Lecture 6: Hardware and Software

Deep Learning Hardware, Dynamic & Static Computational Graph, PyTorch & TensorFlow
Administrative

Assignment 1 is due tomorrow April 16th, 11:59pm.

Assignment 2 will be out tomorrow, due April 30th, 11:50 pm.

Project proposal due Monday April 19.
Administrative

Friday’s section topic: course project
two more layers to go: POOL/FC

Fei-Fei Li, Ranjay Krishna, Danfei Xu    Lecture 6 - 4    April 15, 2021
Convolution Layers (continue from last time)

3x3 CONV, 
stride=1, 
padding=1 
n_filters=64
Pooling layer
- makes the representations smaller and more manageable
- operates over each activation map independently:

![Diagram of pooling layer process]

224x224x64 → pool → 112x112x64

224 → downampling → 112
Single depth slice

max pool with 2x2 filters and stride 2
Pooling layer: summary

Let’s assume input is $W_1 \times H_1 \times C$
Conv layer needs 2 hyperparameters:
- The spatial extent $F$
- The stride $S$

This will produce an output of $W_2 \times H_2 \times C$ where:
- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$

Number of parameters: 0
Fully Connected Layer (FC layer)
- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks
Lecture 6:
Hardware and Software
Deep Learning Hardware, Dynamic & Static Computational Graph, PyTorch & TensorFlow
Where we are now...

Computational graphs

\[ f = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]
Where we are now...

Convolutional Neural Networks

Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1
Learning network parameters through optimization

# Vanilla Gradient Descent

```python
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * weights_grad  # perform parameter update
```
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)
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> (Intel Core i9-7900k)</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 GDDR6X</td>
<td>$1499</td>
<td>~35.6 TFLOPS FP32</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

\[ A \times B \]

\[ B \times C \]

\[ A \times C = \]

\[ \text{cuBLAS::GEMM (GEneral Matrix-to-matrix Multiply)} \]
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

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

cuDNN much faster than “unoptimized” CUDA

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)


Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 6 - 22

April 15, 2021
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></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 i7-7700k)</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>
<tr>
<td><strong>GPU</strong></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>(Data Center) NVIDIA A100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TPU</strong></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>
<tr>
<td>Google Cloud TPUv3</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

**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
- **Stanford CS 149**: [http://cs149.stanford.edu/fall20/](http://cs149.stanford.edu/fall20/)
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 Software
A zoo of frameworks!

Caffe (UC Berkeley)

Caffe2 (Facebook)
mostly features absorbed by PyTorch

Torch (NYU / Facebook)

PyTorch (Facebook)

Theano (U Montreal)

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)
We’ll focus on these

TensorFlow (Google)

PaddlePaddle (Baidu)

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

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

CNTK (Microsoft)

JAX (Google)

And others...
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
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

```
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)
```

Diagram:
- `x` and `y` connected by `*`
- `a` connected by `+` to `b`
- `c` connected by `Σ` to `b`
Computational Graphs

Numpy

```python
import numpy as np
cpython.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)

ggrad_c = 1.0
ggrad_b = ggrad_c * np.ones((N, D))
ggrad_a = ggrad_b.copy()
ggrad_z = ggrad_b.copy()
ggrad_x = ggrad_a * y
ggrad_y = ggrad_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

```
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

```
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)

g = c 1.0
g_b = grad_c * np.ones((N, D))
g_a = grad_b.copy()
gz = grad_b.copy()
gx = grad_a * y
gy = 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 1.7**

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

Forward pass: compute predictions and loss
PyTorch: Tensors

Backward pass: manually compute gradients

```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

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

Gradient descent step on weights
PyTorch: Tensors

To run on GPU, just use a different device!
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.

```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_()
```
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
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
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, just like cache objects from A2

```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, just like cache objects from A2

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)
```

```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.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

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

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 Python function.

```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.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: 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

Forward pass: feed data to model, and compute loss

torch.nn.functional has useful helpers like loss functions

```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

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()
```

Make gradient step on each model parameter (with gradients disabled)
Use an **optimizer** for different update rules.
PyTorch: optim

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

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()
```

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
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

```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

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()
```
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: torch.utils.tensorboard

A python wrapper around Tensorflow’s web-based visualization tool.
PyTorch: Computational Graphs

input image

loss

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

```
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

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

Search for path between loss and \( w_1, w_2 \) (for backprop) AND perform computation

```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

```
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()
```

Throw away the graph, backprop path, and rebuild it from scratch on every iteration.
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
PyTorch: **Dynamic** Computation Graphs

Search for path between loss and $w_1$, $w_2$ (for backprop) AND perform computation

```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

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

```python
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)

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

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))

g = 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, g[0], g[1]], feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

(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

```
assert(tf.executing_eagerly())
```

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 2.0+:
“Eager” Mode by default

```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))
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 1.13

```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))

with tf.Session() as sess:
    sess.run([loss, grad_w1, grad_w2],
              feed_dict={x: np.random.randn(N, D),
                          w1: np.random.randn(D, H),
                          w2: np.random.randn(H, D),
                          y: np.random.randn(N, D),
              })
```
TensorFlow: 2.0+ vs. pre-2.0

TensorFlow 2.0+:
“Eager” Mode by default

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

assert tf.executing_eagerly()
```

TensorFlow 1.13

```python
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

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) gets traced for computing gradient later.

```
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

```
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

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

```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

Backward pass

\[
N, D, H = 64, 1000, 100
\]

\[
x = \text{tf.convert_to_tensor}(\text{np.random.randn}(N, D), \text{np.float32})
\]

\[
y = \text{tf.convert_to_tensor}(\text{np.random.randn}(N, D), \text{np.float32})
\]

\[
w1 = \text{tf.Variable}(\text{tf.random.uniform}((D, H))) \quad \# \text{weights}
\]

\[
w2 = \text{tf.Variable}(\text{tf.random.uniform}((H, D))) \quad \# \text{weights}
\]

\[
\text{with} \ \text{tf.GradientTape()} \ \text{as} \ \text{tape}:
\]

\[
h = \text{tf.maximum}(\text{tf.matmul}(x, w1), 0)
\]

\[
y_{\text{pred}} = \text{tf.matmul}(h, w2)
\]

\[
diff = y_{\text{pred}} - y
\]

\[
\text{loss} = \text{tf.reduce_mean}(\text{tf.reduce_sum}(\text{diff} ** 2, \text{axis}=1))
\]

\[
\text{gradients} = \text{tape.gradient}(\text{loss, [w1, w2]})
\]
Train the network: Run the training step over and over, use gradient to update weights

```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

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

```
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])
```
TensorFlow: Optimizer

Can use an optimizer to compute gradients and update weights

```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)

learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        h = tf.nn.relu(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]))
```
Keras: High-Level Wrapper

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

(Used to be third-party, now merged into TensorFlow)

```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)

losses = []
for t in range(50):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = tf.losses.MeanSquaredError()(y_pred, y)
        gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```
Keras: High-Level Wrapper

Define model as a sequence of layers

Get output by calling the model

Apply gradient to all trainable variables (weights) in the model

```
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)

losses = []
for t in range(50):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = tf.losses.MeanSquaredError()(y_pred, y)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```
Keras: High-Level Wrapper

Keras can handle the training loop for you!

```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)
model.compile(loss=tf.keras.losses.MeanSquaredError(),
    optimizer=optimizer)

history = model.fit(x, y, epochs=50, batch_size=N)
```
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))
```

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

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

Ran on Google Colab, April 2020

```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!

The graph you wrote

```
Conv
ReLU
Conv
ReLU
Conv
ReLU
```

Equivalent graph with fused operations

```
Conv+ReLU
Conv+ReLU
Conv+ReLU
```
Static PyTorch: ONNX Support

You can export a PyTorch model to ONNX ([Open Neural Network Exchange](https://onnx.ai/)).

Run the graph on a dummy input, and save the graph to a file.

Will only work if your model doesn’t actually make use of dynamic graph - must build same graph on every forward pass.

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

dummy_input = torch.randn(N, D_in)
torch.onnx.export(model, dummy_input,
                 'model.proto',
                 verbose=True)
```
Static PyTorch: ONNX Support

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out)

dummy_input = torch.randn(N, D_in)
torch.onnx.export(model, dummy_input,
                  'model.proto',
                  verbose=True)
```

After exporting to ONNX, can run the PyTorch model in Caffe2
ONNX is an open-source standard for neural network models

Goal: Make it easy to train a network in one framework, then run it in another framework

Supported by PyTorch, Caffe2, Microsoft CNTK, Apache MXNet (3rd-party support Tensorflow)

https://github.com/onnx/onnx
Static PyTorch: TorchScript

```python
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)
```

Build static graph with `torch.jit.trace`
PyTorch vs TensorFlow, Static vs Dynamic

**PyTorch**
- Dynamic Graphs
- Static: ONNX, TorchScript

**TensorFlow**
- Dynamic: Eager
- Static: `@tf.function`
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.


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>CONFERENCES</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>

PyTorch vs. TensorFlow: Industry (202)

- No official survey / study on the comparison.
- A quick search on a job posting website turns up 2389 search results for TensorFlow and 1366 for PyTorch.
- The trend is unclear. Industry is also known to be slower on adopting new frameworks.
- TensorFlow mostly dominates mobile deployment / embedded systems.
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. Lots of research repositories are built on 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.).
Next Time:
Training Neural Networks