Lecture 9: Hardware and Software
Assignment 3 Released

Assignment 3 is released! It covers:

• Fully-connected networks
• Dropout
• Update rules: SGD+Momentum, RMSprop, Adam
• Convolutional networks
• Batch normalization

Due **Friday October 9, 11:59pm**
(Website originally said 10/16 – this was a typo!)
Deep Learning Hardware
Inside a computer
Inside a computer

GPU: “Graphics Processing Unit”
Inside a computer

CPU: “Central Processing Unit”

GPU: “Graphics Processing Unit”

This image is licensed under CC-BY 2.0

This image is in the public domain

This image copyright 2017, Justin Johnson
NVIDIA vs AMD
NVIDIA vs AMD
GFLOPs per Dollar

- CPU
- GPU FP32

![Graph showing the GFLOPs per Dollar over time with data points for CPU and GPU FP32.]
# CPU vs GPU

<table>
<thead>
<tr>
<th></th>
<th>Cores</th>
<th>Clock Speed (GHz)</th>
<th>Memory</th>
<th>Price</th>
<th>TFLOP/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ryzen Threadripper 3970X</td>
<td>64 (128 threads with hyperthreading)</td>
<td>3.7 (4.5 boost)</td>
<td>System RAM</td>
<td>$1999</td>
<td>~6.9 FP32</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVIDIA RTX 3090</td>
<td>10496</td>
<td>1.4 (1.7 boost)</td>
<td>24 GB GDDR6X</td>
<td>$1499</td>
<td>~35.6 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.
Inside a GPU: RTX 3090

12x 2GB memory modules
Inside a GPU: RTX 3090
Inside a GPU: RTX 3090

12x 2GB memory modules

Processor
Inside a GPU: RTX 3090
Inside a GPU: RTX 3090

82 Streaming multiprocessors (SMs)
Inside a GPU: RTX 3090
Inside a GPU: RTX 3090

64 FP32 cores per SM
64 INT32/FP32 cores per SM
Inside a GPU: RTX 3090

- 64 FP32 cores per SM
- 64 INT32/FP32 cores per SM
- 35.6 FP32 TFLOP/sec

Multiply:
- 82 SM
- 128 FP32 core/SM
- 2 FLOP/cycle
- 1.7 GCycle / sec
= 35.6 TFLOP/sec
Inside a GPU: RTX 3090

64 FP32 cores per SM
64 INT32/FP32 cores per SM
35.6 FP32 TFLOP/sec
4 Tensor Core per SM

Special hardware!

Let A, B, C be matrices (A 4x4, B,C 4x8).
Compute AB+C in one clock cycle (256 FLOP)
Inside a GPU: RTX 3090

- 64 FP32 cores per SM
- 64 INT32/FP32 cores per SM
- 35.6 FP32 TFLOP/sec
- 4 Tensor Core per SM
- 35.6 FP32 TFLOP/sec

Multiply:
- 82 SM
- 4 Tensor Core/SM
- 256 FLOP/cycle
- 1.7 GCycle / sec
= 142 TFLOP/sec
## CPU vs GPU

<table>
<thead>
<tr>
<th></th>
<th>Cores</th>
<th>Clock Speed (GHz)</th>
<th>Memory</th>
<th>Price</th>
<th>TFLOP/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ryzen 3970X</td>
<td>64</td>
<td>3.7 (4.5 boost)</td>
<td>System RAM</td>
<td>$1999</td>
<td>~6.9 FP32</td>
</tr>
<tr>
<td></td>
<td>(128 threads with hyperthreading)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVIDIA RTX 3090</td>
<td>10496</td>
<td>1.4 (1.7 boost)</td>
<td>24 GB GDDR6X</td>
<td>$1499</td>
<td>~35.6 FP32 ~142 TFLOP with Tensor core</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.
GFLOPs per Dollar

GFLOPs per dollar

Date:
- 9/2002
- 5/2005
- 2/2008
- 11/2010
- 8/2013
- 5/2016
- 2/2019
- 10/2021

GFLOPs per dollar

Time

GFLOPs per dollar

GFLOPs per dollar

CPU

GPU FP32

GPU Tensor Core
Example: Matrix Multiplication

\[ A \times B \]
\[ B \times C \]
\[ A \times C \]

Perfect for GPUs! All output elements are independent, can be trivially parallelized.
Programming GPUs

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

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

• EECS 598.009: Applied GPU Programming
Scaling up: Typically 8 GPUs per server

NVIDIA DGX-1: 8x V100 GPUs
Google Tensor Processing Units (TPU)

Special hardware for matrix multiplication, similar to NVIDIA Tensor Cores; also runs in mixed precision (bfloat16)

Cloud TPU v2-8
180 TFLOP/sec
64 GB HBM memory
$6 / hour
(free on Colab!)
Google Tensor Processing Units (TPU)

Cloud TPU v2-8
- 180 TFLOP/sec
- 64 GB HBM memory
- $6 / hour
(free on Colab!)

Cloud TPU v2 Pod
- 16x TPU-v2-8
- 11.5 PFLOPs
- $384 / hour

Cloud TPU v2 Pod
- 16x TPU-v2-8
- 11.5 PFLOPs
- $384 / hour
Cloud TPU v3-8
420 TFLOP/sec
128 GB HBM memory
$8 / hour
Google Tensor Processing Units (TPU)

Cloud TPU v3-8
- 420 TFLOP/sec
- 128 GB HBM memory
- $8 / hour

Cloud TPU v3 Pod
- 256 TPU-v3
- 107 PFLOPs
Cloud TPU v3-8
420 TFLOP/sec
128 GB HBM memory
$8 / hour

Cloud TPU v3 Pod
256 TPU-v3
107 PFLOPs
Contact sales rep for pricing

Cloud TPU v3 image released under a CC-SA 4.0 International license
Deep Learning Software
A zoo of frameworks!

Caffe (UC Berkeley) → Caffe2 (Facebook)

Torch (NYU / Facebook) → PyTorch (Facebook)

Theano (U Montreal) → TensorFlow (Google)

PaddlePaddle (Baidu)

MXNet (Amazon)

CNTK (Microsoft)

Chainer

JAX (Google)
A zoo of frameworks!

Caffe (UC Berkeley)

Caffe2 (Facebook)

PyTorch (Facebook)

TensorFlow (Google)

We’ll focus on these

Chainer

PaddlePaddle (Baidu)

MXNet (Amazon)

CNTK (Microsoft)

JAX (Google)

Torch (NYU / Facebook)

Theano (U Montreal)
Recall: Computational Graphs

\[ f = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]
The point of deep learning frameworks

1. Allow rapid prototyping of new ideas
2. Automatically compute gradients for you
3. Run it all efficiently on GPU (or TPU)
PyTorch
PyTorch: Versions

For this class we are using **PyTorch version 1.6** (Released July 2020)

Be careful if you are looking at older PyTorch code – the API changed a lot before 1.0 (0.3 to 0.4 had big changes!)
PyTorch: Fundamental Concepts

**Tensor**: Like a numpy array, but can run on GPU

**Autograd**: Package for building computational graphs out of Tensors, and automatically computing gradients

**Module**: A neural network layer; may store state or learnable weights
PyTorch: Fundamental Concepts

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

Autograd: Package for building computational graphs out of Tensors, and automatically computing gradients

Module: A neural network layer; may store state or learnable weights
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(H, D_in, 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(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

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!

```
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
```
Creating Tensors with `requires_grad=True` enables autograd

Operations on Tensors with `requires_grad=True` cause PyTorch to build a computational 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_
```
We will not want gradients (of loss) with respect to data.

Do want gradients with respect to 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)
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

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

```
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_
```
Every operation on a tensor with `requires_grad=True` will add to the computational graph, and the resulting tensors will also have `requires_grad=True`. 

```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_()
```
Every operation on a tensor with `requires_grad=True` will add to the computational graph, and the resulting tensors will also have `requires_grad=True`
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

```
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

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_()}
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

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

After backward finishes, gradients are **accumulated** into w1.grad and w2.grad and the graph is destroyed.
After backward finishes, gradients are **accumulated** into `w1.grad` and `w2.grad` and the graph is destroyed.

Make gradient step on weights.
After backward finishes, gradients are accumulated into w1.grad and w2.grad and the graph is destroyed.

Set gradients to zero – forgetting this is a common bug!
After backward finishes, gradients are accumulated into w1.grad and w2.grad and the graph is destroyed.

Tell PyTorch not to build a graph for these operations.
PyTorch: New functions

Can define new operations using Python 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)
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 = torch.sigmoid(x.mm(w1)).mm(w2)
loss = (y_pred - y).pow(2).sum()

loss.backward()
if t % 50 == 0:
    print(t, loss.item())

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

def sigmoid(x):
    return 1.0 / (1.0 + (-x).exp())
PyTorch: New functions

Can define new operations using Python functions

```python
def sigmoid(x):
    return 1.0 / (1.0 + (-x).exp())
```

When our function runs, it will add to the graph

Gradients computed with 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)
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 = sigmoid(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    loss.backward()
    if t % 50 == 0:
        print(t, loss.item())

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

Can define new operations using Python functions

```python
def sigmoid(x):
    return 1.0 / (1.0 + (-x).exp())
```

When our function runs, it will add to the graph

Gradients computed with autograd

```python
class Sigmoid(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        y = 1.0 / (1.0 + (-x).exp())
        ctx.save_for_backward(y)
        return y

    @staticmethod
    def backward(ctx, grad_y):
        y, = ctx.saved_tensors
        grad_x = grad_y * y * (1.0 - y)
        return grad_x

def sigmoid(x):
    return Sigmoid.apply(x)
```

Recall:

$$
\frac{\partial}{\partial x} \sigma(x) = (1 - \sigma(x))\sigma(x)
$$
PyTorch: New functions

Can define new operations using Python functions

```python
def sigmoid(x):
    return 1.0 / (1.0 + (-x).exp())
```

When our function runs, it will add to the graph

Gradients computed with autograd

Now when our function runs, it adds one node to the graph!

Define new autograd operators by subclassing `Function`, define forward and backward

```python
class Sigmoid(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        y = 1.0 / (1.0 + (-x).exp())
        ctx.save_for_backward(y)
        return y

    @staticmethod
    def backward(ctx, grad_y):
        y, = ctx.saved_tensors
        grad_x = grad_y * y * (1.0 - y)
        return grad_x

def sigmoid(x):
    return Sigmoid.apply(x)
```
PyTorch: New functions

Can define new operations using Python functions

```python
def sigmoid(x):
    return 1.0 / (1.0 + (-x).exp())
```

When our function runs, it will add to the graph

Gradients computed with autograd

Define new autograd operators by subclassing `Function`, define forward and backward

```python
class Sigmoid(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        y = 1.0 / (1.0 + (-x).exp())
        ctx.save_for_backward(y)
        return y
    @staticmethod
    def backward(ctx, grad_y):
        y, = ctx.saved_tensors
        grad_x = grad_y * y * (1.0 - y)
        return grad_x

def sigmoid(x):
    return Sigmoid.apply(x)
```

In practice this is pretty rare – in most cases Python functions are good enough
PyTorch: nn

Higher-level wrapper for working with neural nets

Use this! It will make your life easier
PyTorch: nn

Object-oriented API: Define model object as sequence of layers objects, each of which holds weight tensors.

```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

Forward pass: Feed data to model and compute loss

```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

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

```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()
```

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
After computing gradients, use optimizer to update and zero gradients.
A PyTorch **Module** is a neural net layer; it inputs and outputs Tensors.

Modules can contain weights or other modules.

Very common to define your own models or layers as custom Modules.
PyTorch: nn
Defining 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
Defining Modules

Initializer sets up two children (Modules can contain modules)
Define forward pass using child modules and tensor operations

No need to define backward - autograd will handle it
PyTorch: nn Defining 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()
```
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 Defining Modules

Stack multiple instances of the component in a sequential

Very easy to quickly build complex network architectures!

```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()
```
A **DataLoader** wraps a **Dataset** and provides minibatching, shuffling, multithreading, for you.

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

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

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

loader = DataLoader(TensorDataset(x, y), batch_size=8)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
for epoch in range(20):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred, y_batch)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```
PyTorch: DataLoaders

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

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

loader = DataLoader(TensorDataset(x, y), batch_size=8)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
for epoch in range(20):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

Iterate over loader to form minibatches
PyTorch: DataLoaders

```python
import torchrom torch.utils.data import TensorDataset, DataLoader

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

loader = DataLoader(TensorDataset(x, y), batch_size=8)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
for epoch in range(20):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred, y_batch)

        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: 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

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

Perform backprop, throw away graph

\[
\text{loss} = (y_{\text{pred}} - y)^2 \text{.sum()}
\]

```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

Perform backprop, throw away 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()
```
Build graph data structure 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()
```

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()
```

Perform backprop, throw away graph
Dynamic graphs let you use regular Python control flow during the forward pass!

```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)
w2a = torch.randn(H, D_out, requires_grad=True)
w2b = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
prev_loss = 5.0
for t in range(500):
    w2 = w2a if prev_loss < 5.0 else w2b
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    loss.backward()
    prev_loss = loss.item()
```
Dynamic graphs let you use regular Python control flow during the forward pass!

Initialize two different weight matrices for second layer

```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)
w2a = torch.randn(H, D_out, requires_grad=True)
w2b = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
prev_loss = 5.0
for t in range(500):
    w2 = w2a if prev_loss < 5.0 else w2b
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    loss.backward()
    prev_loss = loss.item()
```
Dynamic graphs let you use regular Python control flow during the forward pass!

Decide which one to use at each layer based on loss at previous iteration

(this model doesn’t make sense! Just a simple dynamic example)
Alternative: **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)
```
PyTorch: Static Graphs with JIT

Define model as a Python function

```python
import torch

def model(x, y, w1, w2a, w2b, prev_loss):
    w2 = w2a if prev_loss < 5.0 else w2b
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    return loss

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)
w2a = torch.randn(H, D_out, requires_grad=True)
w2b = torch.randn(H, D_out, requires_grad=True)

graph = torch.jit.script(model)

prev_loss = 5.0
learning_rate = 1e-6
for t in range(500):
    loss = graph(x, y, w1, w2a, w2b, prev_loss).
    loss.backward()
    prev_loss = loss.item()
```
PyTorch: Static Graphs with JIT

Just-In-Time compilation:
- Introspect the source code of the function, **compile** it into a graph object.
- Lots of magic here!

```python
import torch

def model(x, y, w1, w2a, w2b, prev_loss):
    w2 = w2a if prev_loss < 5.0 else w2b
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    return loss

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)
w2a = torch.randn(H, D_out, requires_grad=True)
w2b = torch.randn(H, D_out, requires_grad=True)

graph = torch.jit.script(model)

prev_loss = 5.0
learning_rate = 1e-6
for t in range(500):
    loss = graph(x, y, w1, w2a, w2b, prev_loss).
    loss.backward()
    prev_loss = loss.item()
```
Graph includes a conditional node to handle both cases!

```python
import torch

def model(x, y, w1, w2a, w2b, prev_loss):
    w2 = w2a if prev_loss < 5.0 else w2b
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    return loss

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)
w2a = torch.randn(H, D_out, requires_grad=True)
w2b = torch.randn(H, D_out, requires_grad=True)

graph = torch.jit.script(model)

prev_loss = 5.0
learning_rate = 1e-6
for t in range(500):
    loss = graph(x, y, w1, w2a, w2b, prev_loss).
    loss.backward()
    prev_loss = loss.item()
```
PyTorch: Static Graphs with JIT

```python
import torch

def model(x, y, w1, w2a, w2b, prev_loss):
    w2 = w2a if prev_loss < 5.0 else w2b
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    return loss

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)
w2a = torch.randn(H, D_out, requires_grad=True)
w2b = torch.randn(H, D_out, requires_grad=True)

graph = torch.jit.script(model)

prev_loss = 5.0
learning_rate = 1e-6
for t in range(500):
    loss = graph(x, y, w1, w2a, w2b, prev_loss)
    loss.backward()
    prev_loss = loss.item()
```

Use our compiled graph object at each forward pass
Even easier: add **annotation** to function, Python function compiled to a graph when it is defined

Calling function uses graph

```python
import torch

@torch.jit.script
def model(x, y, w1, w2a, w2b, prev_loss):
    w2 = w2a if prev_loss < 5.0 else w2b
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()
    return loss

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)
w2a = torch.randn(H, D_out, requires_grad=True)
w2b = torch.randn(H, D_out, requires_grad=True)

prev_loss = 5.0
learning_rate = 1e-6
for t in range(500):
    loss = model(x, y, w1, w2a, w2b, prev_loss)
    loss.backward()
    prev_loss = loss.item()
```
Static vs Dynamic Graphs: Optimization

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

The graph you wrote:
- Conv
- ReLU
- Conv
- ReLU
- Conv
- ReLU

Equivalent graph with **fused operations**:
- Conv+ReLU
- Conv+ReLU
- Conv+ReLU
Static vs Dynamic Graphs: Serialization

**Static**

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

e.g. train model in Python, deploy in C++

**Dynamic**

Graph building and execution are intertwined, so always need to keep code around
Static vs Dynamic Graphs: Debugging

**Static**
Lots of indirection between the code you write and the code that runs – can be hard to debug, benchmark, etc

**Dynamic**
The code you write is the code that runs! Easy to reason about, debug, profile, etc
Model structure depends on the input:
- Recurrent Networks
Dynamic Graph Applications

Model structure depends on the input:
- Recurrent Networks
- Recursive Networks

Dynamic Graph Applications

Model structure depends on the input:
- Recurrent Networks
- Recursive Networks
- Modular Networks

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

Model structure depends on the input:
- Recurrent Networks
- Recursive Networks
- Modular Networks
- (Your idea here!)

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

TensorFlow 1.0
- Final release: 1.15.3
- Default: **static graphs**
- Optional: dynamic graphs (eager mode)

TensorFlow 2.0
- Current release: 2.3.1
  - Released 9/24
- Default: **dynamic graphs**
- Optional: static graphs
import numpy as np
import tensorflow as tf

(Assume imports at the top of each snippet)
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))
gradients = 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, gradients],
feed_dict=values)
loss_val, grad_w1_val, grad_w2_val = out
```
Create TensorFlow Tensors for data and weights

Weights need to be wrapped in tf.Variable so we can mutate them

```python
import tensorflow as tf

N, Din, H, Dout = 16, 1000, 100, 10

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))
w1 = tf.Variable(tf.random.normal((Din, H)))
w2 = tf.Variable(tf.random.normal((H, Dout)))

learning_rate = 1e-6
for t in range(1000):
    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 = tape.gradient(loss, [w1, w2])

    w1.assign(w1 - learning_rate * grad_w1)
w2.assign(w2 - learning_rate * grad_w2)
```
import tensorflow as tf

N, Din, H, Dout = 16, 1000, 100, 10

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))
w1 = tf.Variable(tf.random.normal((Din, H)))
w2 = tf.Variable(tf.random.normal((H, Dout)))

learning_rate = 1e-6
for t in range(1000):
    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 = tape.gradient(loss, [w1, w2])

    w1.assign(w1 - learning_rate * grad_w1)
    w2.assign(w2 - learning_rate * grad_w2)
import tensorflow as tf

N, Din, H, Dout = 16, 1000, 100, 10

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))
w1 = tf.Variable(tf.random.normal((Din, H)))
w2 = tf.Variable(tf.random.normal((H, Dout)))

learning_rate = 1e-6
for t in range(1000):
    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 = tape.gradient(loss, [w1, w2])
    w1.assign(w1 - learning_rate * grad_w1)
    w2.assign(w2 - learning_rate * grad_w2)
Gradient descent step, update weights

```python
import tensorflow as tf

N, Din, H, Dout = 16, 1000, 100, 10

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))
w1 = tf.Variable(tf.random.normal((Din, H)))
w2 = tf.Variable(tf.random.normal((H, Dout)))

learning_rate = 1e-6
for t in range(1000):
    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 = tape.gradient(loss, [w1, w2])

    w1.assign(w1 - learning_rate * grad_w1)
w2.assign(w2 - learning_rate * grad_w2)
```
TensorFlow 2.0: Static Graphs

Define a function that implements forward, backward, and update.

Annotating with `tf.function` will compile the function into a graph! (similar to `torch.jit.script`)

```python
@tf.function
def step(x, y, w1, w2):
    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 = tape.gradient(loss, [w1, w2])

        w1.assign(w1 - learning_rate * grad_w1)
        w2.assign(w2 - learning_rate * grad_w2)

        return loss
```

```python
N, Din, H, Dout = 16, 1000, 100, 10
x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))
w1 = tf.Variable(tf.random.normal((Din, H)))
w2 = tf.Variable(tf.random.normal((H, Dout)))

learning_rate = 1e-6
for t in range(1000):
    loss = step(x, y, w1, w2)
```
TensorFlow 2.0: Static Graphs

Define a function that implements forward, backward, and update.

Annotating with `tf.function` will compile the function into a graph! (similar to `torch.jit.script`)

(note TF graph can include gradient computation and update, unlike PyTorch)

```python
@tf.function
def step(x, y, w1, w2):
    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 = tape.gradient(loss, [w1, w2])

        w1.assign(w1 - learning_rate * grad_w1)
        w2.assign(w2 - learning_rate * grad_w2)

        return loss

N, Din, H, Dout = 16, 1000, 100, 10
x = tf.random.normal(((N, Din)))
y = tf.random.normal(((N, Dout)))
w1 = tf.Variable(tf.random.normal(((Din, H))))
w2 = tf.Variable(tf.random.normal(((H, Dout))))

learning_rate = 1e-6
for t in range(1000):
    loss = step(x, y, w1, w2)
```
TensorFlow 2.0: Static Graphs

Call the compiled step function in the training loop

```python
@tf.function
def step(x, y, w1, w2):
    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 = tape.gradient(loss, [w1, w2])

    w1.assign(w1 - learning_rate * grad_w1)
    w2.assign(w2 - learning_rate * grad_w2)

    return loss

N, Din, H, Dout = 16, 1000, 100, 10
x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))
w1 = tf.Variable(tf.random.normal((Din, H)))
w2 = tf.Variable(tf.random.normal((H, Dout)))

learning_rate = 1e-6
for t in range(1000):
    loss = step(x, y, w1, w2)
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

N, Din, H, Dout = 16, 1000, 100, 10

model = Sequential()
model.add(InputLayer(input_shape=(Din,)))
model.add(Dense(units=H, activation='relu'))
model.add(Dense(units=Dout))
params = model.trainable_variables

loss_fn = tf.keras.losses.MeanSquaredError()
opt = tf.keras.optimizers.SGD(learning_rate=1e-6)

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))

for t in range(1000):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        grads = tape.gradient(loss, params)
        opt.apply_gradients(zip(grads, params))
Keras: High-level API

Object-oriented API: build the model as a stack of layers

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

N, Din, H, Dout = 16, 1000, 100, 10

model = Sequential()
model.add(InputLayer(input_shape=(Din,)))
model.add(Dense(units=H, activation='relu'))
model.add(Dense(units=Dout))
params = model.trainable_variables

loss_fn = tf.keras.losses.MeanSquaredError()
opt = tf.keras.optimizers.SGD(learning_rate=1e-6)

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))

for t in range(1000):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        grads = tape.gradient(loss, params)
        opt.apply_gradients(zip(grads, params))
```
Keras: High-level API

Keras gives you common loss functions and optimization algorithms

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

N, Din, H, Dout = 16, 1000, 100, 10

model = Sequential()
model.add(InputLayer(input_shape=(Din,)))
model.add(Dense(units=H, activation='relu'))
model.add(Dense(units=Dout))
params = model.trainable_variables

loss_fn = tf.keras.losses.MeanSquaredError()
opt = tf.keras.optimizers.SGD(learning_rate=1e-6)

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))

for t in range(1000):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        grads = tape.gradient(loss, params)
        opt.apply_gradients(zip(grads, params))
```
Keras: High-level API

Forward pass:
Compute loss, build graph

Backward pass:
compute gradients

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

N, Din, H, Dout = 16, 1000, 100, 10

model = Sequential()
model.add(InputLayer(input_shape=(Din,)))
model.add(Dense(units=H, activation='relu'))
model.add(Dense(units=Dout))
params = model.trainable_variables

loss_fn = tf.keras.losses.MeanSquaredError()
opt = tf.keras.optimizers.SGD(learning_rate=1e-6)

x = tf.random.normal(((N, Din)))
y = tf.random.normal(((N, Dout)))

for t in range(1000):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        grads = tape.gradient(loss, params)
        opt.apply_gradients(zip(grads, params))
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

N, Din, H, Dout = 16, 1000, 100, 10

model = Sequential()
model.add(InputLayer(input_shape=(Din,)))
model.add(Dense(units=H, activation='relu'))
model.add(Dense(units=Dout))
params = model.trainable_variables

loss_fn = tf.keras.losses.MeanSquaredError()
opt = tf.keras.optimizers.SGD(learning_rate=1e-6)

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))

for t in range(1000):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        grads = tape.gradient(loss, params)
    opt.apply_gradients(zip(grads, params))
Define a function that returns the loss

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

N, Din, H, Dout = 16, 1000, 100, 10

model = Sequential()
model.add(InputLayer(input_shape=(Din,)))
model.add(Dense(units=H, activation='relu'))
model.add(Dense(units=Dout))

params = model.trainable_variables

loss_fn = tf.keras.losses.MeanSquaredError()
opt = tf.keras.optimizers.SGD(learning_rate=1e-6)

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))

def step():
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    return loss

for t in range(1000):
    opt.minimize(step, params)
```
Keras: High-level API

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense

N, Din, H, Dout = 16, 1000, 100, 10

model = Sequential()
model.add(InputLayer(input_shape=(Din,)))
model.add(Dense(units=H, activation='relu'))
model.add(Dense(units=Dout))

params = model.trainable_variables

loss_fn = tf.keras.losses.MeanSquaredError()
opt = tf.keras.optimizers.SGD(learning_rate=1e-6)

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))

def step():
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    return loss

for t in range(1000):
    opt.minimize(step, params)
```

Optimizer computes gradients and updates parameters.
TensorBoard

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

Also works with PyTorch: torch.utils.tensorboard
<table>
<thead>
<tr>
<th><strong>PyTorch</strong></th>
<th><strong>TensorFlow 1.0</strong></th>
<th><strong>TensorFlow 2.0</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>My personal favorite</td>
<td>Static graphs by default</td>
<td>Dynamic by default</td>
</tr>
<tr>
<td>Clean, imperative API</td>
<td>Can be confusing to debug</td>
<td>Standardized on Keras API</td>
</tr>
<tr>
<td>Easy dynamic graphs for debugging</td>
<td>API a bit messy</td>
<td>API still confusing</td>
</tr>
<tr>
<td>JIT allows static graphs for production</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cannot use TPUs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not easy to deploy on mobile</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary: Hardware

CPU

GPU

TPU
Summary: Software

Static Graphs vs Dynamic Graphs

PyTorch vs TensorFlow
Next time:
Training Neural Networks