Topics:

• Deep Learning Hardware and Software
Recap: Second-order Optimization

Newton’s method for optimization: solving for the critical point we obtain the Newton update rule:

\[ x^* = a - H^{-1} \nabla f \]

Bad for deep learning! \( O(n^3) \) hessian inversion
- Consider BGFS (approximate hessian) or L-BFGS (don’t store full hessian in memory)
- L-BFGS works better in full-batch setting.
Disable all stochastic components in training
Administrative

- Reminder: anonymous feedback
- Time to work on the project!
Recap: Test-time performance

Regularization
Recap: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common

Recap: Dropout

\[ p = 0.5 \] # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3

test time is unchanged!

Similar to BatchNorm, different behavior train vs test!
Recap: Data Augmentation

1. Load image and label
2. "cat"
3. Transform image
4. CNN
5. Compute loss
Recap: Transfer Learning

<table>
<thead>
<tr>
<th>Task-specific</th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>very little data</td>
<td>Use Linear Classifier on top layer</td>
<td>You’re in trouble... Try linear classifier from different stages</td>
</tr>
<tr>
<td>quite a lot of data</td>
<td>Finetune a few layers</td>
<td>Finetune a larger number of layers</td>
</tr>
</tbody>
</table>
Today

- Deep learning hardware
  - CPU, GPU
- Deep learning software
  - PyTorch and TensorFlow
  - Static and Dynamic computation graphs
Deep Learning
Hardware
Inside a computer
Spot the CPU!
(central processing unit)

This image is licensed under CC-BY 2.0
Spot the GPUs!
(graphics processing unit)
# CPU vs GPU

<table>
<thead>
<tr>
<th></th>
<th>Cores</th>
<th>Clock Speed</th>
<th>Memory</th>
<th>Price</th>
<th>Speed (throughput)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>10</td>
<td>4.3 GHz</td>
<td>System RAM</td>
<td>$385</td>
<td>~640 GFLOPS FP32</td>
</tr>
<tr>
<td>(Intel Core i9-7900k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>10496</td>
<td>1.6 GHz</td>
<td>24 GB GDDR6X</td>
<td>$1499</td>
<td>~35.6 TFLOPS FP32</td>
</tr>
<tr>
<td>(NVIDIA RTX 3090)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 x B

B x C

cuBLAS::GEMM (GEneral Matrix-to-matrix Multiply)

= 

A x C
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
NVIDIA vs AMD
NVIDIA vs AMD
## CPU vs GPU

<table>
<thead>
<tr>
<th></th>
<th>Cores</th>
<th>Clock Speed</th>
<th>Memor y</th>
<th>Price</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong> (Intel Core i7-7700k)</td>
<td>10</td>
<td>4.3 GHz</td>
<td>System RAM</td>
<td>$385</td>
<td>~640 GFLOPs FP32</td>
</tr>
<tr>
<td><strong>GPU</strong> (NVIDIA RTX 3090)</td>
<td>10496</td>
<td>1.6 GHz</td>
<td>24 GB GDDR6 X</td>
<td>$1499</td>
<td>~35.6 TFLOPs FP32</td>
</tr>
<tr>
<td><strong>GPU (Data Center)</strong> NVIDIA A100</td>
<td>6912 CUDA, 432 Tensor</td>
<td>1.5 GHz</td>
<td>40/80 GB HBM2</td>
<td>$3/hr (GCP)</td>
<td>~9.7 TFLOPs FP64 ~20 TFLOPs FP32 ~312 TFLOPs FP16</td>
</tr>
<tr>
<td><strong>TPU</strong> Google Cloud TPUv3</td>
<td>2 Matrix Units (MXUs) per core, 4 cores</td>
<td>?</td>
<td>128 GB HBM</td>
<td>$8/hr (GCP)</td>
<td>~420 TFLOPs (non-standard FP)</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

**TPU**: Specialized hardware for deep learning
Programming GPUs

- CUDA (NVIDIA only)
  - Write C-like code that runs directly on the GPU
  - Optimized APIs: cuBLAS, cuFFT, cuDNN, etc

- OpenCL
  - Similar to CUDA, but runs on anything
  - Usually slower on NVIDIA hardware

- HIP [https://github.com/ROCm-Developer-Tools/HIP](https://github.com/ROCm-Developer-Tools/HIP)
  - New project that automatically converts CUDA code to something that can run on AMD GPUs
  - CS 8803 – GPU at GaTech
    - Taught by Prof. Hyesoon Kim
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
Software
A zoo of frameworks!

Caffe
(UC Berkeley)

Torch
(NYU / Facebook)

Theano
(U Montreal)

Caffe2
(Facebook)
mostly features absorbed by PyTorch

PyTorch
(Facebook)

TensorFlow
(Google)

PaddlePaddle
(Baidu)

Chainer
(Preferred Networks)
The company has officially migrated its research infrastructure to PyTorch

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

CNTK
(Microsoft)

JAX
(Google)

And others...
A zoo of frameworks!

Caffe
(UC Berkeley)

Torch
(NYU / Facebook)

Theano
(U Montreal)

Caffe2
(Facebook)
mostly features absorbed by PyTorch

PyTorch
(Facebook)

TensorFlow
(Google)

PaddlePaddle
(Baidu)

Chainer
(Preferred Networks)
The company has officially migrated its research infrastructure to PyTorch

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

CNTK
(Microsoft)

JAX
(Google)

And others...

We’ll focus on these
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) Quick to develop and test new ideas
(2) Automatically compute gradients
(3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, OpenCL, etc)
Computational Graphs

Numpy

```python
import numpy as np
cpy.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

```
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
```

Good:
- Clean API, easy to write numeric code

Bad:
- Have to compute our own gradients
- Can’t run on GPU
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)
```

PyTorch

```python
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

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

Looks exactly like numpy!
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
```

PyTorch

```python
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

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

c.backward()
print(x.grad)
```

PyTorch handles gradients for us!
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
```

PyTorch

```python
import torch

device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

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

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!
PyTorch
(More details)
PyTorch: Fundamental Concepts

`torch.Tensor`: Like a numpy array, but can run on GPU

`torch.autograd`: Package for building computational graphs out of Tensors, and automatically computing gradients

`torch.nn.Module`: A neural network layer; may store state or learnable weights
PyTorch: Versions

For this class we are using PyTorch version $\geq 1.10$ (newest is 1.12)

Major API change in release 1.0

Be careful if you are looking at older PyTorch code ($<1.0$)!
PyTorch: Tensors

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

```python
import torch

device = torch.device('cpu')

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

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
device = torch.device('cpu')
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

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

device = torch.device('cpu')

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

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
PyTorch: Tensors

Gradient descent step on weights

```python
import torch
device = torch.device('cpu')

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

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 use a different device!

```python
import torch

device = torch.device('cuda:0')

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

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

Creating Tensors with requires_grad=True enables autograd.

Operations on Tensors with requires_grad=True cause PyTorch to build a computational graph.
PyTorch: Autograd

Forward pass looks exactly the same as before, but we don’t need to track intermediate values - PyTorch keeps track of them for us in the graph.
PyTorch: Autograd

```python
import torch

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

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_
```

Compute gradient of loss with respect to w1 and w2
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
wl1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
y_pred = x.mm(wl1).clamp(min=0).mm(w2)
loss = (y_pred - y).pow(2).sum()
loss.backward()

with torch.no_grad():
w1 -= learning_rate * w1.grad
w2 -= learning_rate * w2.grad
w1.grad.zero_()
w2.grad.zero_()}
PyTorch: Autograd

```python
import torch

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

with torch.no_grad():
    w1 -= learning_rate * w1.grad
    w2 -= learning_rate * w2.grad
    w1.grad.zero_()
    w2.grad.zero_
```

Make gradient step on weights, then zero them. Torch.no_grad means “don’t build a computational graph for this part”
PyTorch methods that end in underscore modify the Tensor in-place; methods that don’t return a new Tensor.
Define your own autograd functions by writing forward and backward functions for Tensors.

Use `ctx` object to “cache” values for the backward pass.

```python
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```
PyTorch: New Autograd Functions

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

Use ctx object to “cache” values for the backward pass

Define a helper function to make it easy to use the new function

class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)
Can use our new autograd function in the forward pass

```python
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)
    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)

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

learning_rate = 1e-6
for t in range(500):
    y_pred = my_relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    loss.backward()
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```
PyTorch: New Autograd Functions

In practice you almost never need to define new autograd functions! Only do it when you need custom backward. In this case we can just use a normal PyTorch function.

```python
def my_relu(x):
    return x.clamp(min=0)
```

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

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```
PyTorch: nn

Higher-level wrapper for working with neural nets

Use this! It will make your life easier

```python
import torch

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

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

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

model.zero_grad()
```
PyTorch: nn

Define our model as a sequence of layers; each layer is an object that holds learnable weights

```python
import torch

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

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

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

model.zero_grad()
```
PyTorch: nn

```python
import torch

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

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

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

model.zero_grad()
```

Forward pass: feed data to model, and compute loss
**PyTorch: nn**

```python
import torch

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

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

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

model.zero_grad()
```

- **Forward pass:** feed data to model, and compute loss
- `torch.nn.functional` has useful helpers like loss functions
PyTorch: nn

```
import torch

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

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

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

model.zero_grad()
```

Backward pass: compute gradient with respect to all model weights (they have requires_grad=True)
PyTorch: nn

```python
import torch

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

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

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```
Use an **optimizer** for different update rules
After computing gradients, use optimizer to update params and zero gradients.
PyTorch: nn
Define new Modules

A PyTorch Module is a neural net layer; it inputs and outputs Tensors.

Modules can contain weights or other modules.

You can define your own Modules using autograd!

```python
import torch

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 = torch.randn(N, D_in)
y = torch.randn(N, D_out)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```
PyTorch: nn

Define new Modules

Define our whole model as a single Module

```python
import torch

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

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```
PyTorch: nn

Define new Modules

Initializer sets up two children (Modules can contain modules)
PyTorch: nn
Define new Modules

Define forward pass using child modules
No need to define backward - autograd will handle it

```python
import torch

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 = torch.randn(N, D_in)
y = torch.randn(N, D_out)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```
PyTorch: nn
Define new Modules

```python
import torch

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

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Construct and train an instance of our model
PyTorch: nn
Define new Modules

Very common to mix and match custom Module subclasses and Sequential containers

```python
import torch

class ParallelBlock(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(ParallelBlock, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)
        self.linear2 = torch.nn.Linear(D_in, D_out)
    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return (h1 * h2).clamp(min=0)

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

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

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```
PyTorch: nn

Define new Modules

Define network component as a Module subclass
Define new Modules

Stack multiple instances of the component in a sequential

```python
import torch

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

    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return (h1 * h2).clamp(min=0)

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

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

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```
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: Computational Graphs

Figure reproduced with permission from a Twitter post by Andrej Karpathy.
PyTorch: **Dynamic** Computation Graphs

```python
import torch

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

    loss.backward()
```
**PyTorch: Dynamic Computation Graphs**

Create Tensor objects:

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = 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()
    loss.backward()
```
PyTorch: **Dynamic** Computation Graphs

Build graph data structure AND perform computation
PyTorch: **Dynamic** Computation Graphs

```python
import torch

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

loss.backward()
```

Build graph data structure AND perform computation
PyTorch: **Dynamic** Computation Graphs

Search for path between loss and w1, w2 (for backprop) AND perform computation
PyTorch: **Dynamic** Computation Graphs

Throw away the graph, backprop path, and rebuild it from scratch on every iteration

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = 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()
    loss.backward()
```
PyTorch: **Dynamic** Computation Graphs

Build graph data structure AND perform computation

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = 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()
    loss.backward()
```
PyTorch: **Dynamic** Computation Graphs

```
import torch

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

Build graph data structure AND perform computation
Search for path between loss and w1, w2 (for backprop) AND perform computation
**PyTorch: Dynamic Computation Graphs**

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

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

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = 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()
    loss.backward()
```
Static Computation Graphs

Alternative: Static graphs

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

Step 2: Reuse the same graph on every iteration

```
graph = build_graph()
for x_batch, y_batch in loader:
    run_graph(graph, x=x_batch, y=y_batch)
```
TensorFlow
TensorFlow Versions

Pre-2.0 (1.14 latest)
Default static graph, optionally dynamic graph (eager mode).

2.0+
Default dynamic graph, optionally static graph.
We use 2.4 in this class.
TensorFlow: Neural Net (Pre-2.0)

(Assume imports at the top of each snippet)
TensorFlow: Neural Net (Pre-2.0)

First **define** computational graph

Then **run** the graph many times

```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: 2.0+ vs. pre-2.0

Tensorflow 2.0+:
“Eager” Mode by default

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

with tf.GradientTape() as tape:
    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))

gradients = tape.gradient(loss, [w1, w2])
```

Tensorflow 1.13

```
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))

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: 2.0+ vs. pre-2.0

TensorFlow 1.13

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
```

```
with tf.GradientTape() as tape:
  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])
```

```
values = {
x: np.random.randn(N, D),
w1: np.random.randn(D, H),
w2: np.random.randn(H, D),
v: 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 2.0+:
“Eager” Mode by default

```
assert(tf.executing_eagerly())
```
TensorFlow: 2.0+ vs. pre-2.0

TensorFlow 2.0+:
"Eager" Mode by default

```python
with tf.GradientTape() as tape:
    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])
```

TensorFlow 1.13

```python
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
```

```python
assert(tf.executing_eagerly())
```
TensorFlow: Neural Net

- Convert input numpy arrays to TF tensors.
- Create weights as `tf.Variable`.

```python
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

with tf.GradientTape() as tape:
    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))
gradients = tape.gradient(loss, [w1, w2]).
```
TensorFlow: Neural Net

Use `tf.GradientTape()` context to build **dynamic** computation graph.

```python
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

with tf.GradientTape() as tape:
    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))
gradients = tape.gradient(loss, [w1, w2]).
```
TensorFlow: Neural Net

All forward-pass operations in the contexts (including function calls) get traced for computing gradient later.

```python
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

with tf.GradientTape() as tape:
    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))
gradients = tape.gradient(loss, [w1, w2])
```
TensorFlow: Neural Net

Forward pass

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

with tf.GradientTape() as tape:
    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))

gradients = tape.gradient(loss, [w1, w2]).
```
TensorFlow: Neural Net

tape.gradient() uses the traced computation graph to compute gradient for the weights

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

with tf.GradientTape() as tape:
    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))
gradients = tape.gradient(loss, [w1, w2])
```
TensorFlow: Neural Net

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    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))

gradients = tape.gradient(loss, [w1, w2])
```
TensorFlow: Neural Net

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

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        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))
        gradients = tape.gradient(loss, [w1, w2])
        w1.assign(w1 - learning_rate * gradients[0])
        w2.assign(w2 - learning_rate * gradients[1])
```
Train the network: Run the training step over and over, use gradient to update weights

```python
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        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))
        gradients = tape.gradient(loss, [w1, w2])
w1.assign(w1 - learning_rate * gradients[0])
w2.assign(w2 - learning_rate * gradients[1])
```
Can use an optimizer to compute gradients and update weights

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

optimizer = tf.optimizers.SGD(1e-6)

learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        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))
        gradients = tape.gradient(loss, [w1, w2])

    optimizer.apply_gradients(zip(gradients, [w1, w2]))
```
TensorFlow: Loss

Use predefined loss functions

```python
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights
optimizer = tf.optimizers.SGD(1e-6)

for t in range(50):
    with tf.GradientTape() as tape:
        h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    gradients = tape.gradient(loss, [w1, w2])
    optimizer.apply_gradients(zip(gradients, [w1, w2]))
```
TensorFlow: High-Level Wrappers

Keras (https://keras.io/)
tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)
tf.estimator (https://www.tensorflow.org/api_docs/python/tf/estimator)

Sonnet (https://github.com/deepmind/sonnet)
TFLearn (http://tflearn.org/)
TensorLayer (http://tensorlayer.readthedocs.io/en/latest/)
@tf.function:
compile static
graph

tf.function decorator
(implicitly) compiles
python functions to
static graph for better
performance

```python
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                 activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

@tf.function
def model_func(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

for t in range(50):
    with tf.GradientTape() as tape:
        y_pred, loss = model_func(x, y)
        gradients = tape.gradient(
            loss, model.trainable_variables)
    optimizer.apply_gradients(
        zip(gradients, model.trainable_variables))
```
@tf.function: compile static graph

Here we compare the forward-pass time of the same model under dynamic graph mode and static graph mode:

```python
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph: ", timeit.timeit(lambda: model_dynamic(x, y), number=10))
print("static graph: ", timeit.timeit(lambda: model_static(x, y), number=10))

dynamic graph:  0.025202492000000535
static graph: 0.0393222699998864
```

Ran on Google Colab, April 2020
@tf.function: compile static graph

Static graph is in theory faster than dynamic graph, but the performance gain depends on the type of model / layer / computation graph.

```python
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph: ", timeit.timeit(lambda: model_dynamic(x, y), number=10))
print("static graph: ", timeit.timeit(lambda: model_static(x, y), number=10))

dynamic graph: 0.02520249200000535
static graph: 0.03932226699998864
```
@tf.function:
compile static
graph

Static graph is in theory faster than dynamic graph, but the performance gain depends on the type of model / layer / computation graph.

```python
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph:", timeit.timeit(lambda: model_dynamic(x, y), number=1000))
print("static graph:", timeit.timeit(lambda: model_static(x, y), number=1000))
dynamic graph: 2.3648411540000325
static graph: 1.1723986679999143
```
Static vs Dynamic: Optimization

With static graphs, framework can **optimize** the graph for you before it runs!
class MyCell(torch.nn.Module):
    def __init__(self):
        super(MyCell, self).__init__()
        self.linear = torch.nn.Linear(4, 4)

    def forward(self, x, h):
        new_h = torch.tanh(self.linear(x) + h)
        return new_h, new_h

my_cell = MyCell()
x, h = torch.rand(3, 4), torch.rand(3, 4)
traced_cell = torch.jit.trace(my_cell, (x, h))
print(traced_cell.graph)
traced_cell(x, h)
<table>
<thead>
<tr>
<th>PyTorch</th>
<th>TensorFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Graphs</td>
<td>Dynamic: Eager</td>
</tr>
<tr>
<td>Static: TorchScript</td>
<td>Static: @tf.function</td>
</tr>
</tbody>
</table>
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
Dynamic Graph Applications

- Recurrent networks

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Dynamic Graph Applications

- Recurrent networks
- Recursive networks

The cat ate a big rat
Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular networks

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

Figure copyright Justin Johnson, 2017. Reproduced with permission.
Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks
- (Your creative idea here)
Model Parallel vs. Data Parallel

Model parallelism: split computation graph into parts & distribute to GPUs/nodes

Data parallelism: split minibatch into chunks & distribute to GPUs/nodes
PyTorch: Data Parallel

`nn.DataParallel`
Pro: Easy to use (just wrap the model and run training script as normal)
Con: Single process & single node. Can be bottlenecked by CPU with large number of GPUs (8+).

`nn.DistributedDataParallel`
Pro: Multi-nodes & multi-process training
Con: Need to hand-designate device and manually launch training script for each process / nodes.

Horovod ([https://github.com/horovod/horovod](https://github.com/horovod/horovod)): Supports both PyTorch and TensorFlow

TensorFlow: Data Parallel

tf.distributed.Strategy

```python
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
    model = tf.keras.Sequential(
        [tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
         tf.keras.layers.MaxPooling2D(),
         tf.keras.layers.Flatten(),
         tf.keras.layers.Dense(64, activation='relu'),
         tf.keras.layers.Dense(10)
    ])

model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

https://www.tensorflow.org/tutorials/distribute/keras
PyTorch vs. TensorFlow: Academia

# PyTorch vs. TensorFlow: Academia

<table>
<thead>
<tr>
<th>CONFERENCE</th>
<th>PT 2018</th>
<th>PT 2019</th>
<th>PT GROWTH</th>
<th>TF 2018</th>
<th>TF 2019</th>
<th>TF GROWTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVPR</td>
<td>82</td>
<td>280</td>
<td>240%</td>
<td>116</td>
<td>125</td>
<td>7.7%</td>
</tr>
<tr>
<td>NAACL</td>
<td>12</td>
<td>66</td>
<td>450%</td>
<td>34</td>
<td>21</td>
<td>-38.2%</td>
</tr>
<tr>
<td>ACL</td>
<td>26</td>
<td>103</td>
<td>296%</td>
<td>34</td>
<td>33</td>
<td>-2.9%</td>
</tr>
<tr>
<td>ICLR</td>
<td>24</td>
<td>70</td>
<td>192%</td>
<td>54</td>
<td>53</td>
<td>-1.9%</td>
</tr>
<tr>
<td>ICML</td>
<td>23</td>
<td>69</td>
<td>200%</td>
<td>40</td>
<td>53</td>
<td>32.5%</td>
</tr>
</tbody>
</table>

My Advice:

**PyTorch** is my personal favorite. Clean API, native dynamic graphs make it very easy to develop and debug. Can build model using the default API then compile static graph using JIT. Almost all academic research uses PyTorch.

**TensorFlow**’s syntax became a lot more intuitive after 2.0. Not perfect but has huge community and wide usage. Can use same framework for research and production. Probably use a higher-level wrapper (Keras, Sonnet, etc.).