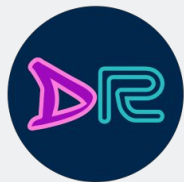


ROB 430/599: Deep Learning for Robot Perception (DeepRob)

Lecture 15: Deep Learning Software

03/09/2026



<https://deeprob.org/w26/>

Today

- Feedback and Recap (5min)
- DL Frameworks (1hr)
 - PyTorch
 - TensorFlow
 - Keras
- Announcements / Reminders (5-10min)
- Summary and Takeaways (5min)

A zoo of frameworks!

*Caffe2 is now a part of PyTorch

Caffe
(UC Berkeley) → Caffe2
(Facebook)

Torch
(NYU / Facebook) → PyTorch
(Facebook)

<https://pytorch.org/>

Theano
(U Montreal) → TensorFlow
(Google)

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

Darknet
(Redmon)

Chainer

MXNet
(Amazon)

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

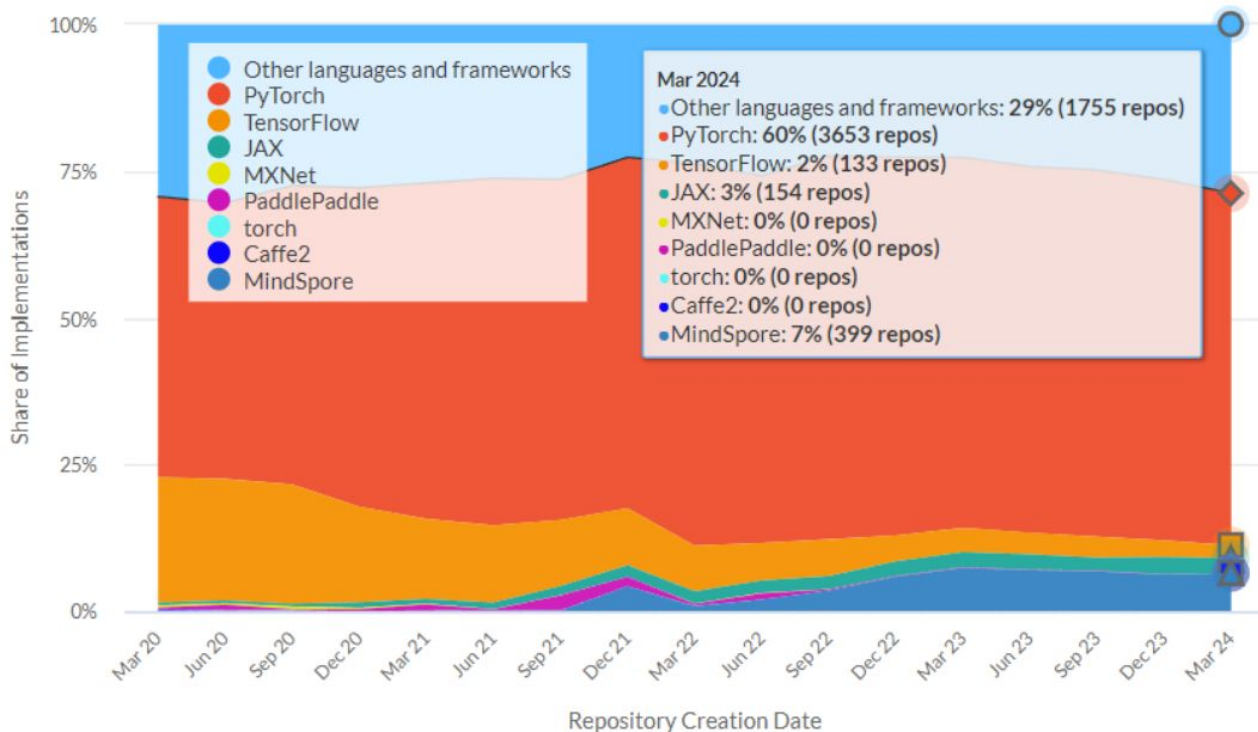
CNTK
(Microsoft)

PaddlePaddle
(Baidu)

JAX
(Google)

A zoo of frameworks!

Paper Implementations grouped by framework Source: <https://paperswithcode.com/trends>



Motivation for DL Frameworks

- Allow rapid prototyping of new ideas
- Automatically compute gradients for you
- Run it all efficiently on GPU or TPU hardware

PyTorch

PyTorch 2.0+

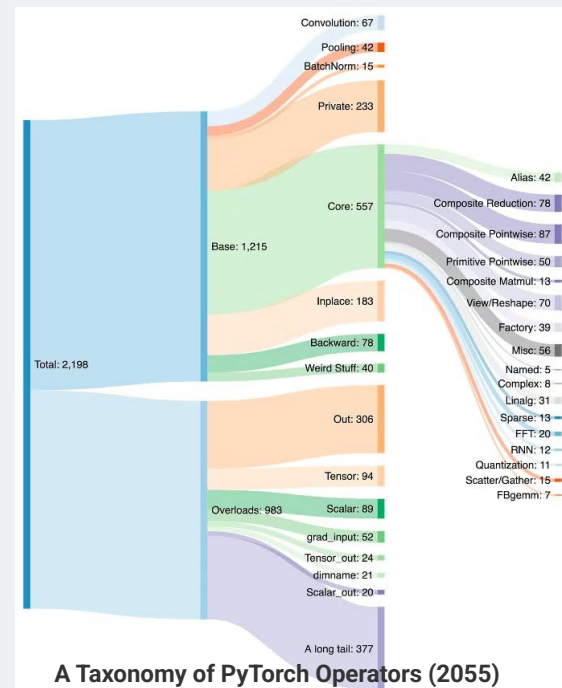
<https://pytorch.org/blog/pytorch-2-10-release-blog/>

Most Recent Release Jan.21, 2026

PyTorch 2.10

The PyTorch 2.10 release features the following changes:

- **Python 3.14** support for `torch.compile()`. Python 3.14t (freethreaded build) is experimentally supported as well.
- Reduced kernel launch overhead with **combo-kernels** horizontal fusion in `torchinductor`
- A new **`varlen_attn()`** op providing support for ragged and packed sequences
- Efficient eigenvalue decompositions with **DnXgeev**
- **Intel GPU** performance optimizations and feature enhancements
- `torch.compile()` now respects **`use_deterministic_mode`**
- **DebugMode** for tracking dispatched calls and debugging numerical divergence – This makes it simpler to track down subtle numerical bugs.



PyTorch: Fundamental Concepts

- **Tensor**: Like a numpy array, but can run on GPU P0,P1,P2
- **Autograd**: Package for building computational graphs out of Tensors, and automatically computing gradients P3, P4, Final Project
- **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

```
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
```

Create random tensors for data and weights

```
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
```

Forward pass: compute predictions and loss



```
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
```

Backward pass: manually
compute gradients



```
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



```
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
```

To run on GPU, just
use a different device!



```
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: Autograd

AhaSlide: What happens with this code?



<https://ahaslides.com/7WDGH>

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

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

Creating Tensors with `requires_grad=True` enables autograd

Operations on Tensors with `requires_grad=True` cause PyTorch to build a computational graph

```
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

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

Do want gradients with
respect to weights

```
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



Q: Let's draw the computational graph!

Compute gradients with respect to all inputs that have `requires_grad=True`!

```
import torch

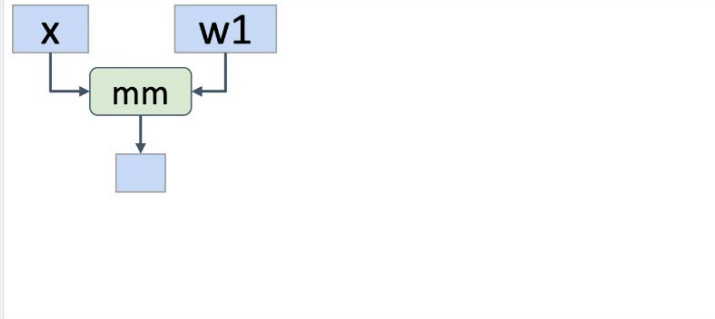
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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



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

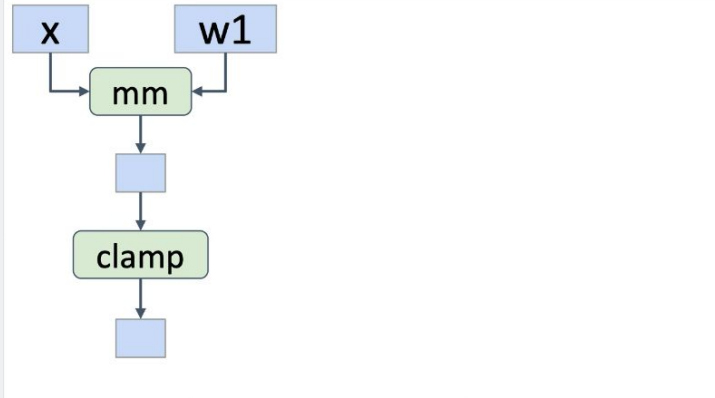
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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



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

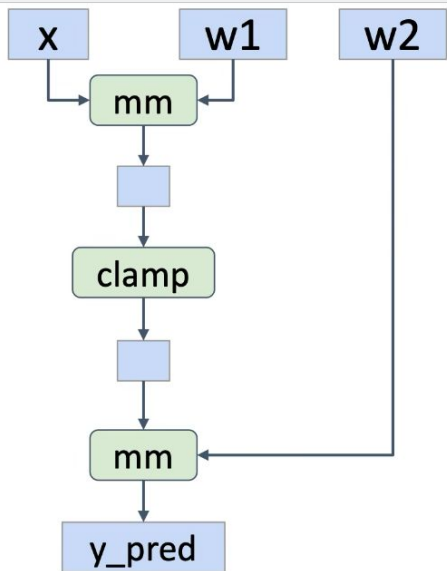
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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
```

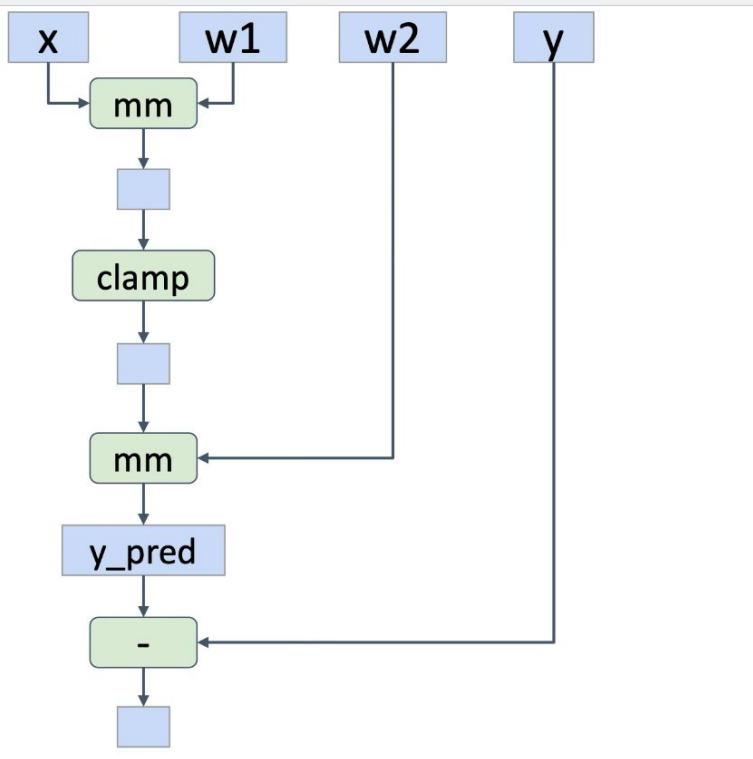
```
for t in range(500):  
    y_pred = x.mm(w1).clamp(min=0).mm(w2)  
    loss = (y_pred - y).pow(2).sum()
```

```
    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

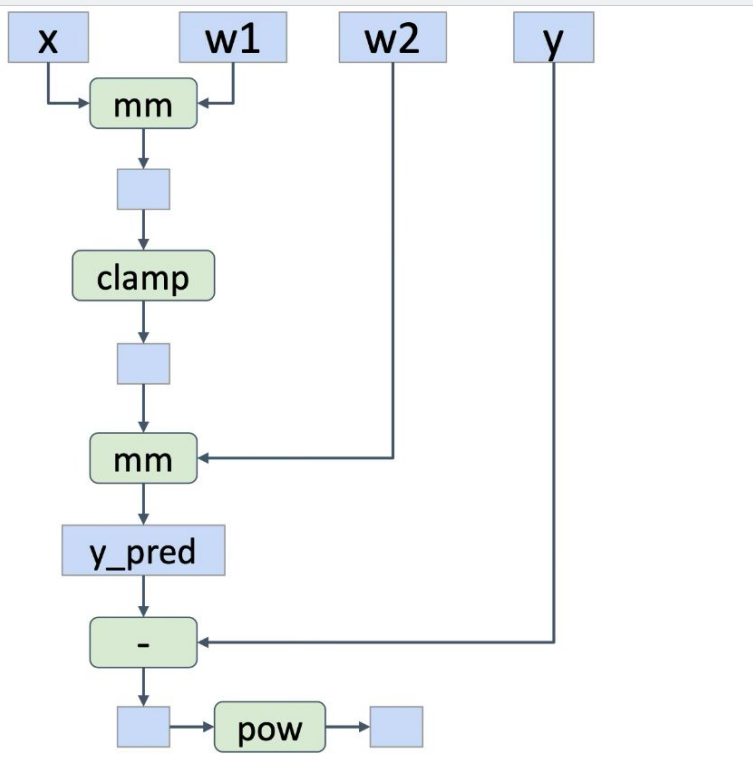
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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_()
```

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
```

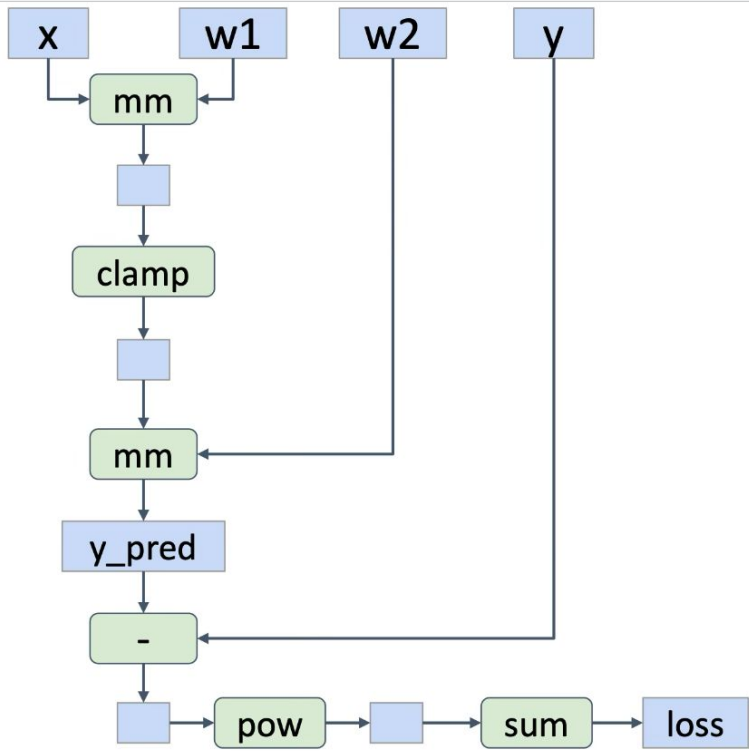
```
for t in range(500):  
    y_pred = x.mm(w1).clamp(min=0).mm(w2)  
    loss = (y_pred - y).pow(2).sum()
```

```
    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

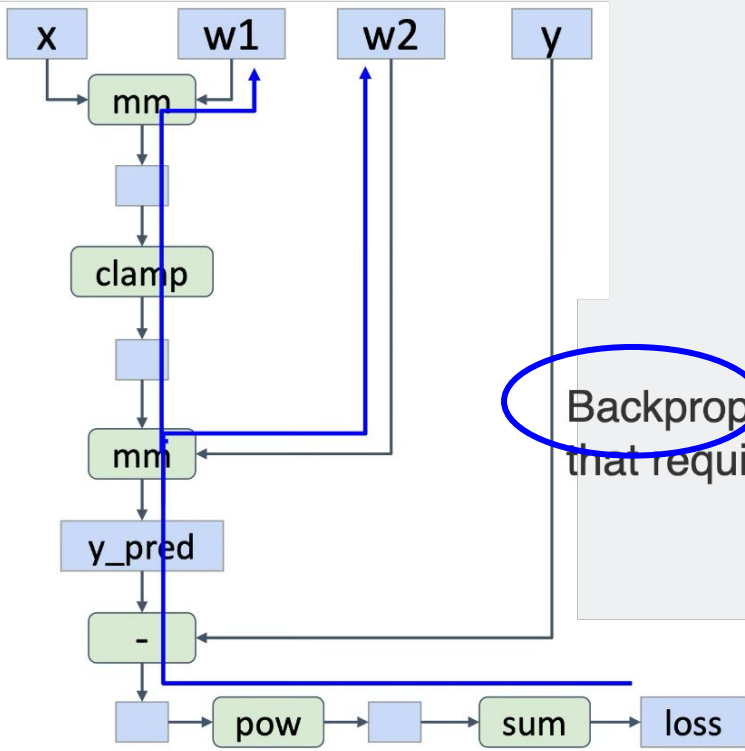
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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_()
```

PyTorch: Autograd



Backprop to all inputs
that require grad

```
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

```
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

Make gradient step on weights

```
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

Set gradients to zero—forgetting this is a common bug!

```
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

Tell PyTorch not to build a graph for these operations

```
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: New Functions

Can define new operations
using Python functions

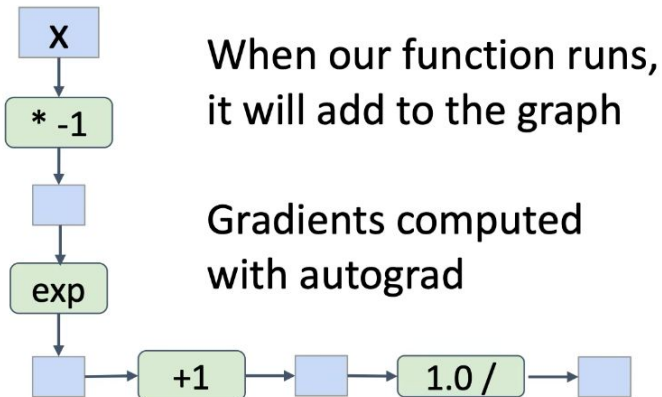
```
def sigmoid(x):  
    return 1.0 / (1.0 + (-x).exp())
```

```
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(N, D_in)  
w2 = torch.randn(N, D_out)  
  
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

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



```
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
```

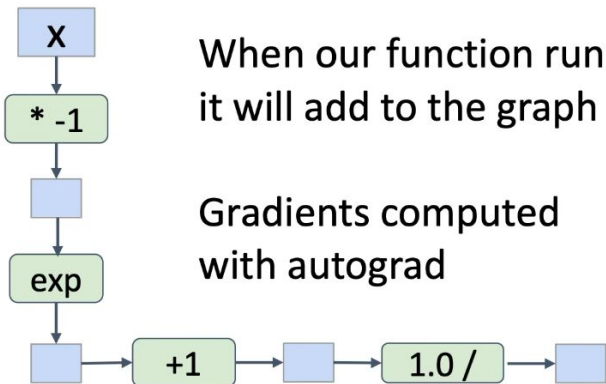
```
    w1.grad.zero_()
```

```
    w2.grad.zero_()
```

PyTorch: New Functions

Can define new operations using Python functions

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



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

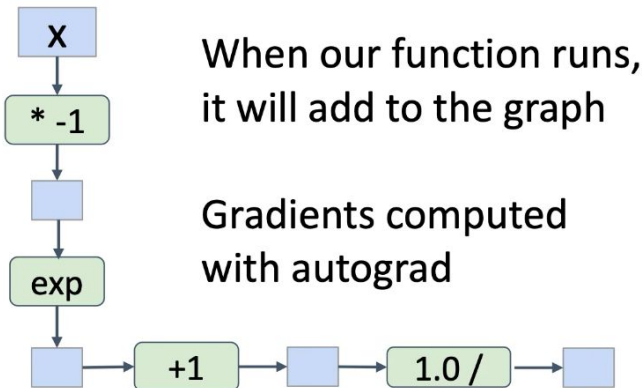
```
class Sigmoid(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x):  
        y = 1.0 / (1.0 + (-x).exp())  
        ctx.save_for_backward(y)  
        return y  
  
    @staticmethod  
    def backward(ctx, grad_y):  
        y, = ctx.saved_tensors  
        grad_x = grad_y *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

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



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

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

```
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)
```

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



PyTorch: nn

Higher-level wrapper for working with neural nets

Use this! It will make your life easier

torch.nn

[Containers](#)

[Convolution Layers](#)

[Pooling layers](#)

[Padding Layers](#)

[Non-linear Activations \(weighted sum, nonlinearity\)](#)

[Non-linear Activations \(other\)](#)

[Normalization Layers](#)

[Recurrent Layers](#)

[Transformer Layers](#)

[Linear Layers](#)

[Dropout Layers](#)

[Sparse Layers](#)

[Distance Functions](#)

[Loss Functions](#)

[Vision Layers](#)

[Shuffle Layers](#)

[DataParallel Layers \(multi-GPU, distributed\)](#)

[Utilities](#)

[Quantized Functions](#)

[Lazy Modules Initialization](#)

[↑ Back to top](#)

PyTorch: nn

Higher-level wrapper for working with neural nets

Use this! It will make your life easier

```
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

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

```
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

```
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

```
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`)

```
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

Make gradient step on
each model parameter
(with gradients disabled)

```
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: optim

Use an **optimizer** for different update rules

```
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()
```

PyTorch: optim

After computing
gradients, use optimizer to
update and zero gradients

```
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()
```

PyTorch: nn – Defining Modules

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

```
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

Define our whole model as
a single Module



```
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)



```
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

Define forward pass using child modules and tensor operations

No need to define backward - autograd will handle it



```
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

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



```
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)

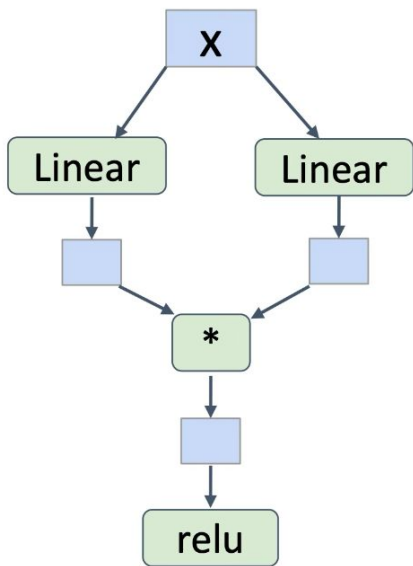
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

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

Define network component as a Module subclass



```
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)
```

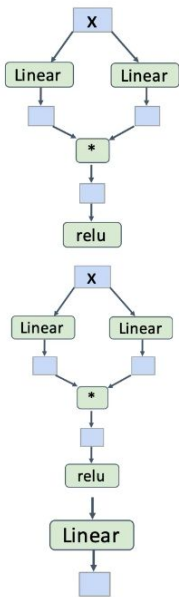
```
model = torch.nn.Sequential(  
    ParallelBlock(D_in, H),  
    ParallelBlock(H, H),  
    torch.nn.Linear(H, D_out))
```

```
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!

```
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: Data Loaders

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

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

```
import torch
from torch.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: Data Loaders

Iterate over loader to
form minibatches



```
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: PreTrained Model

Super easy to use pertained models with torch vision

<https://pytorch.org/vision/stable/>

```
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

```
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

x

w1

w2

y

```
import torch

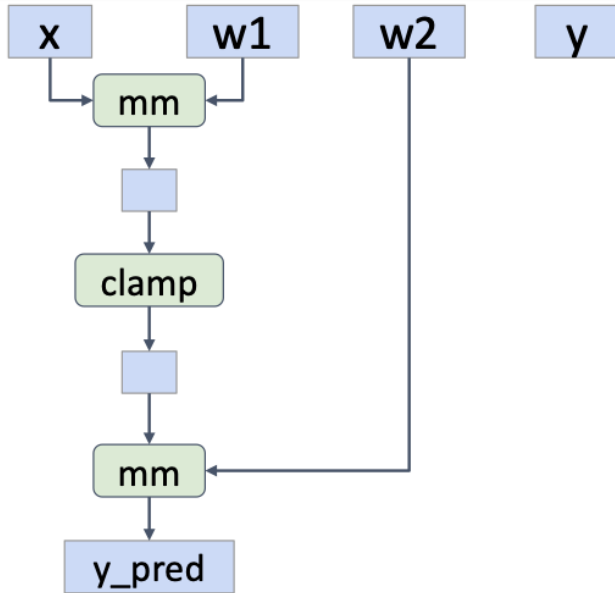
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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()
```

Create Tensor objects

PyTorch: Dynamic Computation Graphs



```
import torch

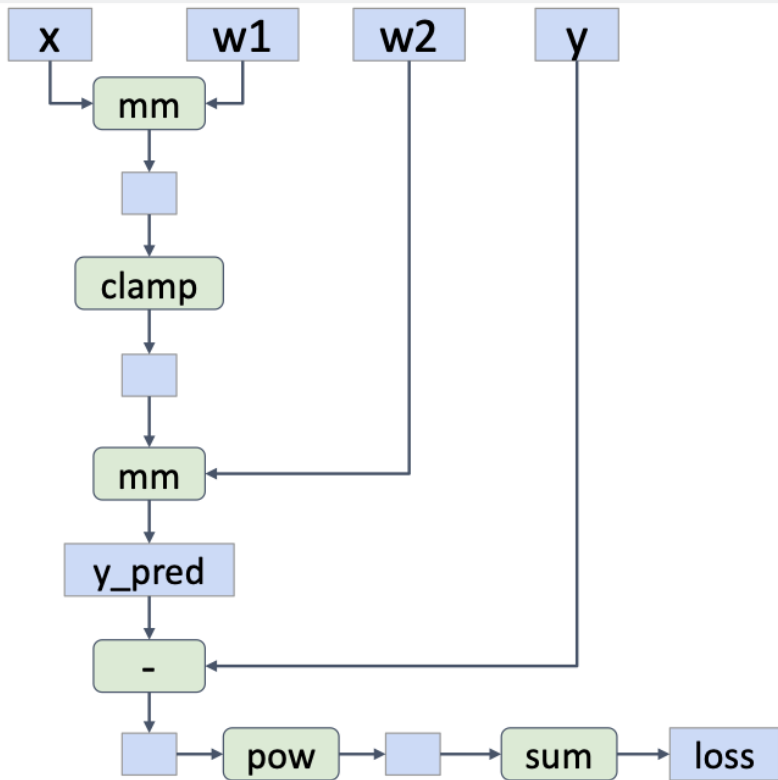
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

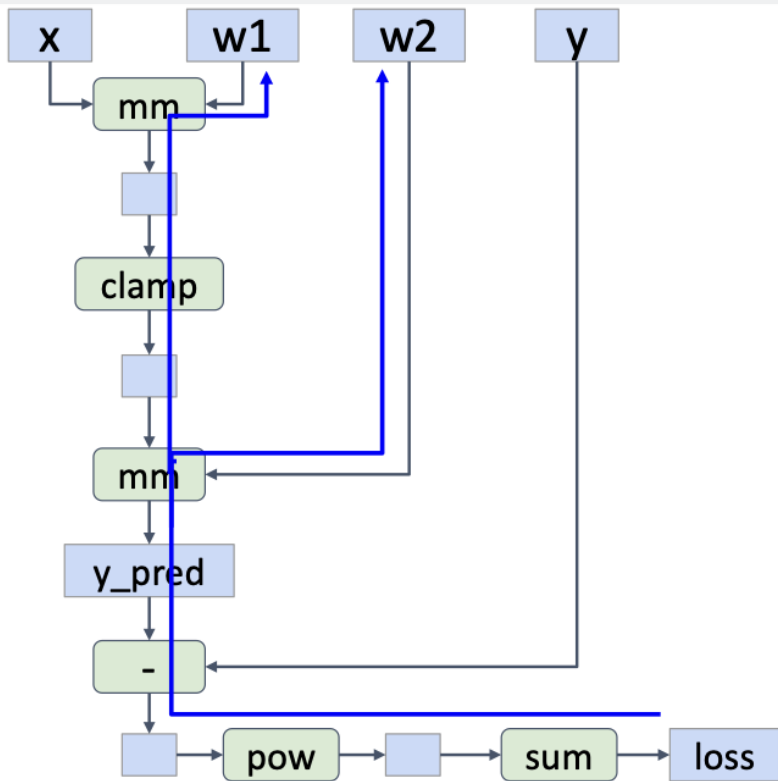
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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()
```

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

PyTorch: Dynamic Computation Graphs

x

w1

w2

y

```
import torch

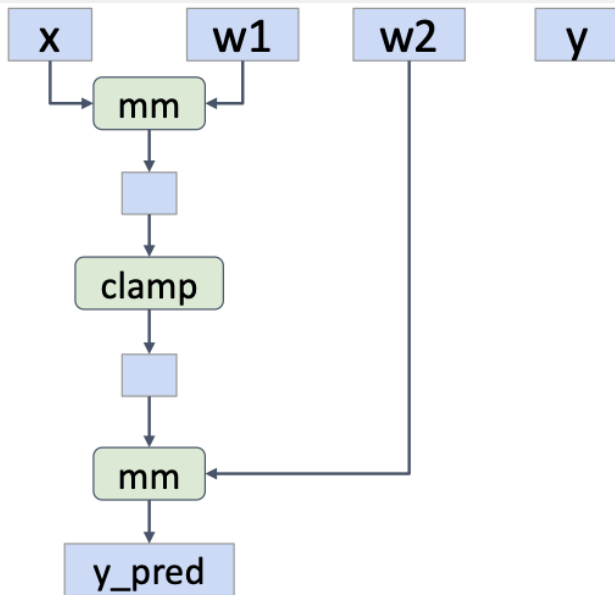
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

PyTorch: Dynamic Computation Graphs



```
import torch

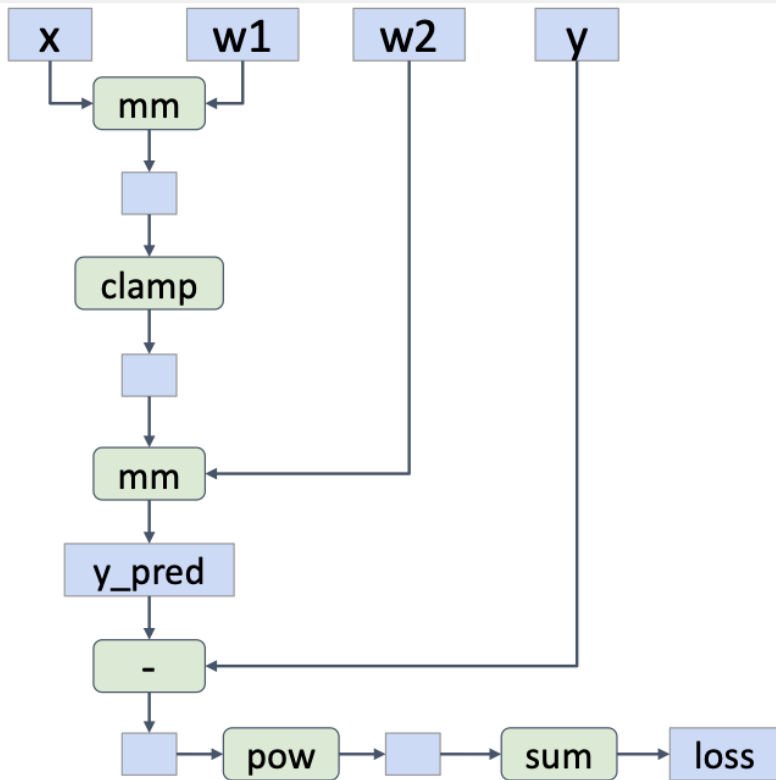
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

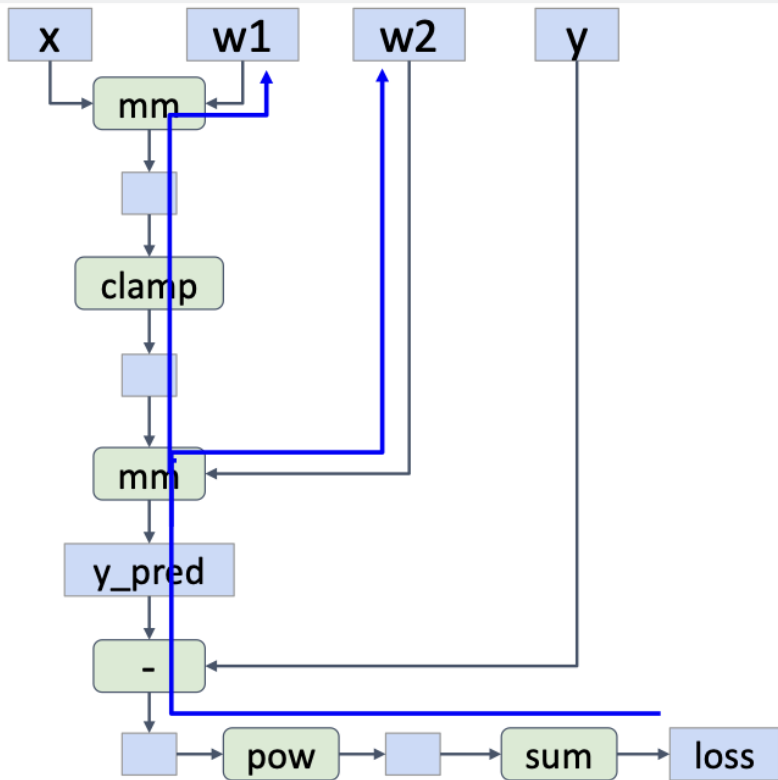
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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()
```

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

AhaSlide: Advantages of Dynamic Computation Graphs?

PyTorch: Dynamic Computation Graphs

Dynamic graphs let you use regular Python control flow during the forward pass!

```
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()
```

PyTorch: Dynamic Computation Graphs

Dynamic graphs let you use regular Python control flow during the forward pass!

Initialize two different weight matrices for second layer

```
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()
```

PyTorch: Dynamic Computation Graphs

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)

```
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()

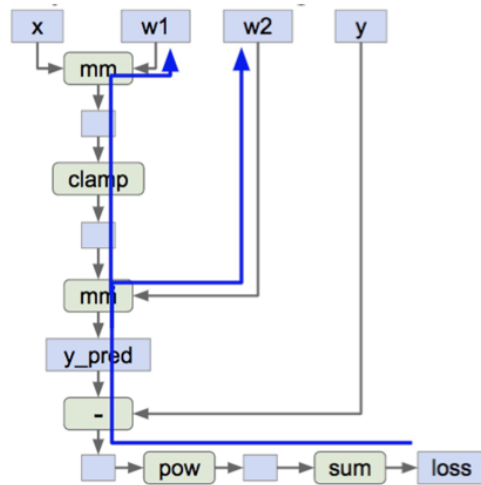
    loss.backward()
    prev_loss = loss.item()
```

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



```
graph = build_graph()
```

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

Alternative: Static Graphs with JIT

Define model as a
Python function

```
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's
“Just-In-Time (JIT)”
compiler

<https://www.geeksforgeeks.org/pytorch-jit-and-torchscript-a-comprehensive-guide/>

Alternative: 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!

```
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

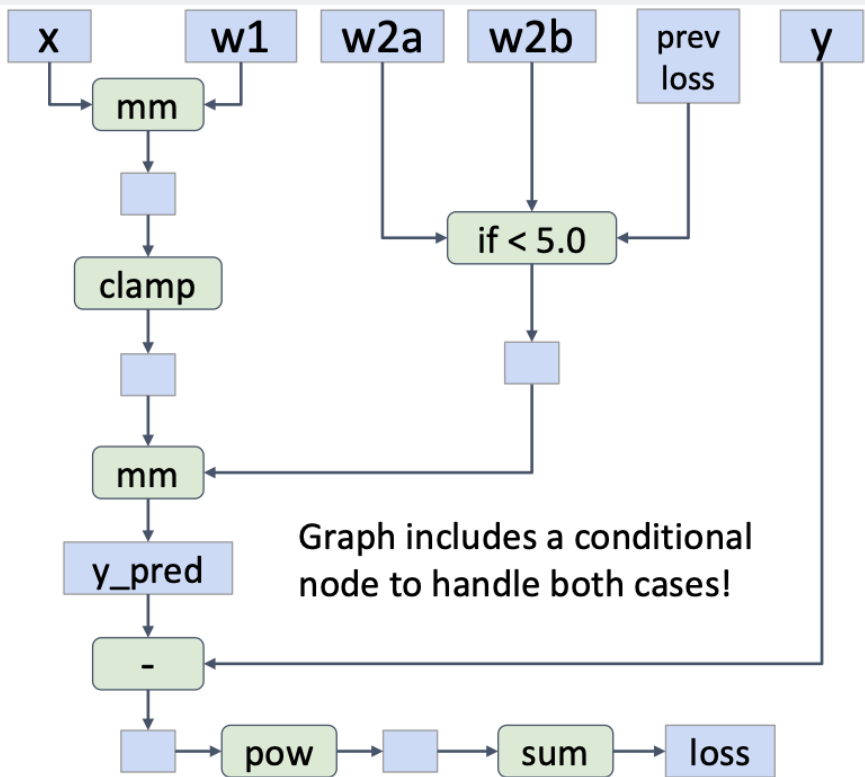
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2a = torch.randn(H, D_out, requires_grad=True)
w2b = torch.randn(H, D_out, requires_grad=True)

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()
```

Alternative: Static Graphs with JIT



```
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()
```

Alternative: Static Graphs with JIT

Use our compiled graph object at each forward pass

```
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()
```

Alternative: Static Graphs with JIT

Even easier: add **annotation** to function, Python function compiled to a graph when it is defined

Calling function uses graph

```
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)

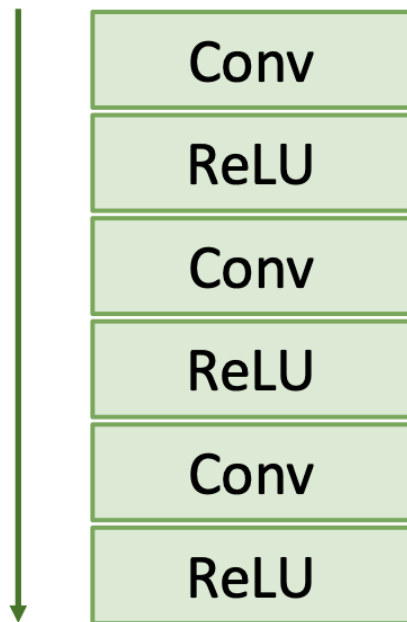
prev_loss = 5.0
learning_rate = 1e-6
for t in range(500):
    loss = model(x, y, w1, w2a, w2b, prev_loss)

    loss.backward()
    prev_loss = loss.item()
```

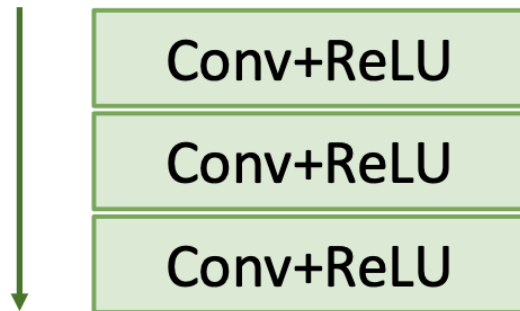
Static vs Dynamic Graphs: Optimization

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

The graph you wrote



Equivalent graph with **fused operations**



Static vs Dynamic Graphs: Optimization

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: Optimization

Static

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

Dynamic

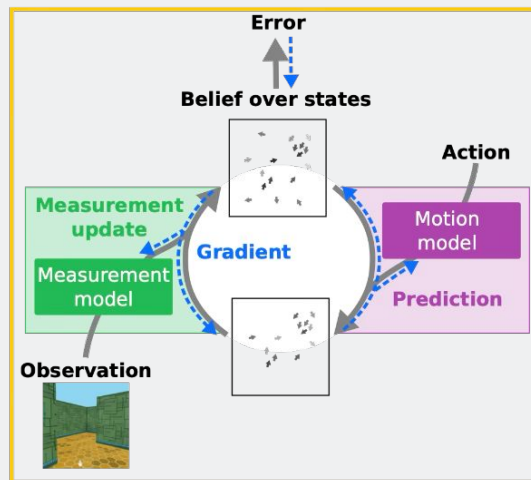
The code you write is the code that runs! Easy to reason about, debug, profile, etc

Dynamic Graph Applications

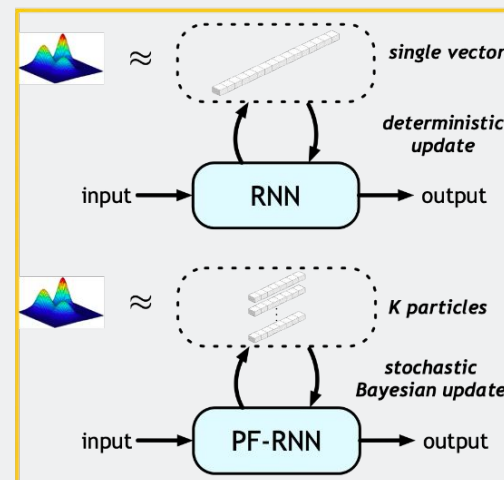
Model structure

depends on the input:

- Recurrent Networks
- Recursive Networks



[1] Ma et al., RSS 2018



[2] Ma et al., AAI 2020

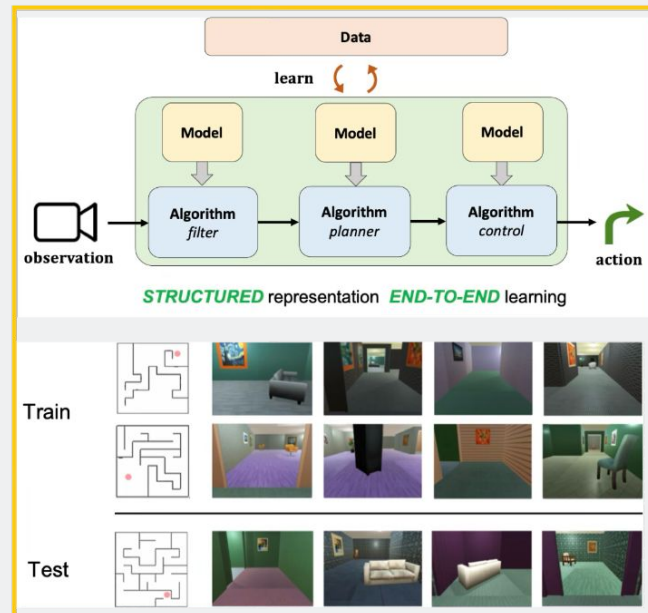
[1] Rico Jonschkowski, Divyam Rastogi, Oliver Brock. "Differentiable Particle Filters: End-to-End Learning with Algorithmic Priors" RSS, 2018

[2] Xiao Ma, Peter Karkus, David Hsu, Wee Sun Lee. "Particle Filter Recurrent Neural Networks" AAI, 2020.

Dynamic Graph Applications

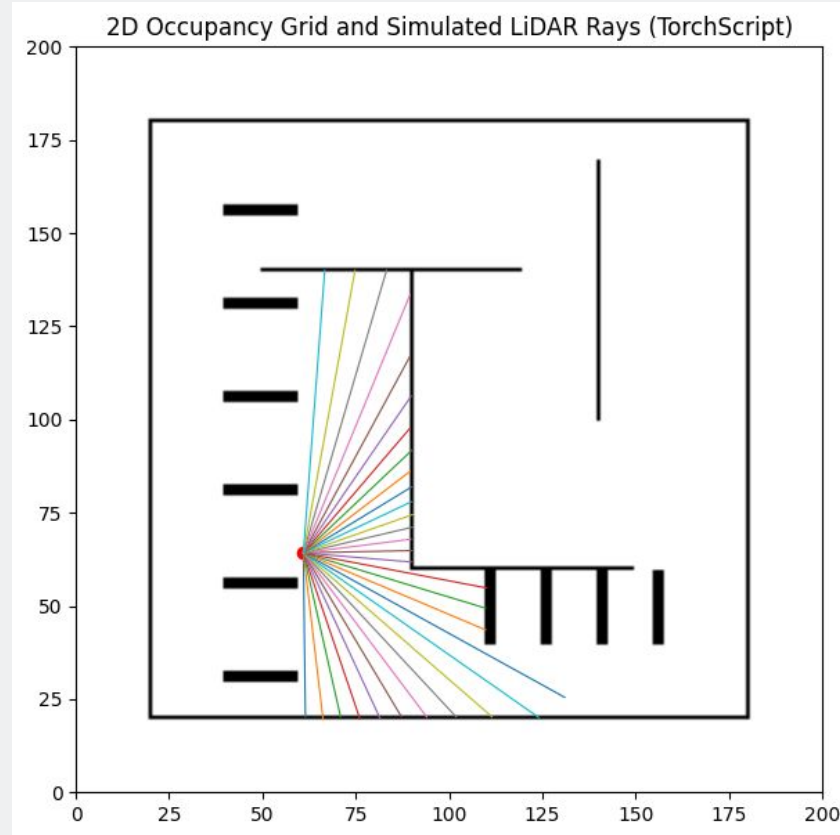
Model structure depends on the input:

- Recurrent Networks
- Recursive Networks
- Modular Networks



[1] Karkus et al., RSS 2019

Demo: **Dynamic vs. Static** for 2D LiDAR Raytracing



Logistics/Reminders

Midterm

- **Midterm:** March 11, 2026, 12pm-1:30pm EST (this Wednesday in class, @ 1303 EECS)
 - Pen/Pencil and paper exam
 - 1 A4/Letter-size note, front and back
 - No GenAI/phone/computer/internet

- **Review Session Tomorrow (Tuesday)**
[Set of review questions here](#)

Sign up for Final Project Teams

<https://docs.google.com/spreadsheets/d/1Jjv6vBQwAtCpUSQDAx0Gj5pDCWWqcvSQEeBFIF5prqo/edit?usp=sharing>

Please **sign up** by Friday March 12
After that, instructors will place you in teams

Should start thinking about final project now!!!

Suggested Papers Updated

https://deeprob.org/w26/papers/important_works/

[Papers](#) / Important Works

Deep Learning Research Papers for Robot Perception: Important Works

This collection highlights foundational papers and promising recent research in deep learning and robot perception.

TABLE OF CONTENTS

- 1 [RGB-D Architectures](#)
- 2 [Point Cloud Processing](#)
- 3 [Object Pose, Geometry, SDF, Implicit surfaces](#)
- 4 [Dense Descriptors, Category-level Representations](#)
- 5 [Recurrent Networks and Object Tracking](#)
- 6 [Visual Odometry and Localization](#)
- 7 [Semantic Scene Graphs and Explicit Representations](#)
- 8 [Neural Radiance Fields and Implicit Representations](#)
- 9 [NeRF SLAM](#)
- 10 [Datasets](#)
- 11 [Self-Supervised & Representation Learning](#)
- 12 [Grasp Pose Detection](#)
- 13 [Tactile Perception for Grasping and Manipulation](#)
- 14 [Pre-training for Robot Manipulation](#)
- 15 [Generative Modeling & Dynamic Scenes](#)

P4

P4: PoseCNN and Transformer

You can get started on the posecnn part now

<https://drive.google.com/drive/folders/1VKleaTROCbOPtJ9m4zW3iTortC6ZSqAl?usp=sharing>

Transformer part will be released later this week)

Refer to [PoseCNN lecture](#) for hints and help.

P4 Due: March 29, 2026

Final Project

Final Project:

- Sign up by **March 12**. Teams released around March 15.
- “Lightning Talk” - **March 31** Discussion 3:30pm-5:30pm
- Final presentation - **April 27** @FRB Atrium
1:30pm-3:30pm
- Final Project video and report Due **April 27 11:59pmEST**

TensorFlow

TensorFlow: Versions

TensorFlow 1.0

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

TensorFlow 2.0

- Current release: **2.21.0**
 - Released **03/06/2026**
- Default: **dynamic graphs**
- Optional: static graphs

TensorFlow 1.0: Static Graphs

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

(Assume imports at the top of each snippet)

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

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

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

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

TensorFlow 1.0: Static Graphs

First **define** computational graph

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

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

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

Then **run** the graph many times

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

TensorFlow 2.0: Dynamic Graphs

Create TensorFlow
Tensors for data and
weights

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

```
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.matmul(x, w1)  
        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: Dynamic Graphs

Scope forward pass
under a GradientTape to
tell TensorFlow to start
building a graph



```
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)
```

In PyTorch, all ops build graph by default; **opt out** via `torch.no_grad`
In Tensorflow, ops do not build graph by default; **opt in** via `GradientTape`

TensorFlow 2.0: Dynamic Graphs

https://www.tensorflow.org/api_docs/python/tf/GradientTape

Ask the tape to
compute gradients



```
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: Dynamic Graphs

Gradient descent
step, update weights



```
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`)

```
@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

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)

```
@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



```
@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)
```

Keras

Keras: High-Level API

Keras 3.0 released

<https://keras.io/>

“Run your Keras workflows on top of either JAX, TensorFlow, PyTorch, or OpenVINO (for inference-only)”

```
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

```
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

```
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

```
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

```
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))
```

Optimizer object
updates parameters



Keras: High-Level API

Define a function
that returns the loss



```
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

```
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

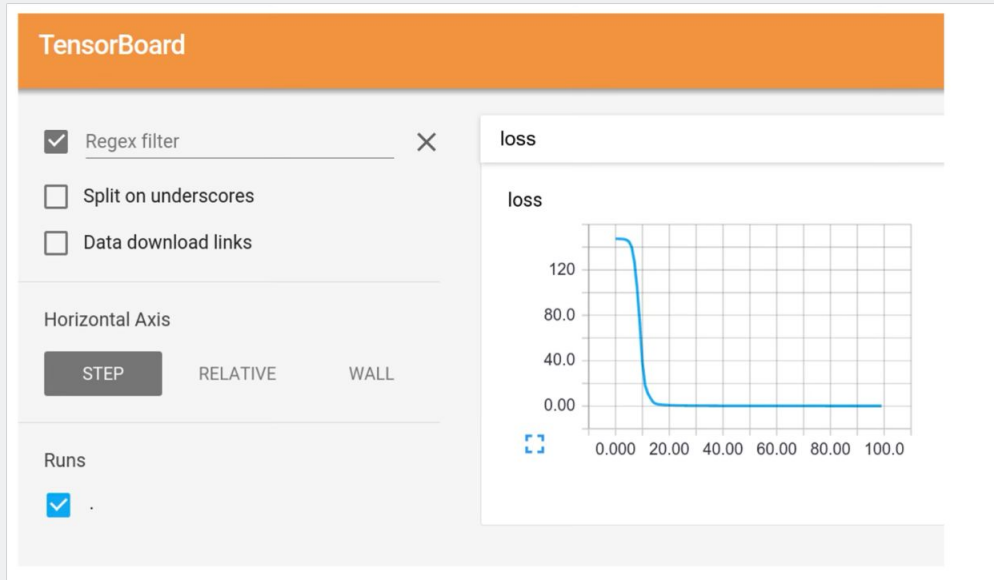
for t in range(1000):
    opt.minimize(step, params)
```

Optimizer computes
gradients and
updates parameters



TensorBoard

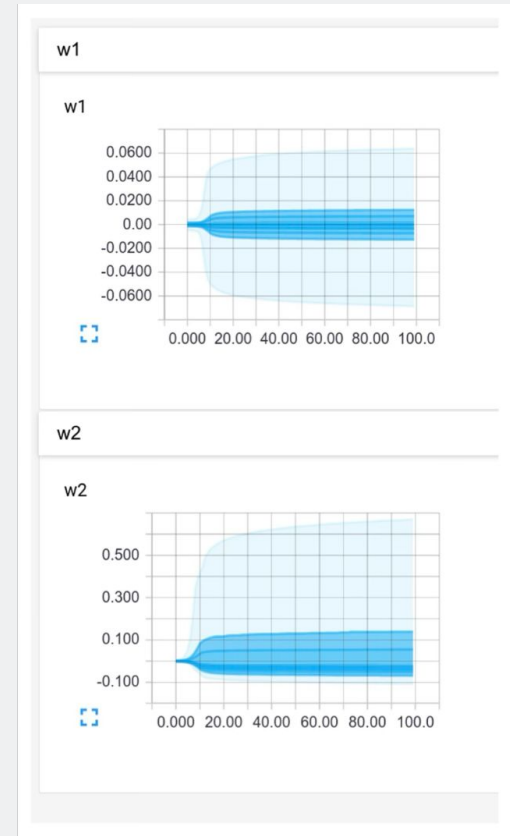
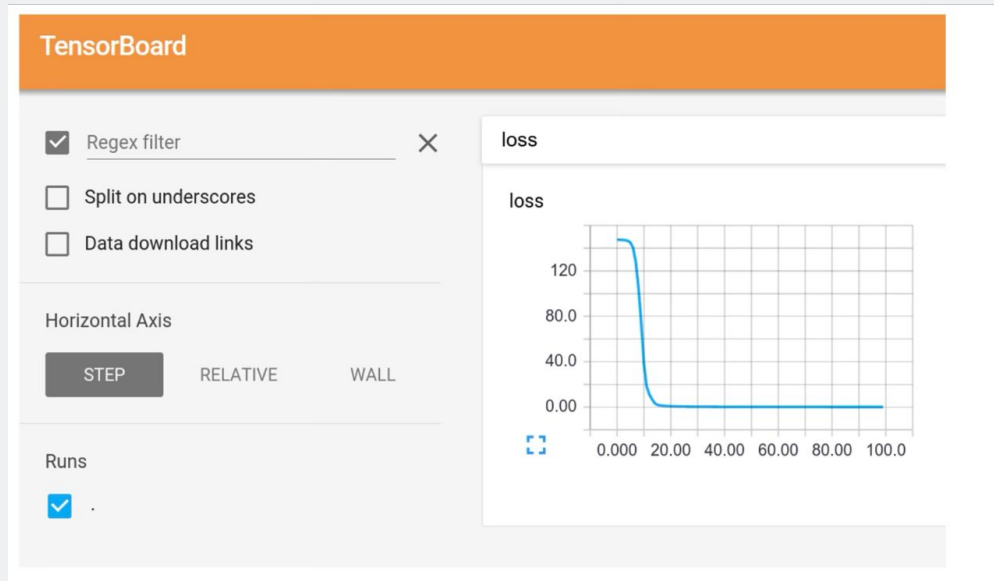
Add logging to code to record loss, stats, etc
Run server and get pretty graphs!



TensorBoard

Also, <https://wandb.ai/site/>

Also works with [PyTorch!](#)



PyTorch vs TensorFlow

PyTorch

- ***What the course is using!***
- Clean, imperative API
- Easy dynamic graphs for debugging
- JIT allows static graphs for production
- **Hard / inefficient to use on TPUs**
- **Not easy to deploy on mobile**

TensorFlow 1.0

- Static graphs by default
- **Can be confusing to debug**
- **API a bit messy**

TensorFlow 2.0

- Dynamic by default
- Standardized on Keras API
- API still confusing

Be aware of versions

My own jupyter notebook for projects:

```
torch.version.cuda      12.8
```

```
torchvision.__version__ 0.25.1+cu128
```

Docker:

<https://docs.docker.com/get-started/>