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**
**Regularization**: Add noise, then marginalize out

**Train** \( y = f_W(x, z) \)

**Test** \( y = f(x) = E_z[f(x, z)] \)

**Optimization**: SGD+Momentum, Nesterov, RMSProp, Adam

**Transfer Learning**

1. Reinitialize this and train
2. Freeze these
Today

- CPU vs GPU
- Deep Learning Frameworks
  - Caffe / Caffe2
  - Theano / TensorFlow
  - Torch / PyTorch
CPU vs GPU
My computer
Spot the CPU!
(central processing unit)
Spot the GPUs!
(graphics processing unit)
NVIDIA vs AMD
NVIDIA vs AMD
## CPU vs GPU

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

**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
  - For deep learning just use existing libraries

https://www.udacity.com/course/cs344
CPU vs GPU in practice

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

(CPU performance not well-optimized, a little unfair)
CPU vs GPU in practice

cuDNN much faster than “unoptimized” CUDA

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

2.8x  3.0x  3.1x  3.4x  2.8x

N=16 Forward + Backward time (ms)

Intel E5-2620 v3  Pascal Titan X (no cuDNN)  Pascal Titan X (cuDNN 5.1)
CPU / GPU Communication

Model is here

Data is here
CPU / GPU Communication

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  
(U Montreal)  →  TensorFlow  
(Google)
This year ...

Caffe  (UC Berkeley)

Torch  (NYU / Facebook)

Theano (U Montreal)

Caffe2 (Facebook)

PyTorch (Facebook)

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

And others...

Paddle  (Baidu)

CNTK  (Microsoft)

MXNet  (Amazon)

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

\[ f = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]
Recall: Computational Graphs

input image

weights

loss

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Recall: Computational Graphs

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

```python
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

```python
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

```python
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

```python
# 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)

g = tf.placeholder(tf.float32, shape=(N, D), name='g')
g = tf.placeholder(tf.float32, shape=(N, D), name='g')
g = tf.placeholder(tf.float32, shape=(N, D), name='g')

g = tf.placeholder(tf.float32, shape=(N, D), name='g')
g = tf.placeholder(tf.float32, shape=(N, D), name='g')
g = tf.placeholder(tf.float32, shape=(N, D), name='g')

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

```
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

```
# 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)

ggrad_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

```python
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

TensorFlow

```
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
```
Computational Graphs

PyTorch

```python
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

PyTorch

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

```python
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

\[ x \times y \rightarrow a \]
\[ a + b \rightarrow b \]
\[ b + z \rightarrow c \]

PyTorch

```python
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)
```

Forward pass looks just like numpy
Computational Graphs

```
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
Σ = torch.sum(b)
c = torch.sum(Σ)

c.backward()
```

Calling `c.backward()` computes all gradients
Computational Graphs

PyTorch

Run on GPU by casting to .cuda()
Numpy

```python
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

```python
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

```python
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)

a.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
TensorFlow: Neural Net

```python
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

Then **run** the graph many times
TensorFlow: Neural Net

Create **placeholders** for input $x$, weights $w_1$ and $w_2$, and targets $y$.

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

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

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

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

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

No computation happens here - just building the graph!
Tell TensorFlow to compute loss of gradient with respect to $w_1$ and $w_2$.

Again, no computation here - just building the graph.
Now done building our graph, so we enter a `session` so we can actually run the graph.
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

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

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_wl, 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
```
Train the network: Run the graph over and over, use gradient to update weights.
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

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

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

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

    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)
Add **assign** operations to update w1 and w2 as part of the graph!
TensorFlow: Neural Net

Run graph once to initialize w1 and w2

Run many times to train

```
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

**Problem:** loss not going down! Assign calls not actually being executed!
TensorFlow: Neural Net

Add dummy graph node that depends on updates

Tell graph to compute dummy node

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

Use predefined common losses

\[
\begin{align*}
N, D, H &= 64, 1000, 100 \\
x &= \text{tf.placeholder(tf.float32, shape=(N, D))} \\
y &= \text{tf.placeholder(tf.float32, shape=(N, D))} \\
w1 &= \text{tf.Variable(tf.random_normal((D, H)))} \\
w2 &= \text{tf.Variable(tf.random_normal((H, D)))} \\
h &= \text{tf.maximum(tf.matmul(x, w1), 0)} \\
y_{\text{pred}} &= \text{tf.matmul(h, w2)} \\
\text{loss} &= \text{tf.losses.mean_squared_error}(y_{\text{pred}}, y) \\
\text{optimizer} &= \text{tf.train.GradientDescentOptimizer(1e-3)} \\
\text{updates} &= \text{optimizer.minimize(loss)} \\
\end{align*}
\]

```python
with \text{tf.Session()} \text{as sess:}
    \text{sess.run(tf.global_variables_initializer())}
    \text{values} = \{\text{x: np.random.randn(N, D)},
                       \text{y: np.random.randn(N, D)},\}

    \text{for t in range(50):}
        \text{loss\_val, \_} = \text{sess.run([loss, updates],}
                          \text{feed\_dict=values)}
```
Use Xavier initializer

```
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(1e0)
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)

```python
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=1e0)
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

```python
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=1e0)
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

```python
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=1e0)
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

```python
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=1e0)
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)
TensorFlow: Other High-Level Wrappers

- **Keras** ([https://keras.io/](https://keras.io/))
- **TFLearn** ([http://tflearn.org/](http://tflearn.org/))
- **tf.layers** ([https://www.tensorflow.org/api_docs/python/tf/layers](https://www.tensorflow.org/api_docs/python/tf/layers))
- **TF-Slim** ([https://github.com/tensorflow/models/tree/master/inception/inception/slim](https://github.com/tensorflow/models/tree/master/inception/inception/slim))
- **tf.contrib.learn** ([https://www.tensorflow.org/get_started/tflearn](https://www.tensorflow.org/get_started/tflearn))
- **Pretty Tensor** ([https://github.com/google/prettytensor](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)
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)

```python
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)

```python
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

```python
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)
Side Note: Theano

Ask Theano to compute **gradients** for us (no computation performed yet)
Compile a function that computes loss, scores, and gradients from data and weights.
Run the function many times to train the network

```python
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')

learning_rate = 1e-1
for t in xrange(50):
    loss, scores, dww1, dww2 = f(xx, yy, w1, w2)
    print loss
    ww1 -= learning_rate * dww1
    ww2 -= learning_rate * dww2

# 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

**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

**TensorFlow equivalent**

- Numpy array
- Tensor, Variable, Placeholder
- 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:

```python
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

```python
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

```python
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[grad_h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```
PyTorch: Tensors

```python
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
```

Backward pass: manually compute gradients
PyTorch: Tensors

Gradient descent step on weights

```python
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!

```python
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

```python
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)

```python
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.

```python
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 is not None:
        w1.grad.data.zero_()
    if w2.grad is not None:
        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 is not None:
        w1.grad.data.zero_()
    if w2.grad is not None:
        w2.grad.data.zero_()
    loss.backward()

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

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

```python
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 is not None:
        w1.grad.data.zero_()
    if w2.grad is not None:
        w2.grad.data.zero_()
    loss.backward()

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

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 is not None:
        w1.grad.data.zero_()
    if w2.grad is not None:
        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
Can use our new autograd function in the forward pass

```
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
```

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 =)

```python
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

```python
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

```python
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
```

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

```python
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

```python
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

```python
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()
```
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!

```python
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

```python
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: \texttt{nn}

Define new Modules

Initializer sets up two children (Modules can contain modules)
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

```python
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

```python
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.

```python
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_batch), Variable(y_batch)
        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

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

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_batch), Variable(y_batch)
        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

```python
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?)

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

```lua
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

```python
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.
Aside: Torch

**Forward**: compute scores and loss

```python
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(y == 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)
Aside: Torch

```python
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
```

Pass callback to optimizer over and over
## Torch vs PyTorch

<table>
<thead>
<tr>
<th></th>
<th>Torch</th>
<th>PyTorch</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-) Lua</td>
<td>(-) No autograd</td>
<td>(+) Python</td>
</tr>
<tr>
<td>(-)</td>
<td>(+) More stable</td>
<td>(+) Autograd</td>
</tr>
<tr>
<td></td>
<td>(+) Lots of existing code</td>
<td>(-) Newer, still changing</td>
</tr>
<tr>
<td>(0) Fast</td>
<td>(0) Fast</td>
<td>(-) Less existing code</td>
</tr>
</tbody>
</table>
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)

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

\[
h = tf.maxout(tf.matmul(x, w1), 0) \\
y\_pred = tf.matmul(h, w2) \\
diff = y\_pred - y \\
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1)) \\
grad\_w1, grad\_w2 = tf.gradients(loss, [w1, w2]) \\
\]

\[
\text{learning\_rate} = 1e-5 \\
\text{for } t \text{ in range(500):} \\
\text{new\_w1 = w1.\_assign(w1 - learning\_rate * grad\_w1)} \\
\text{new\_w2 = w2.\_assign(w2 - learning\_rate * grad\_w2)} \\
\text{updates = tf.group(new\_w1, new\_w2)} \\
\]

\[
\text{with tf.Session() as sess:} \\
\text{sess.run(tf.global\_variables\_initializer())} \\
\text{values = {x: np.random.randn(N, D),} \\
\text{y: np.random.randn(N, D),}} \\
\text{losses =[]} \\
\text{for t in range(50):} \\
\text{loss\_val, _ = sess.run([loss, updates],} \\
\text{feed\_dict=values)} \\
\]

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

\[
\text{import torch} \\
\text{from torch.autograd import Variable} \\
\]

\[
N, D\_in, H, D\_out = 64, 1000, 100, 10 \\
x = Variable(torch.randn(N, D\_in), requires\_grad=False) \\
y = Variable(torch.randn(N, D\_out), requires\_grad=False) \\
w1 = Variable(torch.randn(D\_in, H), requires\_grad=True) \\
w2 = Variable(torch.randn(H, D\_out), requires\_grad=True) \\
\]

\[
\text{learning\_rate} = 1e-6 \\
\text{for } t \text{ in range(500):} \\
\text{y\_pred = x.mm(w1).\_clamp(min=0).mm(w2)} \\
\text{loss = (y\_pred - y).\_pow(2).\_sum()} \\
\]

\[
\text{if w1.\_grad: w1.\_grad\_data.\_zero()} \\
\text{if w2.\_grad: w2.\_grad\_data.\_zero()} \\
\text{loss.\_backward()} \\
\text{w1.\_data = learning\_rate * w1.\_grad\_data} \\
\text{w2.\_data = learning\_rate * w2.\_grad\_data} \\
\]

Fei-Fei Li & Justin Johnson & Serena Yeung
Static vs Dynamic: Optimization

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

**The graph you wrote**

- Conv
- ReLU
- Conv
- ReLU
- Conv
- ReLU

**Equivalent graph with fused operations**

- Conv+ReLU
- Conv+ReLU
- Conv+ReLU
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

\[ y = \begin{cases} 
  w_1 \times x & \text{if } z > 0 \\
  w_2 \times x & \text{otherwise} 
\end{cases} \]
Static vs **Dynamic**: Conditional

\[
y = \begin{cases} 
  w_1 \times x & \text{if } z > 0 \\
  w_2 \times x & \text{otherwise}
\end{cases}
\]

**PyTorch**: Normal Python

```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

\[ y = \begin{cases} 
  w_1 \times x & \text{if } z > 0 \\
  w_2 \times x & \text{otherwise} 
\end{cases} \]

**PyTorch**: Normal Python

\[
N, D, H = 3, 4, 5
\]

```python
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!

\[
N, D, H = 3, 4, 5
\]

```python
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

\[ y_t = (y_{t-1} + x_t) \times w \]
Static vs **Dynamic**: Loops

\[ y_t = (y_{t-1} + x_t) \times w \]

**PyTorch**: Normal Python

```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]) \times w
    y.append(next_y)
```
**Static vs Dynamic: Loops**

\[ y_t = (y_{t-1} + x_t) \times w \]

**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]) \times 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) \times 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**

[https://github.com/tensorflow/fold](https://github.com/tensorflow/fold)
Dynamic Graph Applications

- Recurrent networks
Dynamic Graph Applications

- Recurrent networks
- Recursive networks

The cat ate a big rat
Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks

What color is the cat?

This image is in the public domain

Andreas et al, “Neural Module Networks”, CVPR 2016
Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks

Are there more cats than dogs?

Find [cat] count
Find [dog] count
Compare

This image is in the public domain

Andreas et al, “Neural Module Networks”, CVPR 2016
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
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

```
layer {
  name: "ResNet-152"
  input: "data"
}
```

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

```python
net: "models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU
```
Caffe step 4: Train!

```bash
./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

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)

Layers and Blobs often have same name!
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
  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"
top: "label"
  top: "loss"
  name: "loss"
type: SOFTMAX_LOSS
```
Caffe step 2: Define Network (prototxt)

Layers and Blobs often have same name!

Number of output classes

Learning rates (weight + bias)

Regularization (weight + bias)
Caffe step 2: Define Network (prototxt)

Layers and Blobs often have same name!

Number of output classes

Set these to 0 to freeze a layer

Learning rates (weight + bias)

Regularization (weight + bias)