

# Create Your Model: PyTorch

CSE 849 Deep Learning  
Spring 2025

Zijun Cui

# A zoo of frameworks!

Caffe  
(UC Berkeley)



Caffe2  
(Facebook)

Torch  
(NYU / Facebook)



PyTorch  
(Facebook)

We'll focus on this

Theano  
(U Montreal)



TensorFlow  
(Google)

PaddlePaddle  
(Baidu)

MXNet  
(Amazon)

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

Chainer

CNTK  
(Microsoft)

JAX  
(Google)

# 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 1.10**
- (Released October 2021)
- 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: 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
```

# PyTorch: Tensors

Create random tensors  
for data and weights

X: [64, 1000]

Y: [64, 10]

W1: [1000, 100]

W2: [100, 10]

```
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

X: [64, 1000]  
Y: [64, 10]  
W1: [1000, 100]  
W2: [100, 10]

Forward pass: compute  
predictions and loss

$h = XW1 : [64, 100]$   
 $h\_relu = \text{ReLU}(h) : [64, 100]$   
 $Y\_pred = h\_reluW2 : [64, 10]$

```
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



```
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

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
```



# 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
```

# 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

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

```
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

Computes 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)

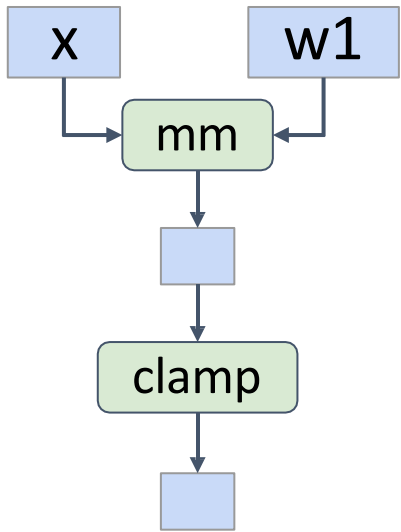
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    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

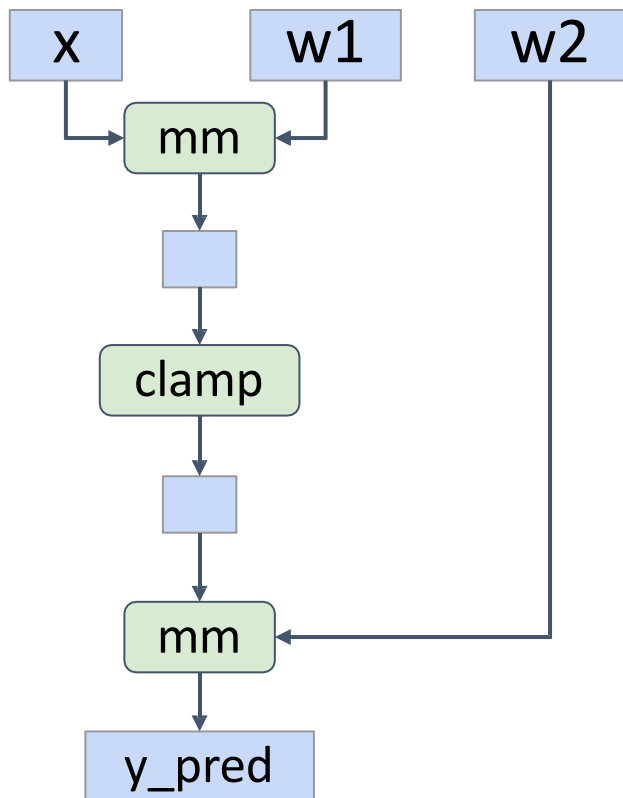
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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



```
import torch

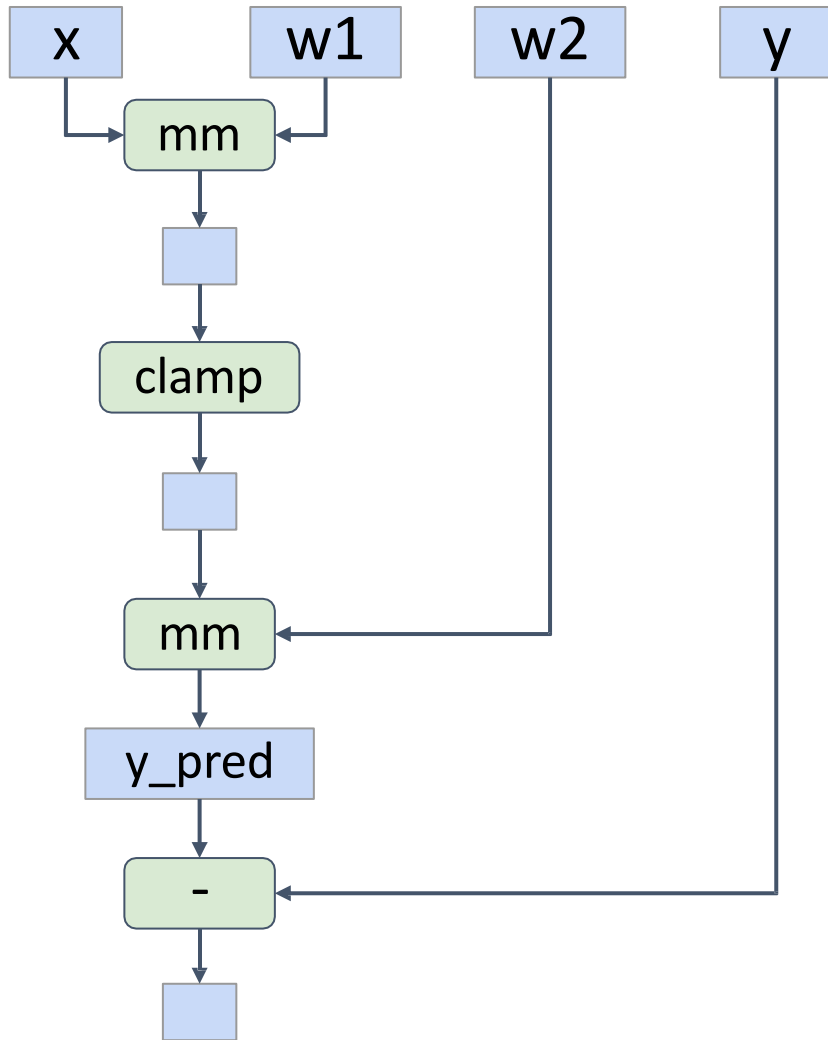
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

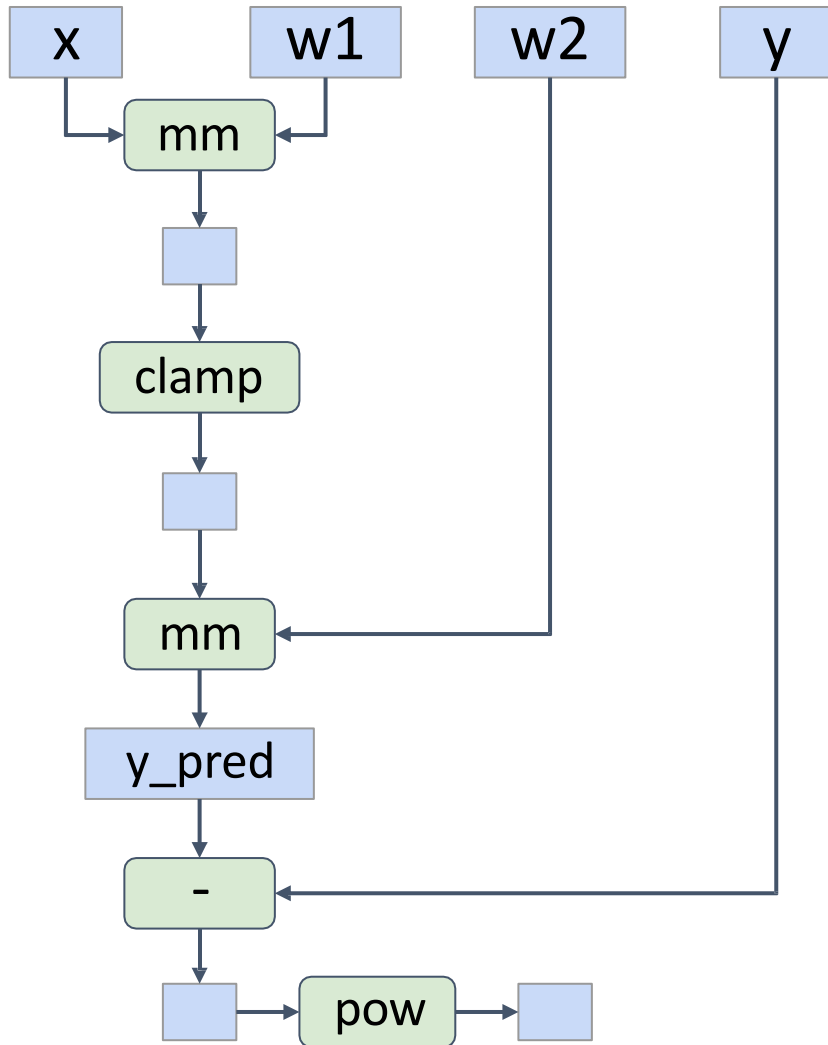
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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)

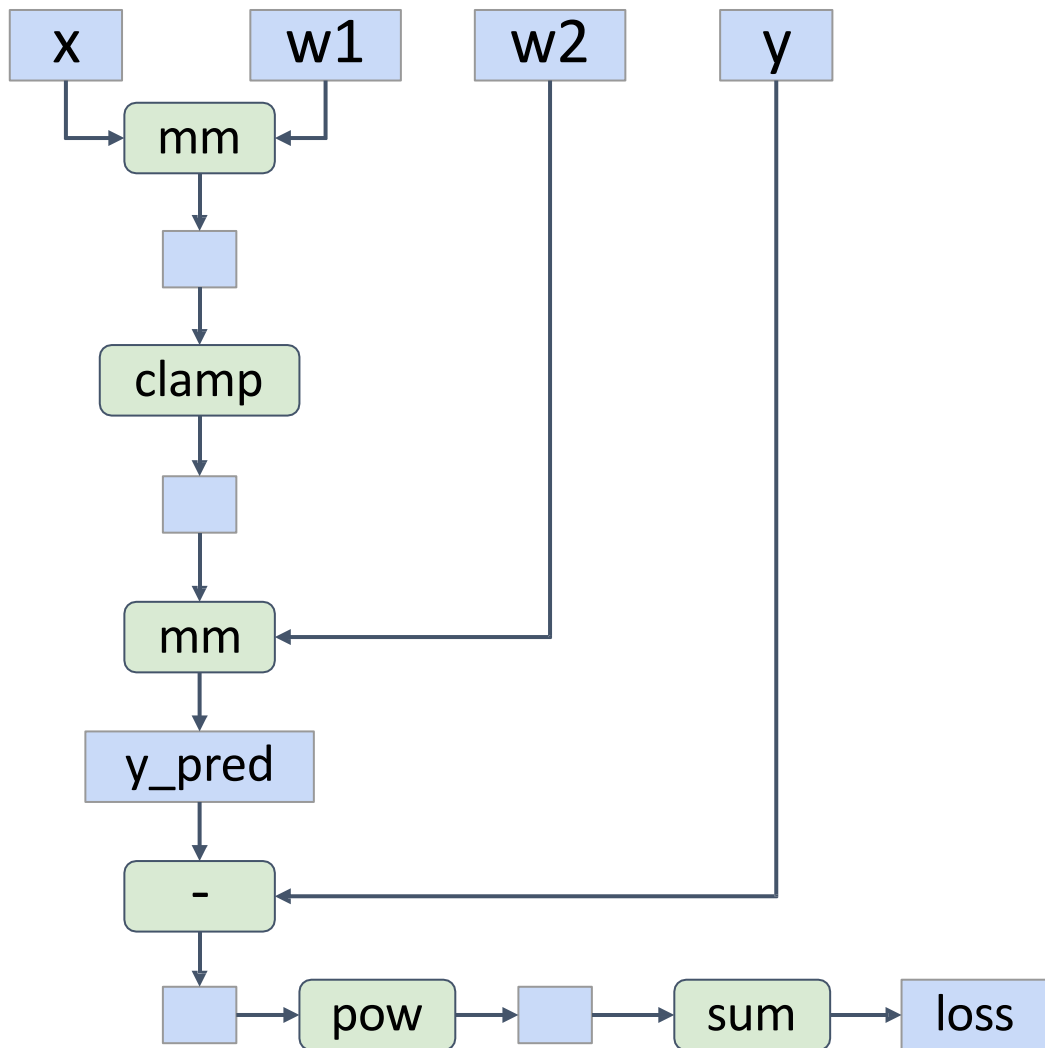
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    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

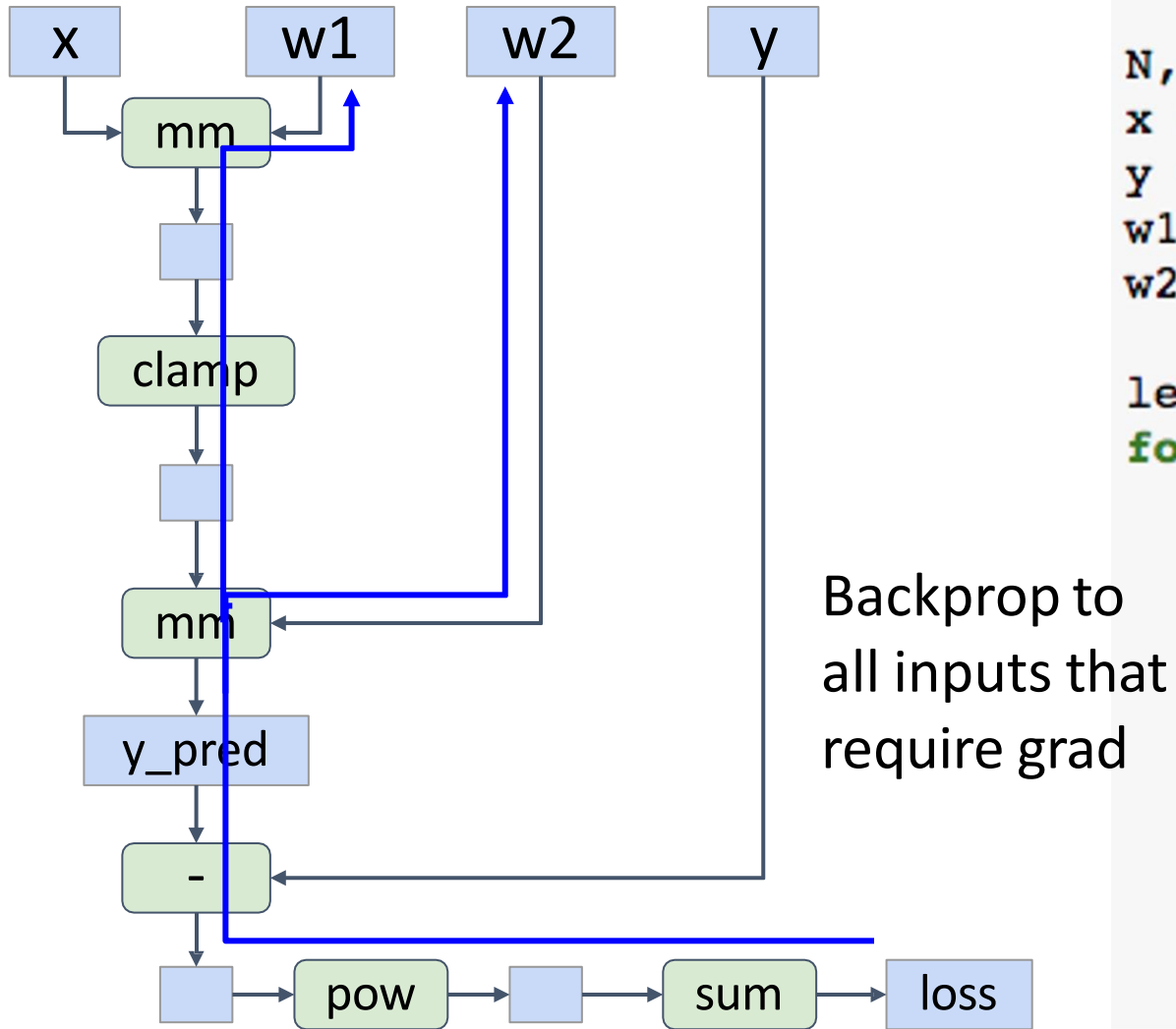
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

x

w1

w2

y

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

x

w1

w2

y

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

x

w1

w2

y

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

x

w1

w2

y

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())
```

When our function runs, it will add to the graph  
Gradients computed with 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)
```

```
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

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

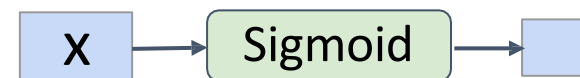
$$\frac{\partial}{\partial x} [\sigma(x)] = (1 - \sigma(x))\sigma(x)$$

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

```
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`)

Q: where did we specify `requires_grad = True`?

by default, all parameters of layers in `torch.nn` modules have `requires_grad=True` automatically set

```
for param in model.parameters():  
    print(param.requires_grad)  
# print True for all trainable parameters
```

Today's Star

```
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

```
import torch

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

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

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                               lr=learning_rate)

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

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

After computing  
gradients, use optimizer to  
update and zero gradients





# 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

In this case, `self.linear 1` and `self.linear 2` are submodules of the `TwoLayerNet` 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

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)

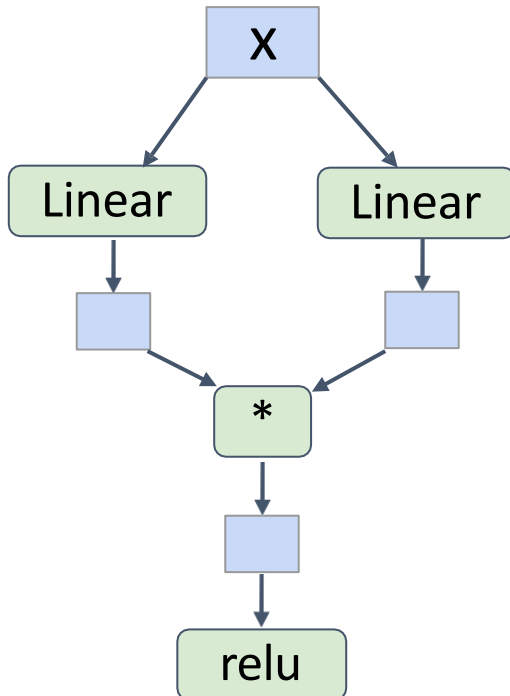
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

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)
```

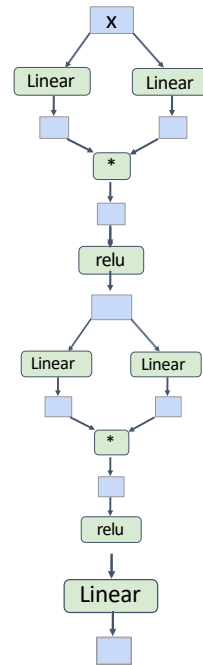
```
N, D_in, H, D_out = 64, 1000, 100, 10  
x = torch.randn(N, D_in)  
y = torch.randn(N, D_out)
```

```
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!

Today's Star

```
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: DataLoaders

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

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

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

- Tensors; Autograd; with new function
- nn; optim; define modules
- dataloaders



```
import torch

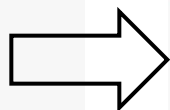
device = torch.device('cuda:0')

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
```



```
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 Models

Super easy to use pretrained models with torchvision

<https://github.com/pytorch/vision>

```
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)
```

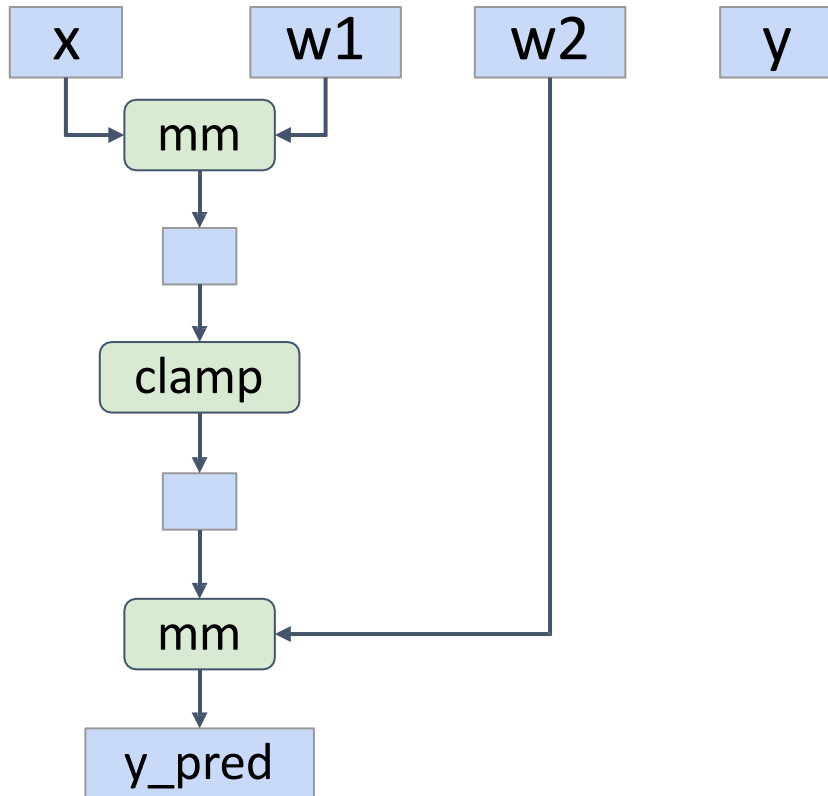
```
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    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)

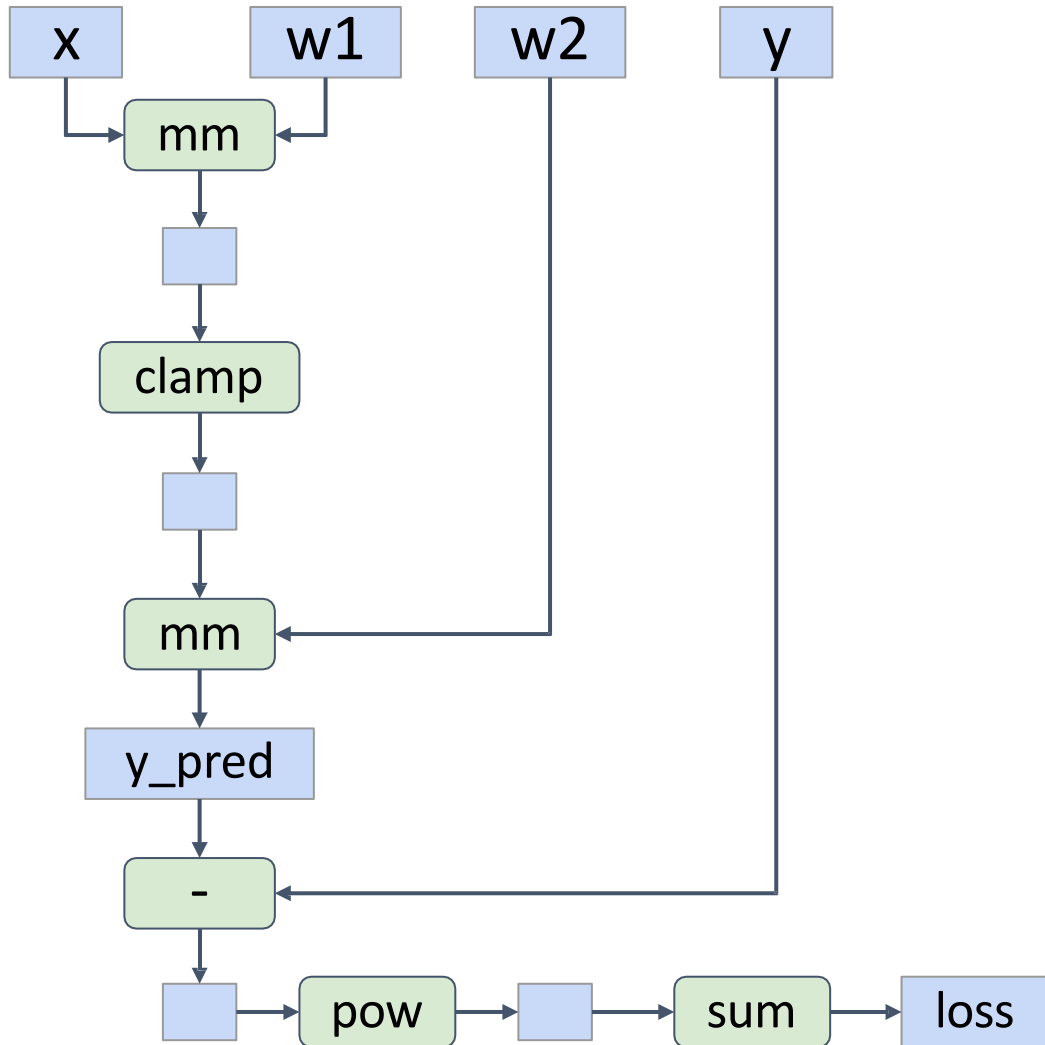
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure  
AND perform computation



# PyTorch: Dynamic Computation Graphs



```
import torch

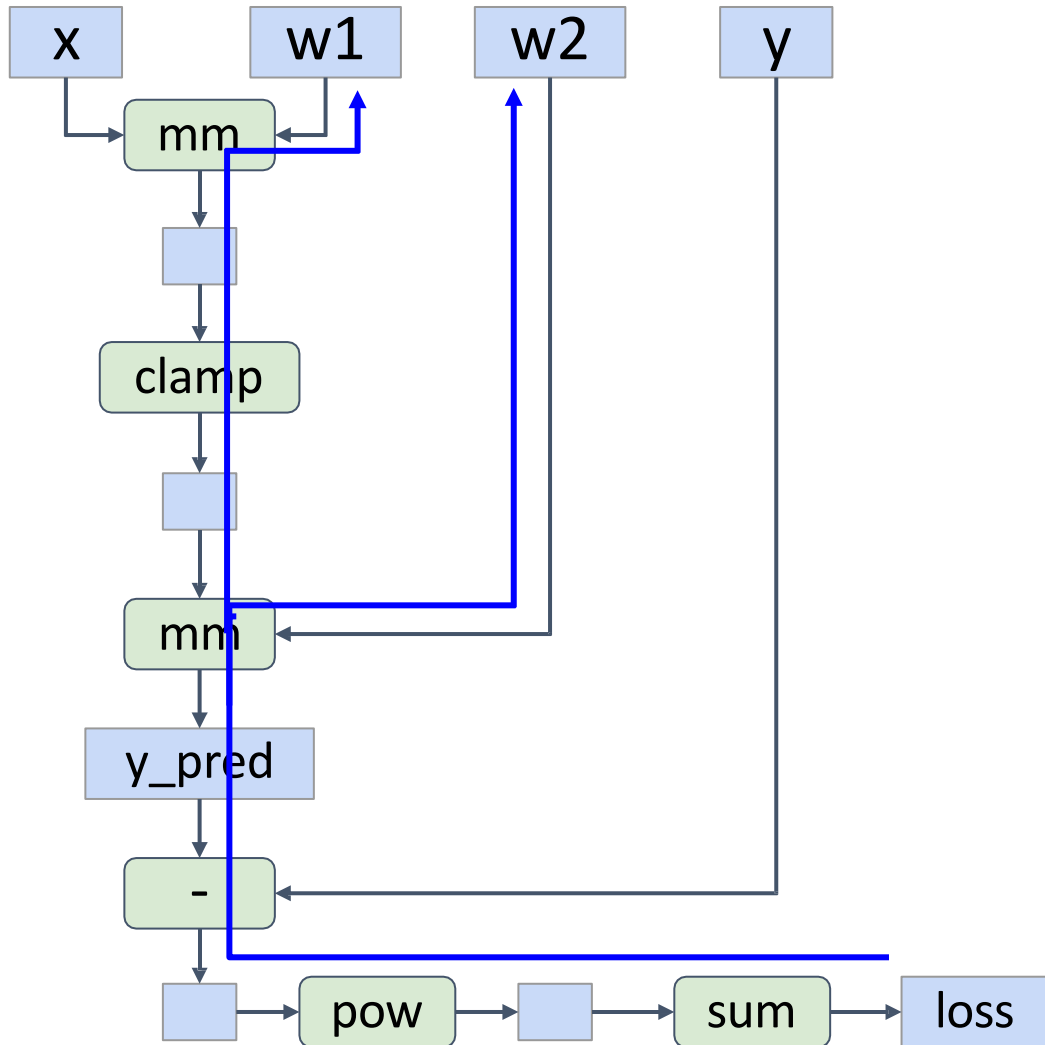
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

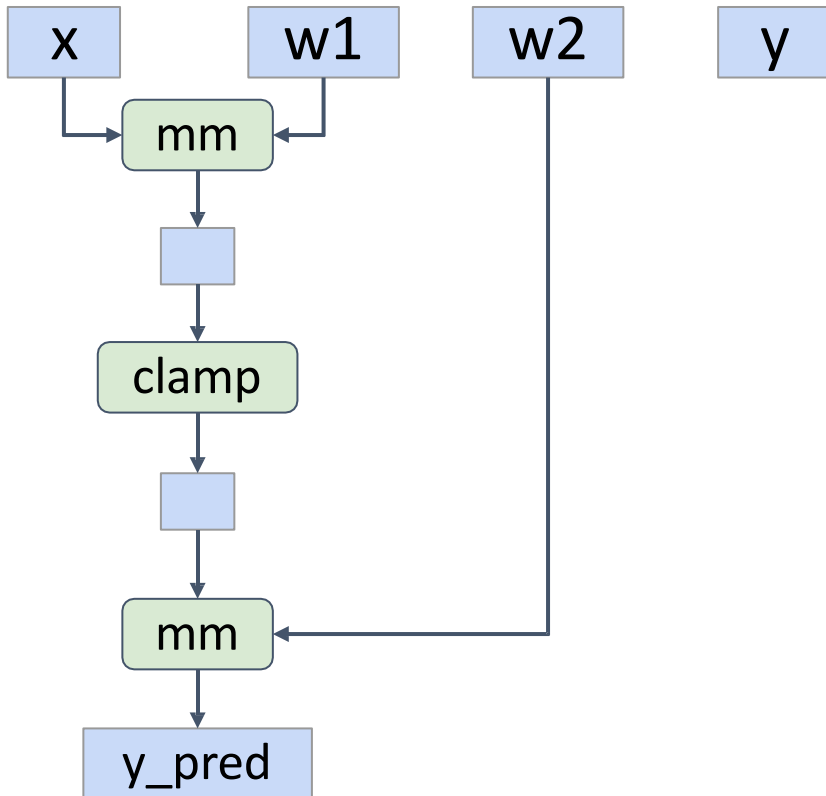
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

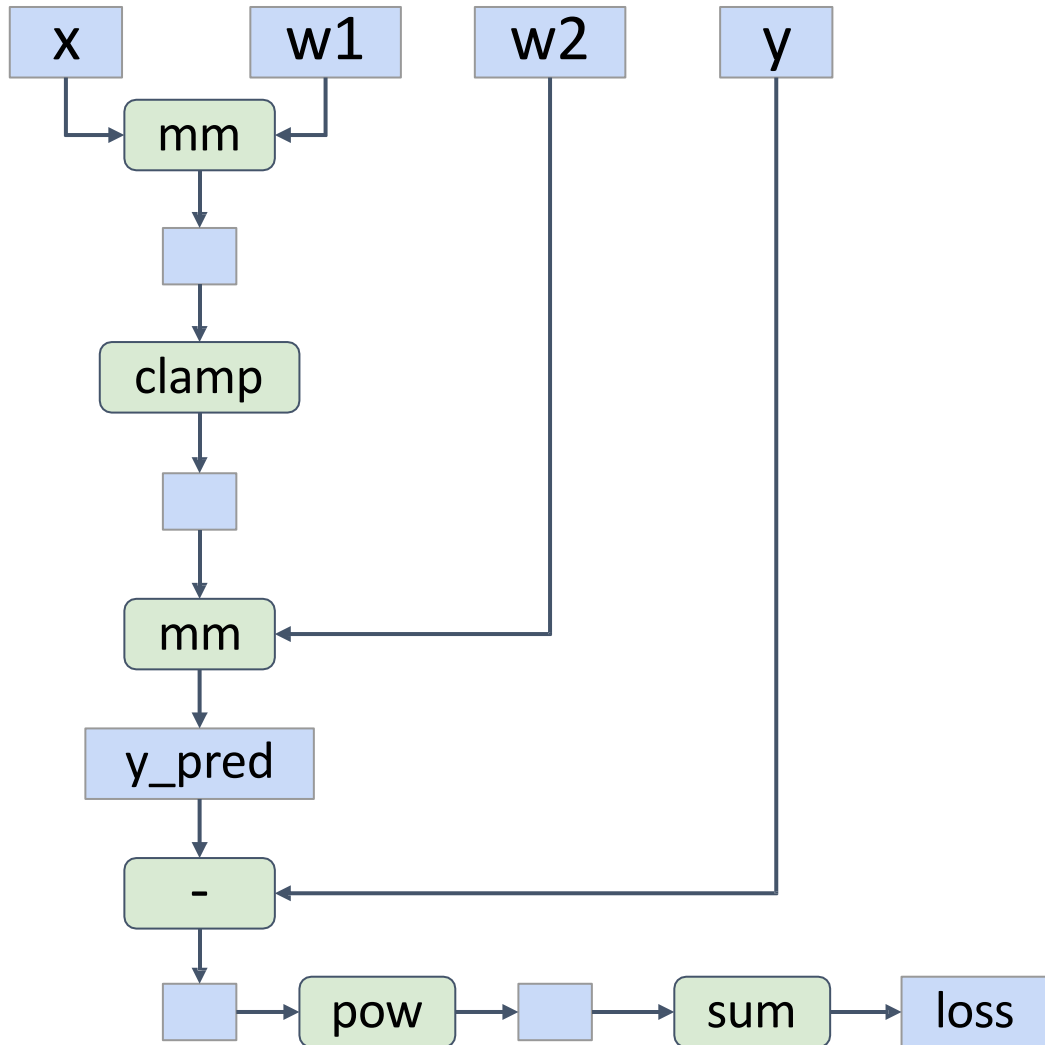
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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)

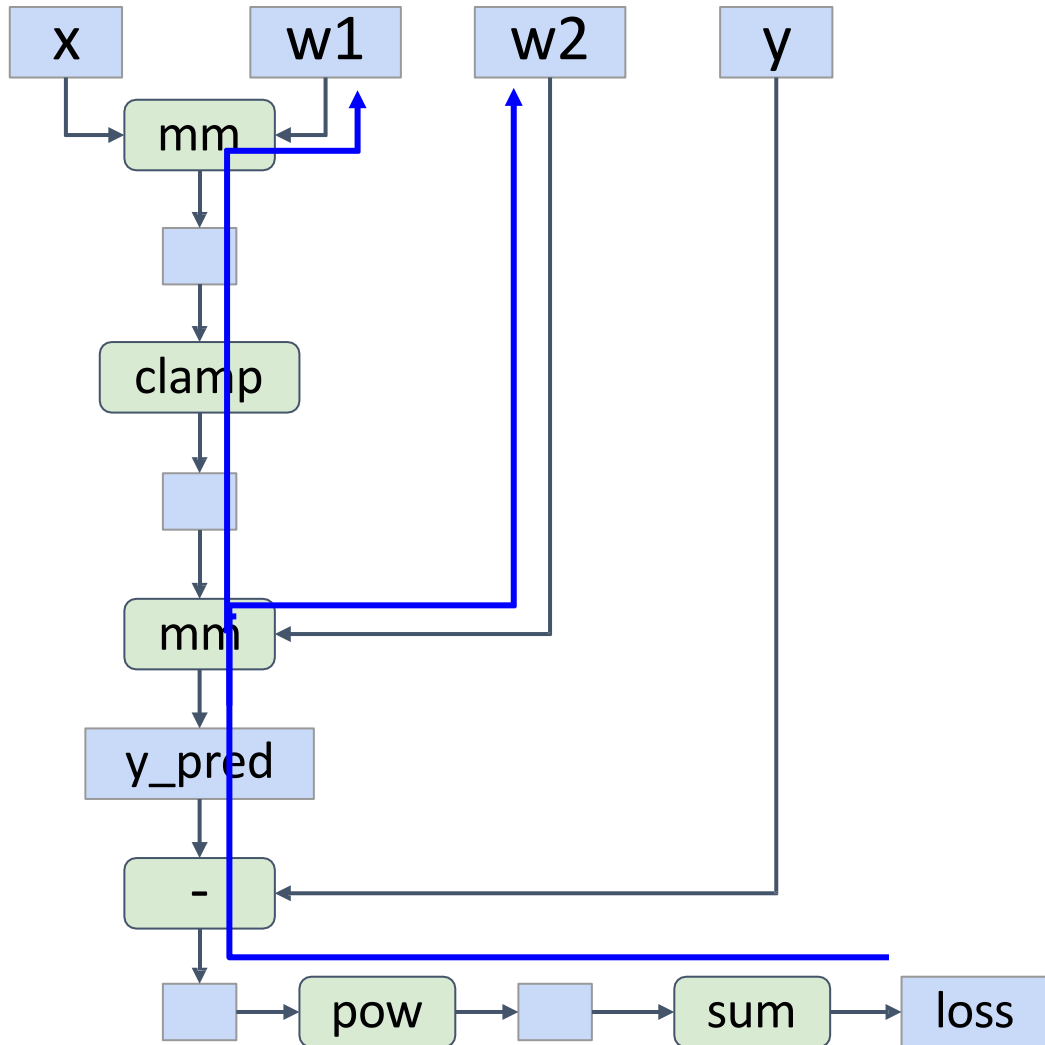
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    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)

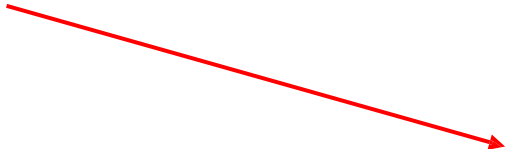
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```


Perform backprop,  
throw away graph

# PyTorch: Dynamic Computation Graphs

Initialize two different weight matrices for second layer



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



```
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()
```

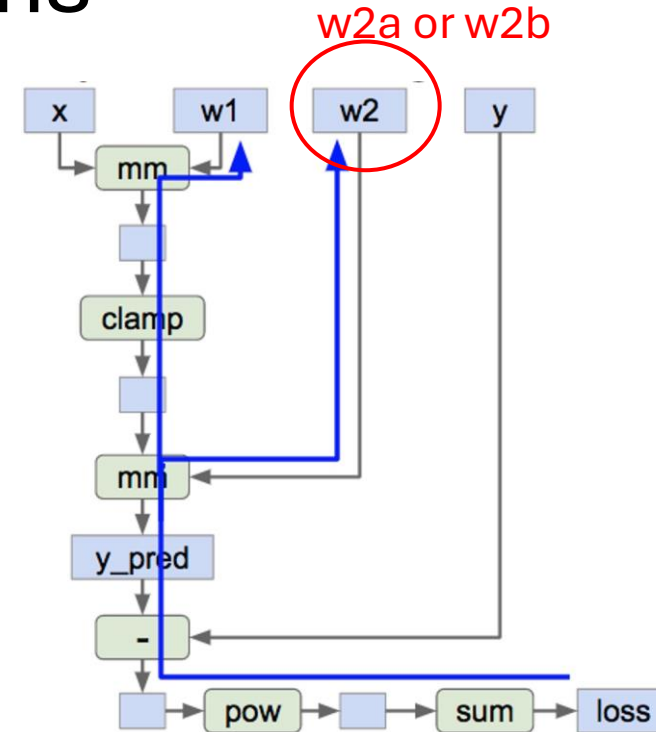
*Note: this model is only a simple dynamic example, doesn't make much sense in real applications.*

# With Static Computation Graphs

## Static graphs

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

Step 2: Reuse the same graph on every iteration



Need to decide w2a or w2b every time before building the graph  
Can't be adapted during runtime

```
graph = build_graph()
```

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

## Solution with Dynamic Computation Graph?

# PyTorch: 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: 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)

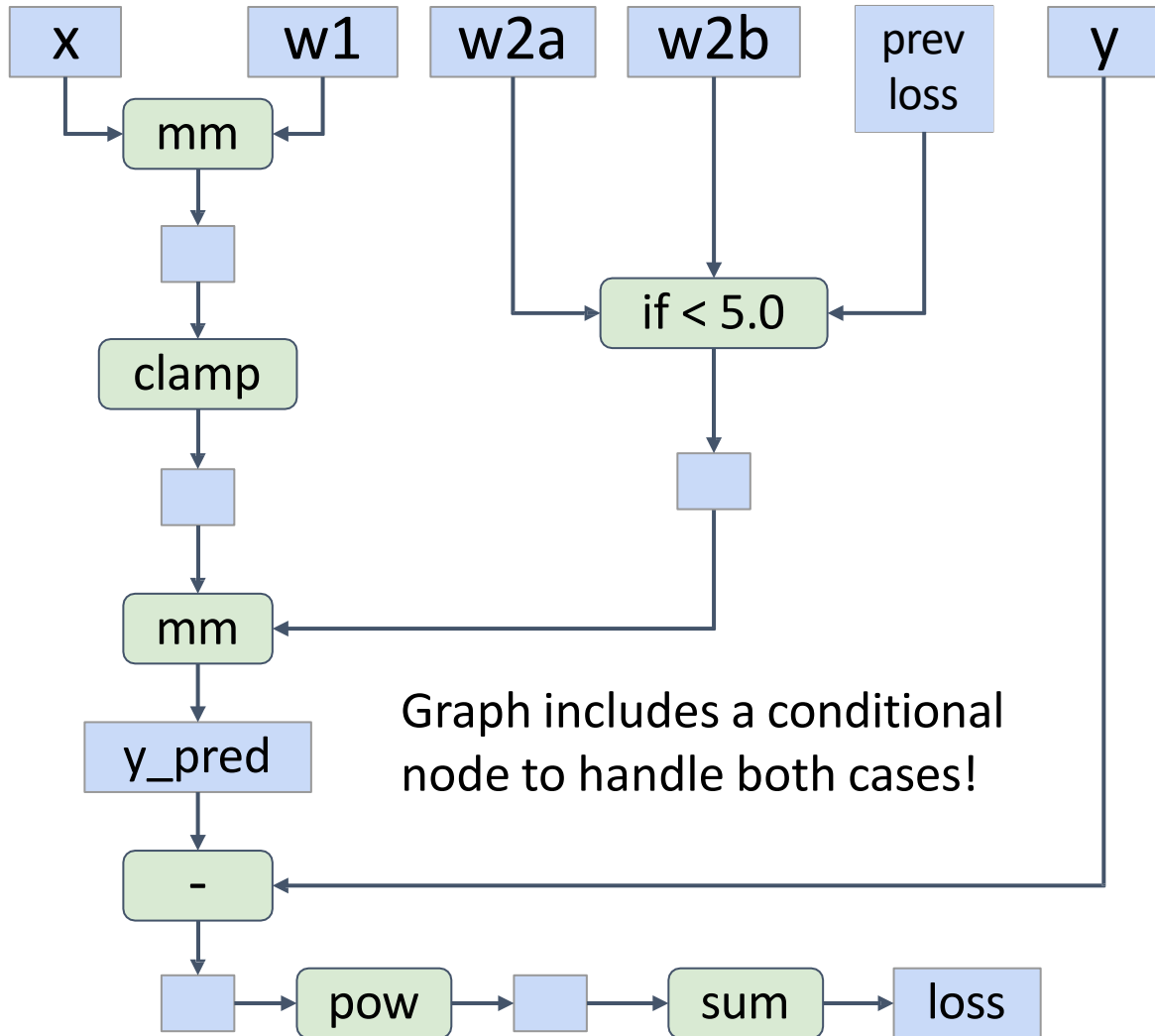
graph = torch.jit.script(model)

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()
```



# PyTorch: 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()
```

# PyTorch: 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()
```

# PyTorch: 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: Flexibility

**Static** computation graphs (e.g., Tensorflow 1.x) can handle if or for theoretically, but require special mechanisms instead of regular Python control flow.

**Dynamic** graphs (PyTorch) let you use **regular Python control flow** (if condition, for loops) during the forward pass.

All you need is to call JIT

```
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)

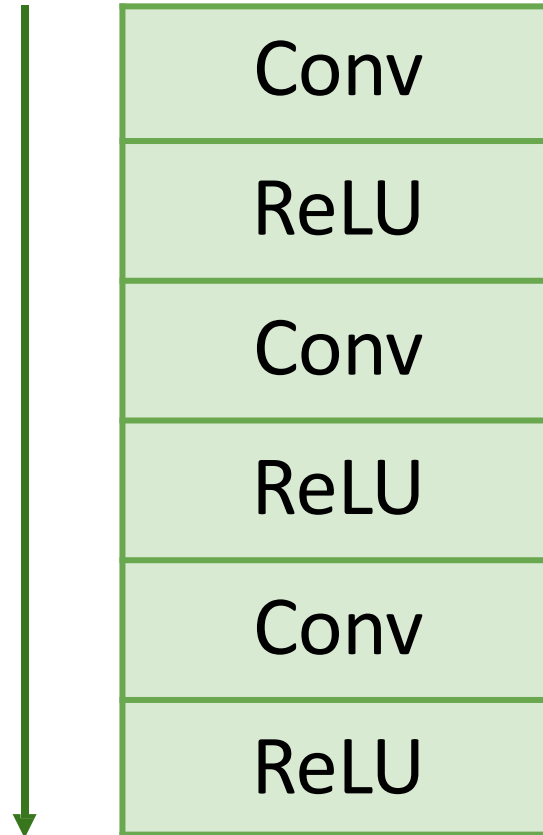
learning_rate = 1e-6
prev_loss = 5.0
for t in range(500):
    w2 = w2a if prev_loss < 5.0 else w2b
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

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

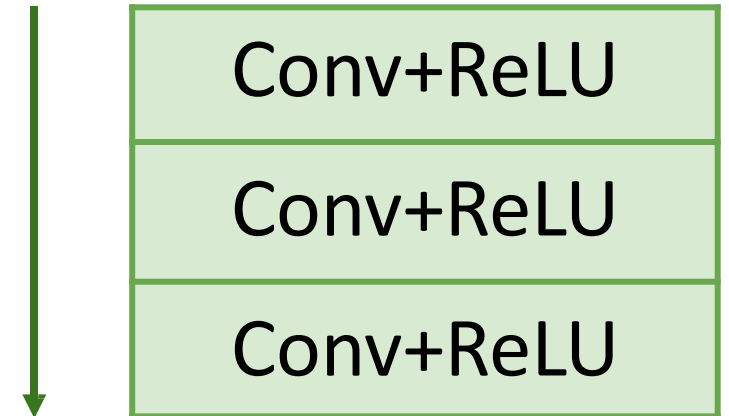
# Static vs Dynamic Graphs: Optimization

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: Serialization

- **Static**

- Once graph is built, can **serialize** it (save it as a standalone file) 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

Today's Star

**Q: PyTorch checkpoints are instances for serialization?**

**A: No.**

# PyTorch Checkpoints

## **Saving Model Weights and State:**

```
torch.save({'model_state_dict': model.state_dict(), 'epoch': epoch,}, "checkpoint.pth")
```

## **Reloading Checkpoints:**

```
model = MyModel() # You need to define the model architecture here.  
checkpoint = torch.load("checkpoint.pth")  
model.load_state_dict(checkpoint['model_state_dict'])
```

The checkpoints requires the model's definition to recreate the computation graph first.  
Otherwise, PyTorch can't interpret model state\_dict entries and will throw errors

# 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**

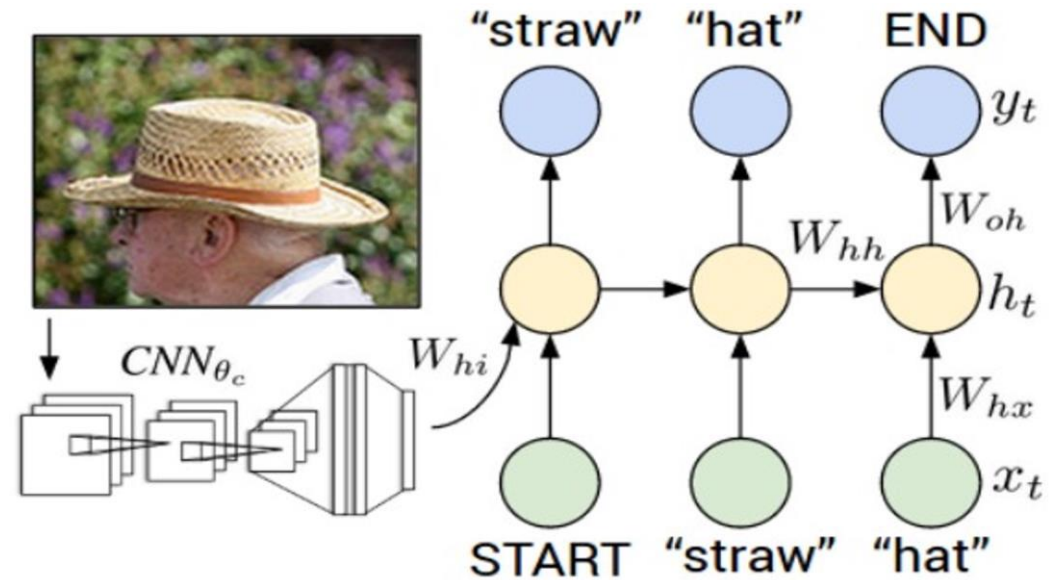
The code you write is the code that runs! Easy to reason about, debug, profile, etc

# Dynamic Graph Applications?

Your idea here!

# Dynamic Graph Applications

Model structure  
depends on the input:  
- Recurrent Networks

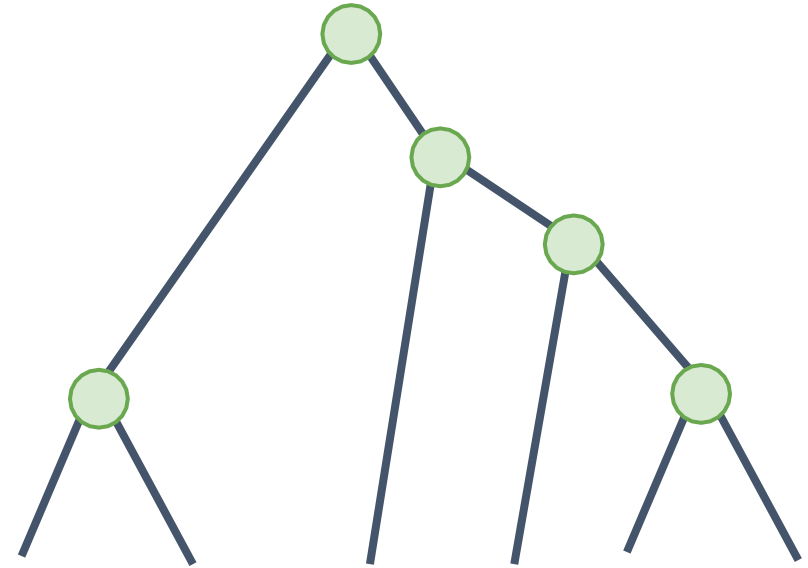




# Dynamic Graph Applications

Model structure  
depends on the input:

- Recurrent Networks
- Recursive Networks

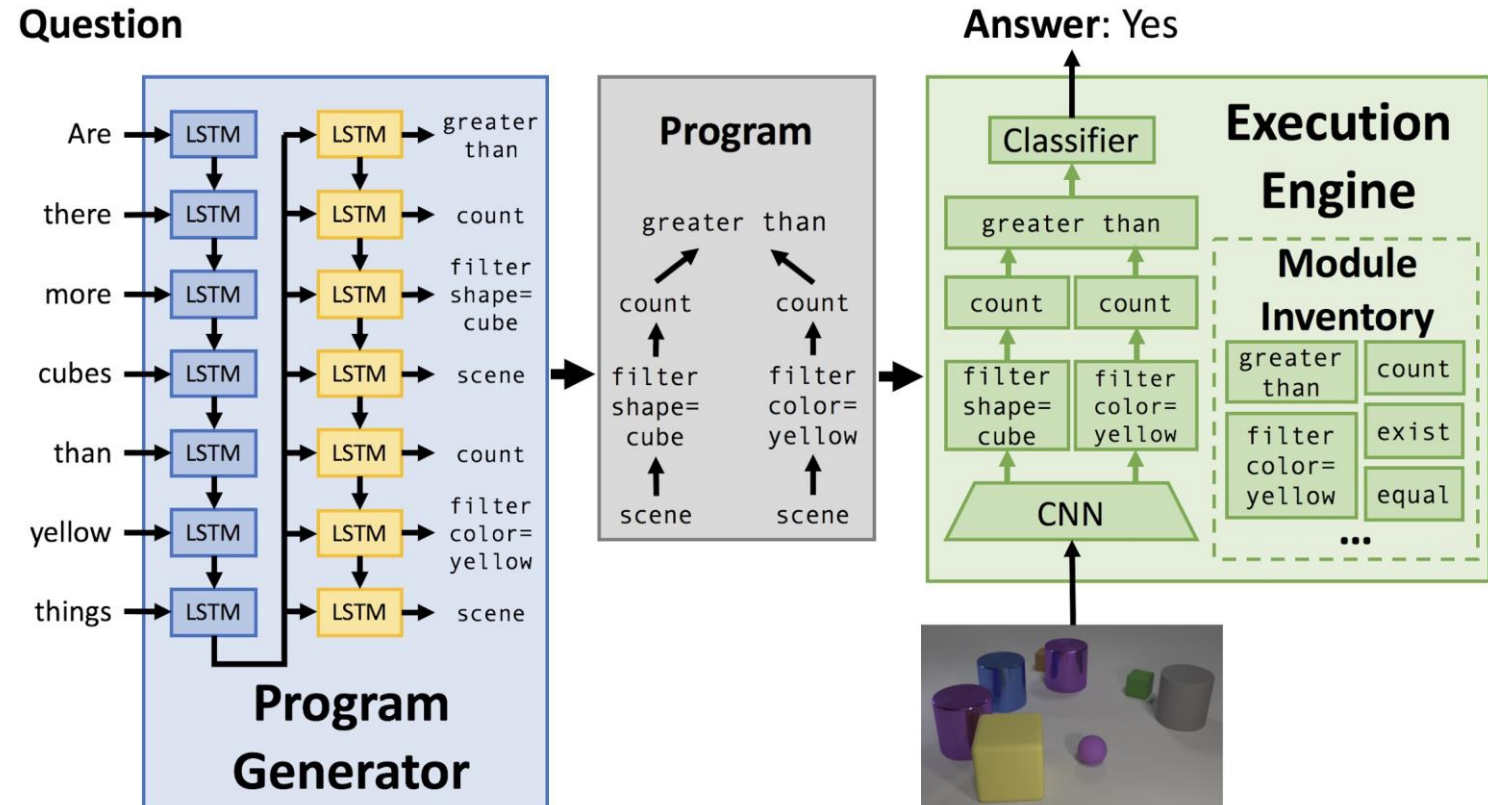


• The cat ate a big rat

# Dynamic Graph Applications

Model structure depends on the input:

- Recurrent Networks
- Recursive Networks
- Modular Networks



Andreas et al, "Neural Module Networks", CVPR 2016

Andreas et al, "Learning to Compose Neural Networks for Question Answering", NAACL 2016

Johnson et al, "Inferring and Executing Programs for Visual Reasoning", ICCV 2017