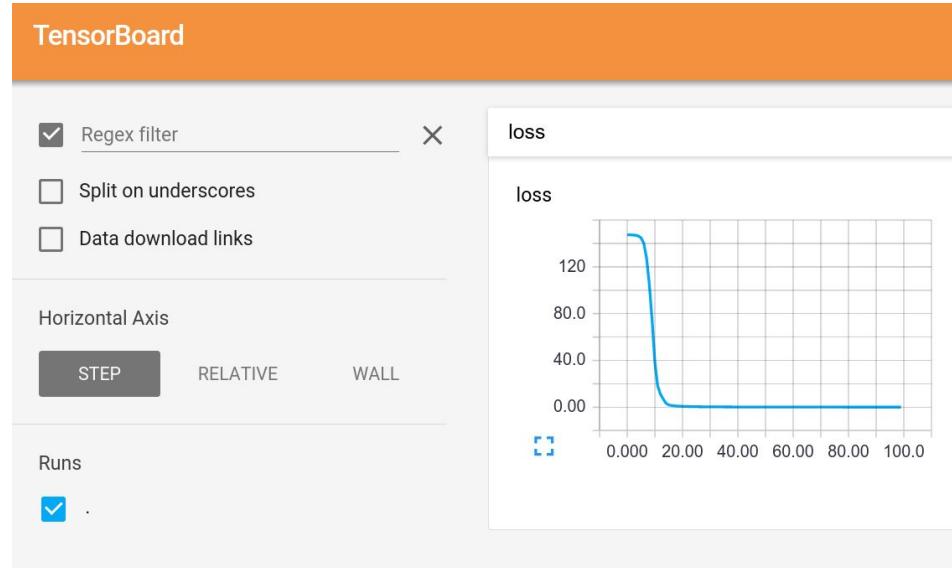
TF-Slim: (<https://github.com/tensorflow/models/tree/master/slim/nets>)

Keras: (<https://github.com/fchollet/deep-learning-models>)

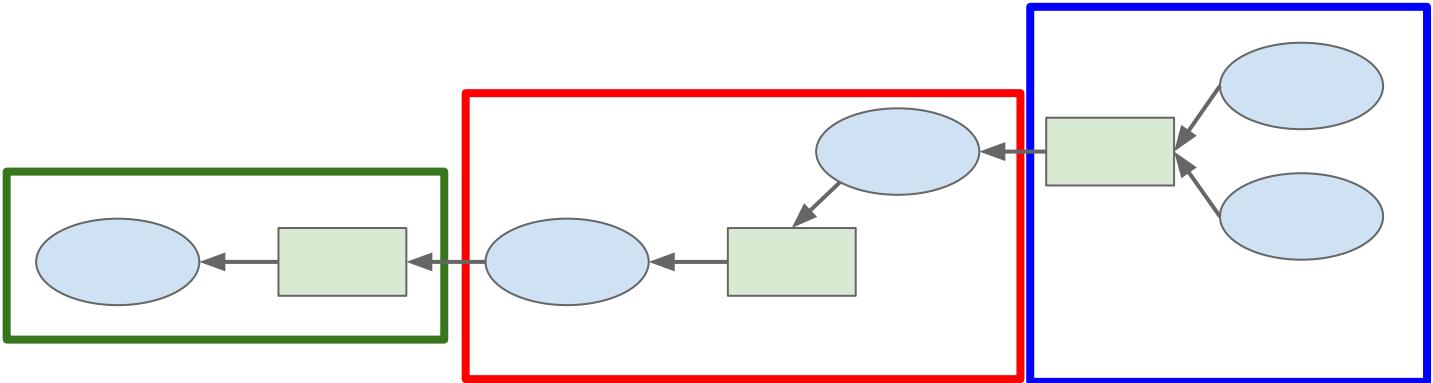


TensorFlow: Tensorboard

Add logging to code to record loss, stats, etc
Run server and get pretty graphs!



TensorFlow: Distributed Version



Split one graph
over multiple
machines!



<https://www.tensorflow.org/deploy/distributed>

Side Note: Theano

TensorFlow is similar in many ways to **Theano** (earlier framework from Montreal)

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Side Note: Theano

Define symbolic variables
(similar to TensorFlow
placeholder)



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Side Note: Theano

Forward pass: compute predictions and loss



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Side Note: Theano

Forward pass: compute predictions and loss
(no computation performed yet)



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Side Note: Theano

Ask Theano to compute
gradients for us
(no computation performed yet)



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Side Note: Theano

Compile a **function** that computes loss, scores, and gradients from data and weights



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Side Note: Theano

```
# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-2 * np.random.randn(D, H)
ww2 = 1e-2 * np.random.randn(H, C)

learning_rate = 1e-1
for t in xrange(50):
    loss, scores, dww1, dww2 = f(xx, yy, ww1, ww2)
    print loss
    ww1 -= learning_rate * dww1
    ww2 -= learning rate * dww2
```

Run the function many times to train the network

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

PyTorch

(Facebook)

PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray,
but runs on GPU

Variable: Node in a
computational graph; stores
data and gradient

Module: A neural network
layer; may store state or
learnable weights

PyTorch: Three Levels of Abstraction

TensorFlow equivalent

Tensor: Imperative ndarray,
but runs on GPU

Numpy array

Variable: Node in a
computational graph; stores
data and gradient

Tensor, Variable, Placeholder

Module: A neural network
layer; may store state or
learnable weights

`tf.layers`, or `TFSlim`, or `TFLearn`,
or `Sonnet`, or

PyTorch: Tensors

PyTorch Tensors are just like numpy arrays, but they can run on GPU.

No built-in notion of computational graph, or gradients, or deep learning.

Here we fit a two-layer net using PyTorch Tensors:

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Create random tensors
for data and weights



```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Forward pass: compute predictions and loss



```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Backward pass:
manually compute
gradients



```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Gradient descent
step on weights

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

To run on GPU, just cast tensors to a cuda datatype!

```
import torch
```

```
dtype = torch.cuda.FloatTensor
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in).type(dtype)
```

```
y = torch.randn(N, D_out).type(dtype)
```

```
w1 = torch.randn(D_in, H).type(dtype)
```

```
w2 = torch.randn(H, D_out).type(dtype)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    h = x.mm(w1)
```

```
    h_relu = h.clamp(min=0)
```

```
    y_pred = h_relu.mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    grad_y_pred = 2.0 * (y_pred - y)
```

```
    grad_w2 = h_relu.t().mm(grad_y_pred)
```

```
    grad_h_relu = grad_y_pred.mm(w2.t())
```

```
    grad_h = grad_h_relu.clone()
```

```
    grad_h[h < 0] = 0
```

```
    grad_w1 = x.t().mm(grad_h)
```

```
    w1 -= learning_rate * grad_w1
```

```
    w2 -= learning_rate * grad_w2
```

PyTorch: Autograd

A PyTorch **Variable** is a node in a computational graph

x.data is a Tensor

x.grad is a Variable of gradients
(same shape as x.data)

x.grad.data is a Tensor of gradients

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

PyTorch Tensors and Variables
have the same API!

Variables remember how they were
created (for backprop)

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

We will not want gradients
(of loss) with respect to data

Do want gradients with
respect to weights

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out)) requires grad=False
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

Forward pass looks exactly the same as the Tensor version, but everything is a variable now

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

Compute gradient of loss
with respect to w1 and w2
(zero out grads first)

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

Make gradient step on weights



PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward for Tensors

(similar to modular layers in A2)

```
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

PyTorch: New Autograd Functions

```
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

Can use our new autograd function in the forward pass

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    relu = ReLU()
    y_pred = relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: nn

Higher-level wrapper for working with neural nets

Similar to Keras and friends ...
but only one, and it's good =)

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

PyTorch: nn

Define our model as a sequence of layers

nn also defines common loss functions

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

PyTorch: nn

Forward pass: feed data
to model, and prediction
to loss function



```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

PyTorch: nn

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Backward pass:
compute all gradients



PyTorch: nn

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Make gradient step on
each model parameter



PyTorch: optim

Use an **optimizer** for different update rules

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

PyTorch: optim

Update all parameters
after computing gradients

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

PyTorch: nn Define new Modules

A PyTorch **Module** is a neural net layer; it inputs and outputs Variables

Modules can contain weights (as Variables) or other Modules

You can define your own Modules using autograd!

```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

PyTorch: nn

Define new Modules

Define our whole model
as a single Module

```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

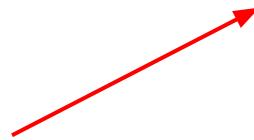
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

PyTorch: nn

Define new Modules

Initializer sets up two children (Modules can contain modules)



```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

PyTorch: nn

Define new Modules

Define forward pass using child modules and autograd ops on Variables

No need to define backward - autograd will handle it

```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

PyTorch: nn

Define new Modules

Construct and train an instance of our model

```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

PyTorch: DataLoaders

A **DataLoader** wraps a **Dataset** and provides minibatching, shuffling, multithreading, for you

When you need to load custom data, just write your own Dataset class

```
import torch
from torch.autograd import Variable
from torch.utils.data import TensorDataset, DataLoader

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

loader = DataLoader(TensorDataset(x, y), batch_size=8)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for epoch in range(10):
    for x_batch, y_batch in loader:
        x_var, y_var = Variable(x), Variable(y)
        y_pred = model(x_var)
        loss = criterion(y_pred, y_var)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

PyTorch: DataLoaders

Iterate over loader to form minibatches

Loader gives Tensors so you need to wrap in Variables

```
import torch
from torch.autograd import Variable
from torch.utils.data import TensorDataset, DataLoader

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

loader = DataLoader(TensorDataset(x, y), batch_size=8)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

for epoch in range(10):
    for x_batch, y_batch in loader:
        x_var, y_var = Variable(x), Variable(y)
        y_pred = model(x_var)
        loss = criterion(y_pred, y_var)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

PyTorch: Pretrained Models

Super easy to use pretrained models with torchvision
<https://github.com/pytorch/vision>

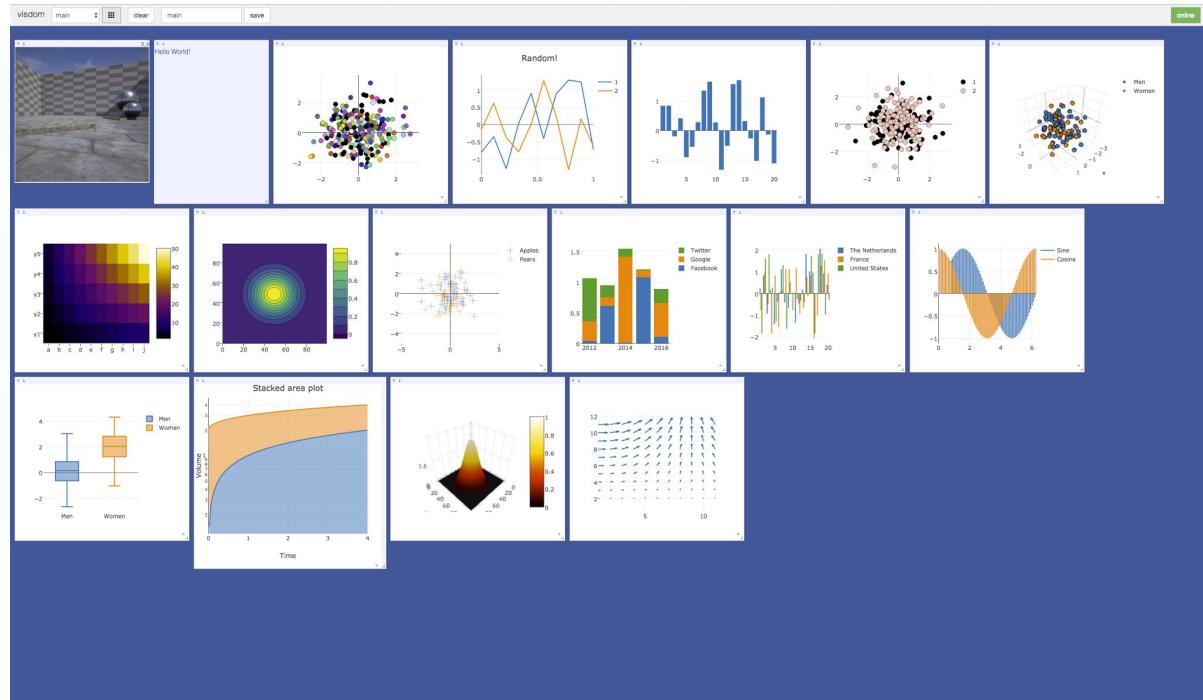
```
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)
```

PyTorch: Visdom

Somewhat similar to
TensorBoard: add logging
to your code, then
visualized in a browser

Can't visualize
computational graph
structure (yet?)



This image is licensed under CC-BY 4.0; no changes were made to the image

<https://github.com/facebookresearch/visdom>

Aside: Torch

Direct ancestor of PyTorch
(they share a lot of C backend)

Written in Lua, not Python

PyTorch has 3 levels of abstraction: **Tensor**, **Variable**, and **Module**

Torch only has 2: **Tensor**, **Module**

More details: Check 2016 slides

```
require 'torch'
require 'nn'
require 'optim'

local N, D, H, C = 64, 256, 512, 10

local model = nn.Sequential()
model:add(nn.Linear(D, H))
model:add(nn.ReLU())
model:add(nn.Linear(H, C))
local loss_fn = nn.CrossEntropyCriterion()

local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)
local weights, grad_weights = model:getParameters()

local function f(w)
    assert(w == weights)
    local scores = model:forward(x)
    local loss = loss_fn:forward(scores, y)

    grad_weights:zero()
    local grad_scores = loss_fn:backward(scores, y)
    local grad_x = model:backward(x, grad_scores)

    return loss, grad_weights
end

local state = {learningRate=1e-3}
for t = 1, 100 do
    optim.adam(f, weights, state)
end
```

Aside: Torch

Build a model as a sequence of layers, and a loss function



```
require 'torch'
require 'nn'
require 'optim'

local N, D, H, C = 64, 256, 512, 10

local model = nn.Sequential()
model:add(nn.Linear(D, H))
model:add(nn.ReLU())
model:add(nn.Linear(H, C))
local loss_fn = nn.CrossEntropyCriterion()

local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)
local weights, grad_weights = model:getParameters()

local function f(w)
    assert(w == weights)
    local scores = model:forward(x)
    local loss = loss_fn:forward(scores, y)

    grad_weights:zero()
    local grad_scores = loss_fn:backward(scores, y)
    local grad_x = model:backward(x, grad_scores)

    return loss, grad_weights
end

local state = {learningRate=1e-3}
for t = 1, 100 do
    optim.adam(f, weights, state)
end
```

Aside: Torch

Define a callback
that inputs weights,
produces loss and
gradient on weights



```
require 'torch'
require 'nn'
require 'optim'

local N, D, H, C = 64, 256, 512, 10

local model = nn.Sequential()
model:add(nn.Linear(D, H))
model:add(nn.ReLU())
model:add(nn.Linear(H, C))
local loss_fn = nn.CrossEntropyCriterion()

local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)
local weights, grad_weights = model:getParameters()

local function f(w)
    assert(w == weights)
    local scores = model:forward(x)
    local loss = loss_fn:forward(scores, y)

    grad_weights:zero()
    local grad_scores = loss_fn:backward(scores, y)
    local grad_x = model:backward(x, grad_scores)

    return loss, grad_weights
end

local state = {learningRate=1e-3}
for t = 1, 100 do
    optim.adam(f, weights, state)
end
```

Aside: Torch

Forward: compute scores and loss

```
require 'torch'
require 'nn'
require 'optim'

local N, D, H, C = 64, 256, 512, 10

local model = nn.Sequential()
model:add(nn.Linear(D, H))
model:add(nn.ReLU())
model:add(nn.Linear(H, C))
local loss_fn = nn.CrossEntropyCriterion()

local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)
local weights, grad_weights = model:getParameters()

local function f(w)
    assert(w == weights)
    local scores = model:forward(x)
    local loss = loss_fn:forward(scores, y)

    grad_weights:zero()
    local grad_scores = loss_fn:backward(scores, y)
    local grad_x = model:backward(x, grad_scores)

    return loss, grad_weights
end

local state = {learningRate=1e-3}
for t = 1, 100 do
    optim.adam(f, weights, state)
end
```

Aside: Torch

Backward: compute gradient

(no autograd, need to pass grad_scores around)

```
require 'torch'
require 'nn'
require 'optim'

local N, D, H, C = 64, 256, 512, 10

local model = nn.Sequential()
model:add(nn.Linear(D, H))
model:add(nn.ReLU())
model:add(nn.Linear(H, C))
local loss_fn = nn.CrossEntropyCriterion()

local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)
local weights, grad_weights = model:getParameters()

local function f(w)
    assert(w == weights)
    local scores = model:forward(x)
    local loss = loss_fn:forward(scores, y)

grad_weights:zero()
local grad_scores = loss_fn:backward(scores, y)
local grad_x = model:backward(x, grad_scores)

    return loss, grad_weights
end

local state = {learningRate=1e-3}
for t = 1, 100 do
    optim.adam(f, weights, state)
end
```

Aside: Torch

Pass callback to
optimizer over and over

```
require 'torch'  
require 'nn'  
require 'optim'  
  
local N, D, H, C = 64, 256, 512, 10  
  
local model = nn.Sequential()  
model:add(nn.Linear(D, H))  
model:add(nn.ReLU())  
model:add(nn.Linear(H, C))  
local loss_fn = nn.CrossEntropyCriterion()  
  
local x = torch.randn(N, D)  
local y = torch.Tensor(N):random(C)  
local weights, grad_weights = model:getParameters()  
  
local function f(w)  
    assert(w == weights)  
    local scores = model:forward(x)  
    local loss = loss_fn:forward(scores, y)  
  
    grad_weights:zero()  
    local grad_scores = loss_fn:backward(scores, y)  
    local grad_x = model:backward(x, grad_scores)  
  
    return loss, grad_weights  
end  
  
local state = {learningRate=1e-3}  
for t = 1, 100 do  
    optim.adam(f, weights, state)  
end
```

Torch vs PyTorch

Torch

- (-) Lua
- (-) No autograd
- (+) More stable
- (+) Lots of existing code
- (0) Fast

PyTorch

- (+) Python
- (+) Autograd
- (-) Newer, still changing
- (-) Less existing code
- (0) Fast

Torch vs PyTorch

Torch

- (-) Lua
- (-) No autograd
- (+) More stable
- (+) Lots of existing code
- (0) Fast

PyTorch

- (+) Python
- (+) Autograd
- (-) Newer, still changing
- (-) Less existing code
- (0) Fast

Conclusion: Probably use PyTorch for new projects

Static vs Dynamic Graphs

TensorFlow: Build graph once, then run many times (**static**)

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

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

Build graph

Run each iteration

PyTorch: Each forward pass defines a new graph (**dynamic**)

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

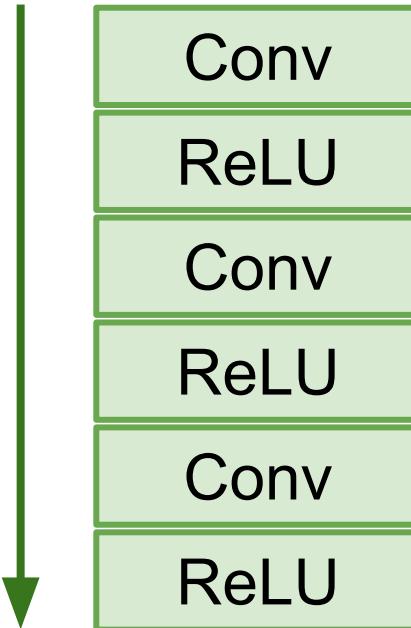
    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

New graph each iteration

Static vs Dynamic: Optimization

With static graphs,
framework can
optimize the
graph for you
before it runs!

The graph you wrote



Equivalent graph with
fused operations



Static vs Dynamic: Serialization

Static

Once graph is built, can **serialize** it and run it without the code that built the graph!

Dynamic

Graph building and execution are intertwined, so always need to keep code around

Static vs Dynamic: Conditional

$$y = \begin{cases} w_1 * x & \text{if } z > 0 \\ w_2 * x & \text{otherwise} \end{cases}$$

Static vs Dynamic: Conditional

$$y = \begin{cases} w_1 * x & \text{if } z > 0 \\ w_2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

```
N, D, H = 3, 4, 5

x = Variable(torch.randn(N, D))
w1 = Variable(torch.randn(D, H))
w2 = Variable(torch.randn(D, H))

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

Static vs Dynamic: Conditional

$$y = \begin{cases} w_1 * x & \text{if } z > 0 \\ w_2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

```
N, D, H = 3, 4, 5

x = Variable(torch.randn(N, D))
w1 = Variable(torch.randn(D, H))
w2 = Variable(torch.randn(D, H))

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

TensorFlow: Special TF control flow operator!

```
N, D, H = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=None)
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(D, H))

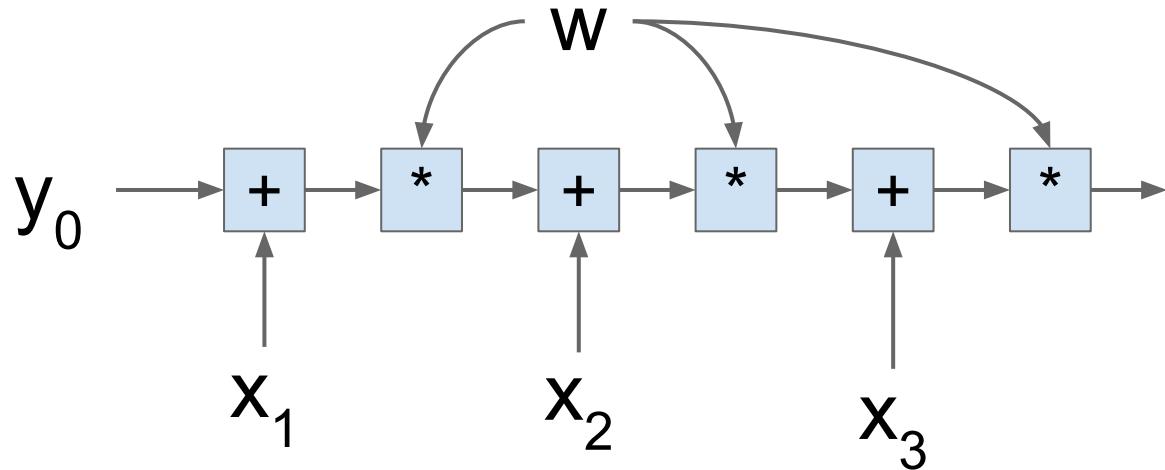
def f1(): return tf.matmul(x, w1)
def f2(): return tf.matmul(x, w2)
y = tf.cond(tf.less(z, 0), f1, f2)

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        z: 10,
        w1: np.random.randn(D, H),
        w2: np.random.randn(D, H),
    }
    y_val = sess.run(y, feed_dict=values)
```



Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$



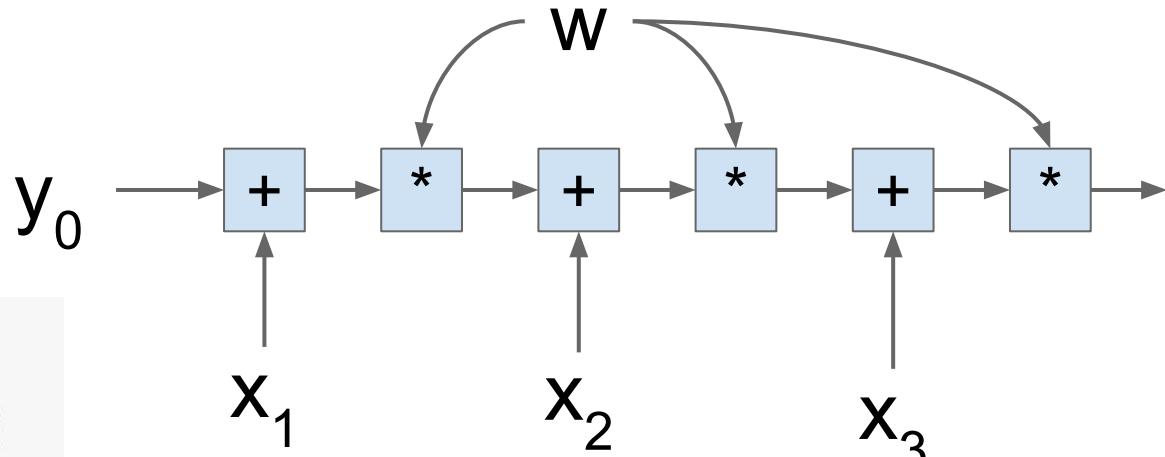
Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$

PyTorch: Normal Python

```
T, D = 3, 4
y0 = Variable(torch.randn(D))
x = Variable(torch.randn(T, D))
w = Variable(torch.randn(D))

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```



Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$

PyTorch: Normal Python

```
T, D = 3, 4
y0 = Variable(torch.randn(D))
x = Variable(torch.randn(T, D))
w = Variable(torch.randn(D))

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```

TensorFlow: Special TF control flow

```
T, N, D = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(T, D))
y0 = tf.placeholder(tf.float32, shape=(D,))
w = tf.placeholder(tf.float32, shape=(D,))

def f(prev_y, cur_x):
    return (prev_y + cur_x) * w

y = tf.foldl(f, x, y0)

with tf.Session() as sess:
    values = {
        x: np.random.randn(T, D),
        y0: np.random.randn(D),
        w: np.random.randn(D),
    }
    y_val = sess.run(y, feed_dict=values)
```

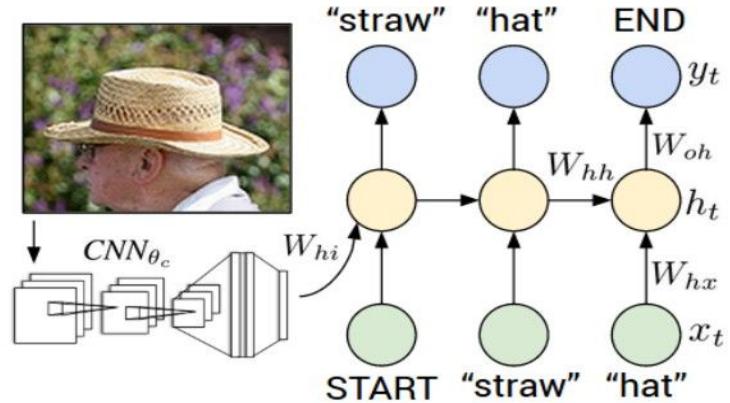
Dynamic Graphs in TensorFlow

TensorFlow Fold make dynamic graphs easier in TensorFlow through **dynamic batching**

Looks et al, "Deep Learning with Dynamic Computation Graphs", ICLR 2017
<https://github.com/tensorflow/fold>

Dynamic Graph Applications

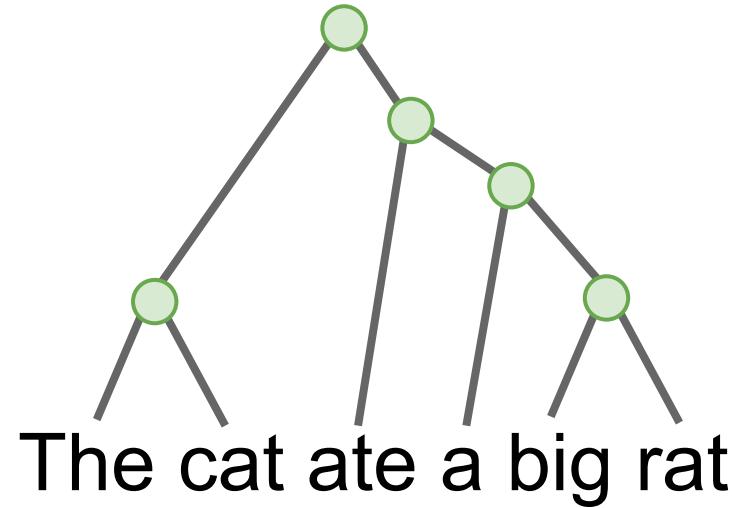
- Recurrent networks



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Dynamic Graph Applications

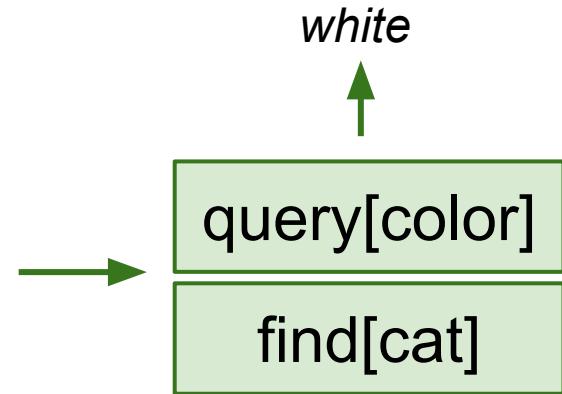
- Recurrent networks
- Recursive networks



Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks

*What color
is the cat?*



Andreas et al, "Neural Module Networks", CVPR 2016

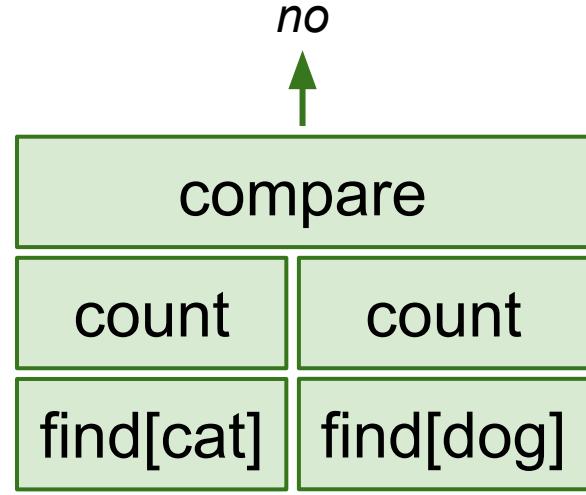
Andreas et al, "Learning to Compose Neural Networks for Question Answering", NAACL 2016

[This image](#) is in the public domain

Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks

*Are there
more cats
than dogs?*



Andreas et al, "Neural Module Networks", CVPR 2016

Andreas et al, "Learning to Compose Neural Networks for Question Answering", NAACL 2016

[This image](#) is in the public domain

Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks
- (Your creative idea here)

Caffe

(UC Berkeley)

Caffe Overview

- Core written in C++
- Has Python and MATLAB bindings
- Good for training or finetuning feedforward classification models
- Often no need to write code!
- Not used as much in research anymore, still popular for deploying models

Caffe: Training / Finetuning

No need to write code!

1. Convert data (run a script)
2. Define net (edit prototxt)
3. Define solver (edit prototxt)
4. Train (with pretrained weights) (run a script)

Caffe step 1: Convert Data

- DataLayer reading from LMDB is the easiest
- Create LMDB using [convert_imageset](#)
- Need text file where each line is
 - “[path/to/image.jpeg] [label]”
- Create HDF5 file yourself using h5py

Caffe step 1: Convert Data

- `ImageDataLayer`: Read from image files
- `WindowDataLayer`: For detection
- `HDF5Layer`: Read from HDF5 file
- From memory, using Python interface
- All of these are harder to use (except Python)

Caffe step 2: Define Network (prototxt)

```
name: "LogisticRegressionNet"
layers {
    top: "data"
    top: "label"
    name: "data"
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
}
inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe step 2: Define Network (prototxt)

- .prototxt can get ugly for big models
- ResNet-152 prototxt is 6775 lines long!
- Not “compositional”; can’t easily define a residual block and reuse

```
1 name: "ResNet-152"
2 input: "data"
3 input_dim: 1
4 input_dim: 3
5 input_dim: 224
6 input_dim: 224
7
8 layer {
9     bottom: "data"
10    top: "conv1"
11    name: "conv1"
12    type: "Convolution"
13    convolution_param {
14        num_output: 64
15        kernel_size: 7
16        pad: 3
17        stride: 2
18        bias_term: false
19    }
20}
21
22 layer {
23    bottom: "conv1"
24    top: "conv1"
25    name: "bn_conv1"
26    type: "BatchNorm"
27    batch_norm_param {
28        use_global_stats: true
29    }
30}

6747
6748
6749
6750
6751
6752
6753
6754
6755
6756
6757
6758
6759
6760
6761
6762
6763
6764
6765
6766
6767
6768
6769
6770
6771
6772
6773
6774

layer {
    bottom: "res5c"
    top: "pool5"
    name: "pool5"
    type: "Pooling"
    pooling_param {
        kernel_size: 7
        stride: 1
        pool: AVE
    }
}

layer {
    bottom: "pool5"
    top: "fc1000"
    name: "fc1000"
    type: "InnerProduct"
    inner_product_param {
        num_output: 1000
    }
}

layer {
    bottom: "fc1000"
    top: "prob"
    name: "prob"
    type: "Softmax"
}
```

<https://github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-152-deploy.prototxt>

Caffe step 3: Define Solver (prototxt)

- Write a prototxt file defining a [SolverParameter](#)
- If finetuning, copy existing solver.prototxt file
 - Change net to be your net
 - Change snapshot_prefix to your output
 - Reduce base learning rate (divide by 100)
 - Maybe change max_iter and snapshot

```
1 | net: "models/bvlc_alexnet/train_val.prototxt"
2 | test_iter: 1000
3 | test_interval: 1000
4 | base_lr: 0.01
5 | lr_policy: "step"
6 | gamma: 0.1
7 | stepsize: 100000
8 | display: 20
9 | max_iter: 450000
10 | momentum: 0.9
11 | weight_decay: 0.0005
12 | snapshot: 10000
13 | snapshot_prefix: "models/bvlc_alexnet/caffe_alexnet_train"
14 | solver_mode: GPU
```

Caffe step 4: Train!

```
./build/tools/caffe train \
-gpu 0 \
-model path/to/trainval.prototxt \
-solver path/to/solver.prototxt \
-weights path/to/pretrained_weights.caffemodel
```

<https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp>

Caffe step 4: Train!

```
./build/tools/caffe train \
    -gpu 0 \
    -model path/to/trainval.prototxt \
    -solver path/to/solver.prototxt \
    -weights path/to/pretrained_weights.caffemodel
```

-gpu -1 for CPU-only
-gpu all for multi-gpu

<https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp>

Caffe Model Zoo

AlexNet, VGG,
GoogLeNet, ResNet,
plus others

The screenshot shows a GitHub repository page for 'BVLC / caffe'. The main heading is 'Model Zoo'. Below it, a note says 'Alex Kendall edited this page 13 days ago · 61 revisions'. A section titled 'Check out the [model zoo documentation](#) for details.' follows. Another section, 'To acquire a model:', lists two steps: 1. download the model gist by `./scripts/download_model_from_gist.sh <gist_id>` where `<dirname>` is optional and defaults to `caffe/models`. 2. download the model weights by `./scripts/download_model_binary.py <model_dir>` where `<model_dir>` is the gist directory from the first step. At the bottom, it says 'or visit the [model zoo documentation](#) for complete instructions.' On the right side, there's a sidebar with a 'Pages' section containing links to 'Home', 'Caffe on EC2 Ubuntu 14.04 Cuda 7', 'Contributing', 'Development', 'IDE Nvidia's Eclipse Insight', and 'Install Caffe on EC2 from scratch'.

<https://github.com/BVLC/caffe/wiki/Model-Zoo>

Caffe Python Interface

Not much documentation...

Read the code! Two most important files:

- [caffe/python/caffe/_caffe.cpp](#):
 - Exports Blob, Layer, Net, and Solver classes
- [caffe/python/caffe/pycaffe.py](#)
 - Adds extra methods to Net class

Caffe Python Interface

Good for:

- Interfacing with numpy
- Extract features: Run net forward
- Compute gradients: Run net backward (DeepDream, etc)
- Define layers in Python with numpy (CPU only)

Caffe Pros / Cons

- (+) Good for feedforward networks
- (+) Good for finetuning existing networks
- (+) Train models without writing any code!
- (+) Python interface is pretty useful!
- (+) Can deploy without Python
- (-) Need to write C++ / CUDA for new GPU layers
- (-) Not good for recurrent networks
- (-) Cumbersome for big networks (GoogLeNet, ResNet)

Caffe2

(Facebook)

Caffe2 Overview

- Very new - released a week ago =)
- Static graphs, somewhat similar to TensorFlow
- Core written in C++
- Nice Python interface
- Can train model in Python, then serialize and deploy without Python
- Works on iOS / Android, etc

Google:
TensorFlow



*“One framework
to rule them all”*

Facebook:
PyTorch + Caffe2



Research



Production

My Advice:

TensorFlow is a safe bet for most projects. Not perfect but has huge community, wide usage. Maybe pair with high-level wrapper (Keras, Sonnet, etc)

I think **PyTorch** is best for research. However still new, there can be rough patches.

Use **TensorFlow** for one graph over many machines

Consider **Caffe**, **Caffe2**, or **TensorFlow** for production deployment

Consider **TensorFlow** or **Caffe2** for mobile

Next Time: CNN Architecture Case Studies

Caffe step 2: Define Network (prototxt)

```
name: "LogisticRegressionNet"
layers {
    top: "data" ← Layers and Blobs
    top: "label"
    name: "data" ← often have same
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
}
inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe step 2: Define Network (prototxt)

```
name: "LogisticRegressionNet"
layers {
    top: "data"           ← Layers and Blobs
    top: "label"
    name: "data"          ← often have same
    type: HDF5_DATA       name!
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1           ← Learning rates
    blobs_lr: 2           ← (weight + bias)
    weight_decay: 1         ← Regularization
    weight_decay: 0         ← (weight + bias)
}

inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe step 2: Define Network (prototxt)

```
name: "LogisticRegressionNet"
layers {
    top: "data"           ← Layers and Blobs
    top: "label"
    name: "data"          ← often have same name!
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1           ← Learning rates (weight + bias)
    blobs_lr: 2
    weight_decay: 1        ← Regularization (weight + bias)
    weight_decay: 0
}
inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe step 2: Define Network (prototxt)

```
name: "LogisticRegressionNet"
layers {
    top: "data"           ← Layers and Blobs
    top: "label"
    name: "data"          ← often have same name!
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1           ← Learning rates (weight + bias)
    blobs_lr: 2
    weight_decay: 1        ← Regularization (weight + bias)
    weight_decay: 0
}

inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Set these to 0 to freeze a layer