

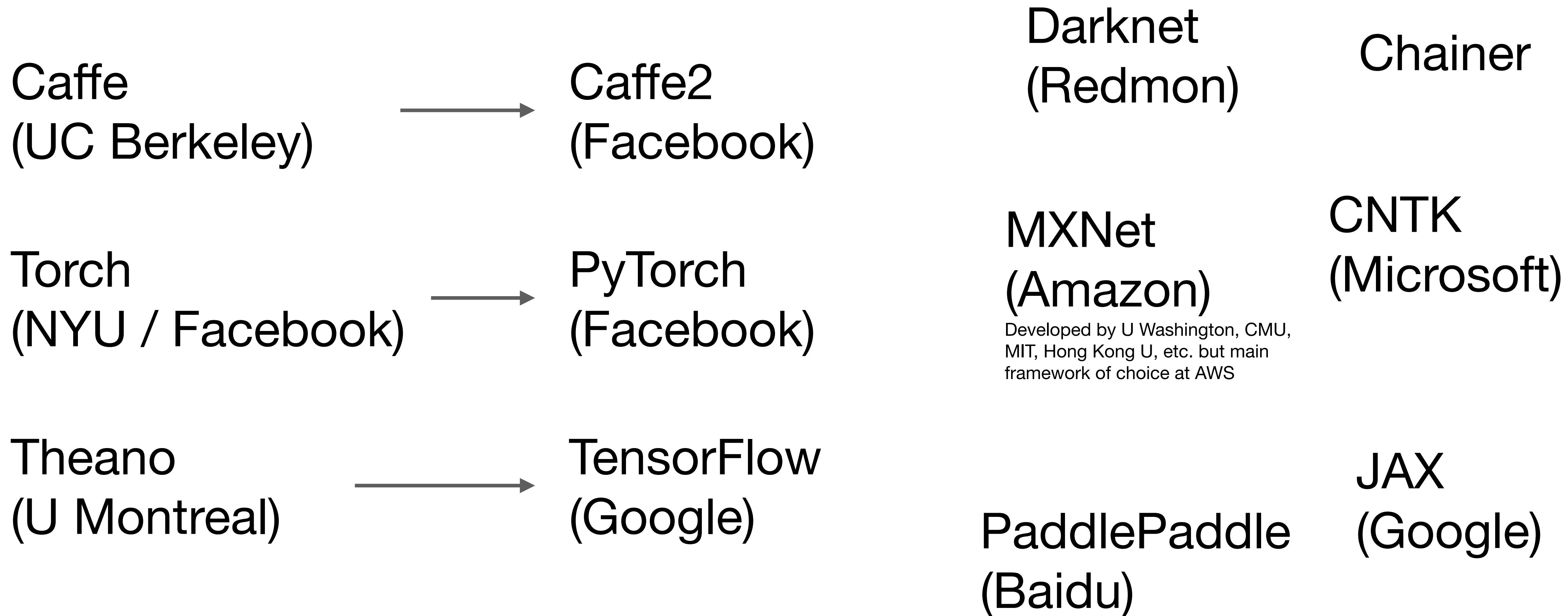


DeepRob

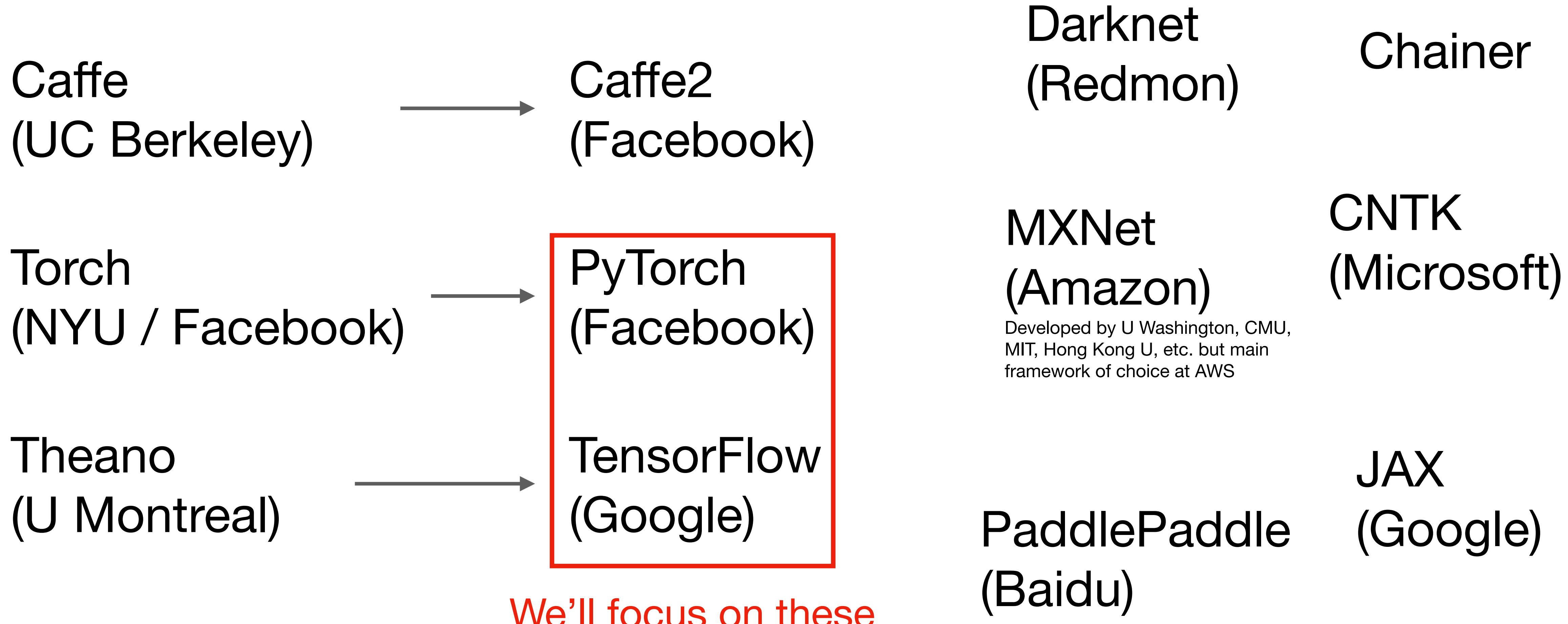
Deep Learning Software
University of Minnesota



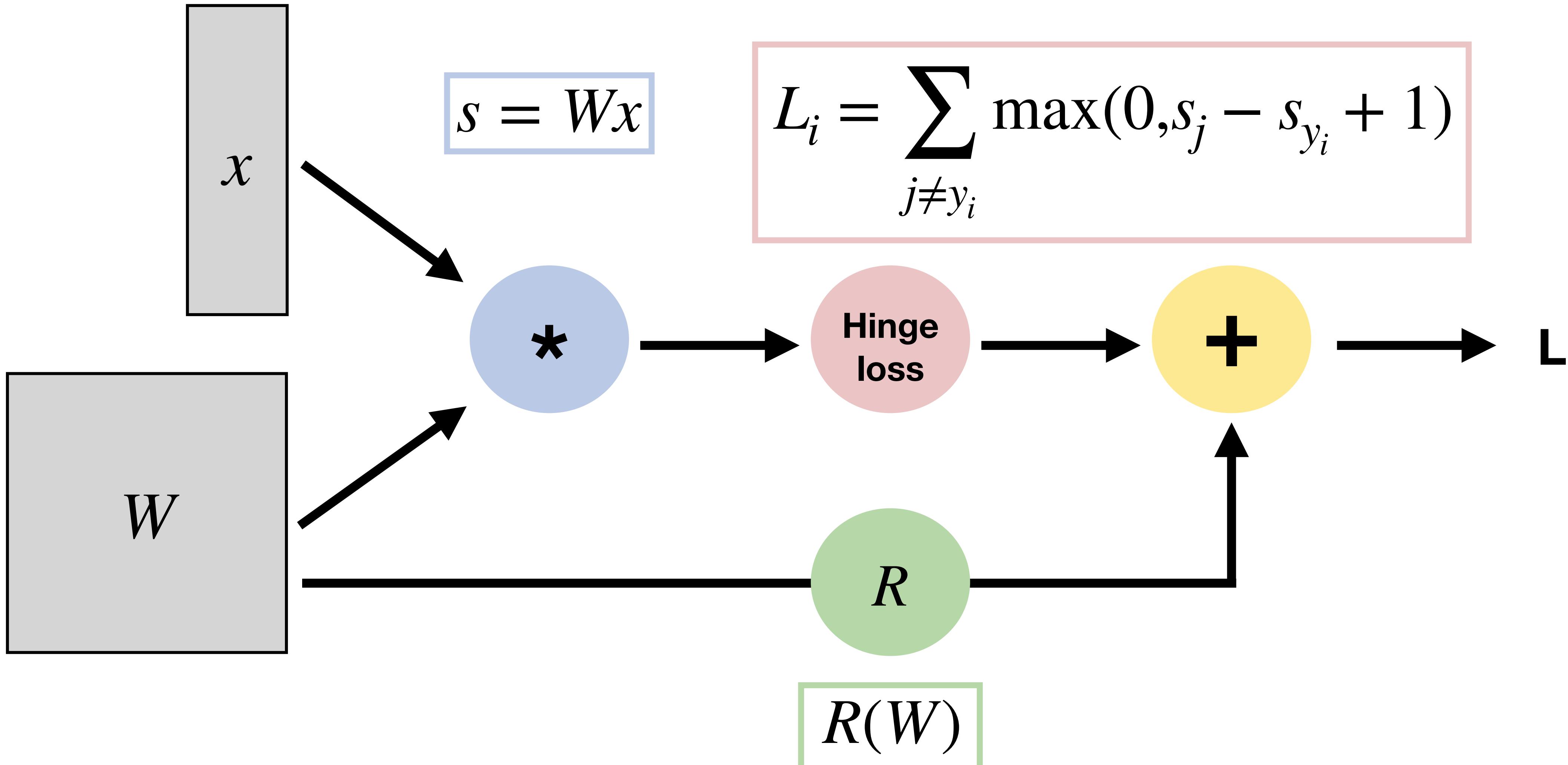
A zoo of frameworks!



A zoo of frameworks!



Recall: Computational Graphs





The motivation for 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 hardware





PyTorch





PyTorch: Versions

For this class we are using **PyTorch version 1.13**
(Released October 2022)

Be careful if you are looking at older PyTorch code—
the API changed a lot before 1.0



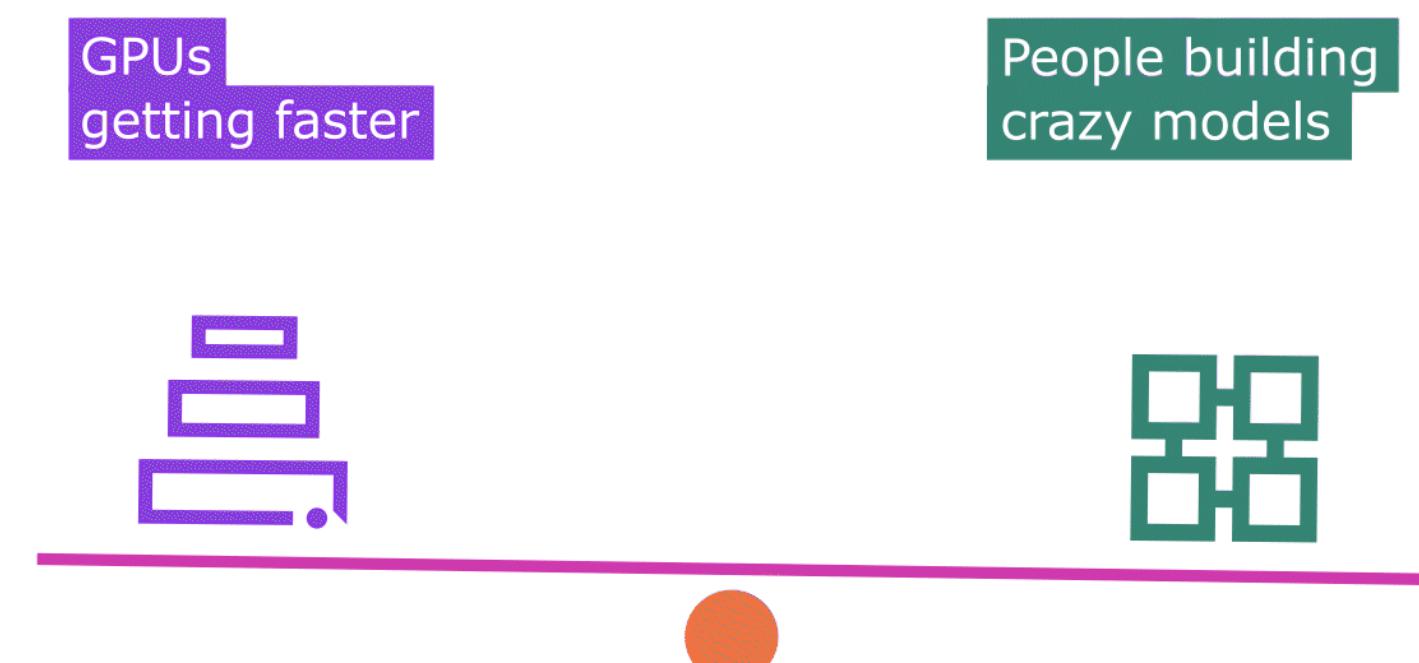


PyTorch: Version 2.0

Introduced to further optimize models (`torch.compile`)

Intended to be backwards compatible with 1.x

Expected stable release in March 2023



Video credit: [PyTorch](#)





PyTorch: Fundamental Concepts

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

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

Module: A neural network layer; may store state or learnable weights





PyTorch: Fundamental Concepts

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

P0, P1, P2

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

P3
P4
Final

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



```
import torch

device = torch.device('cpu')

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

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

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```





PyTorch: Tensors

Forward pass: compute predictions and loss



```
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()

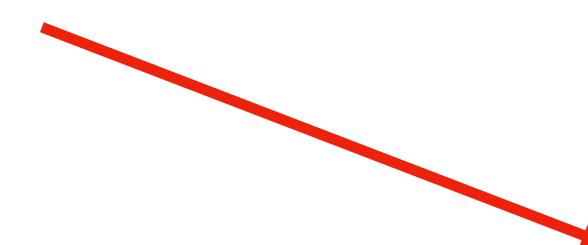
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    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('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

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





PyTorch: Autograd

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)

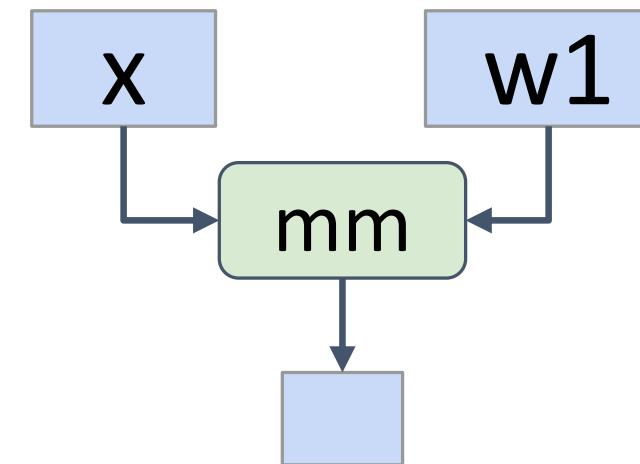
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    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

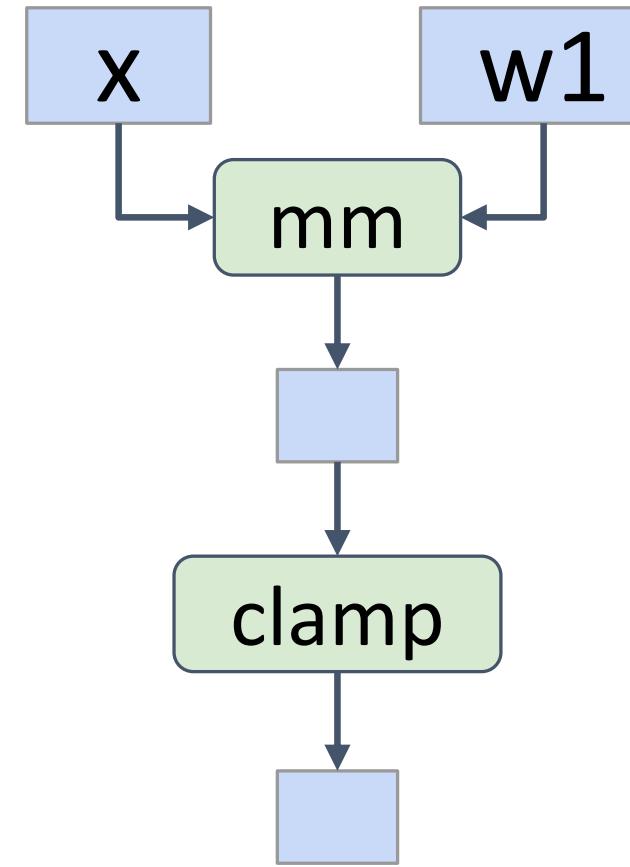
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

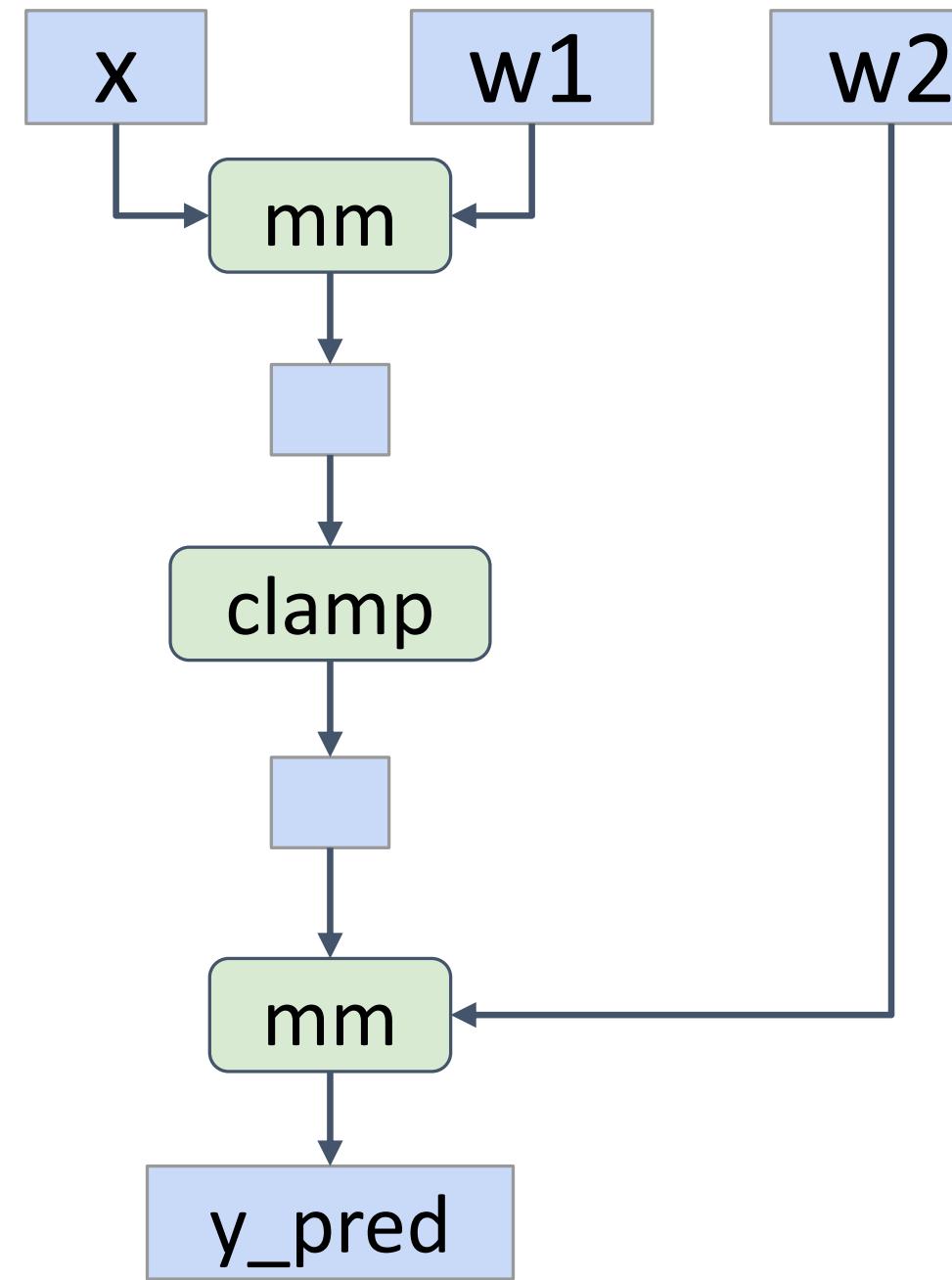
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

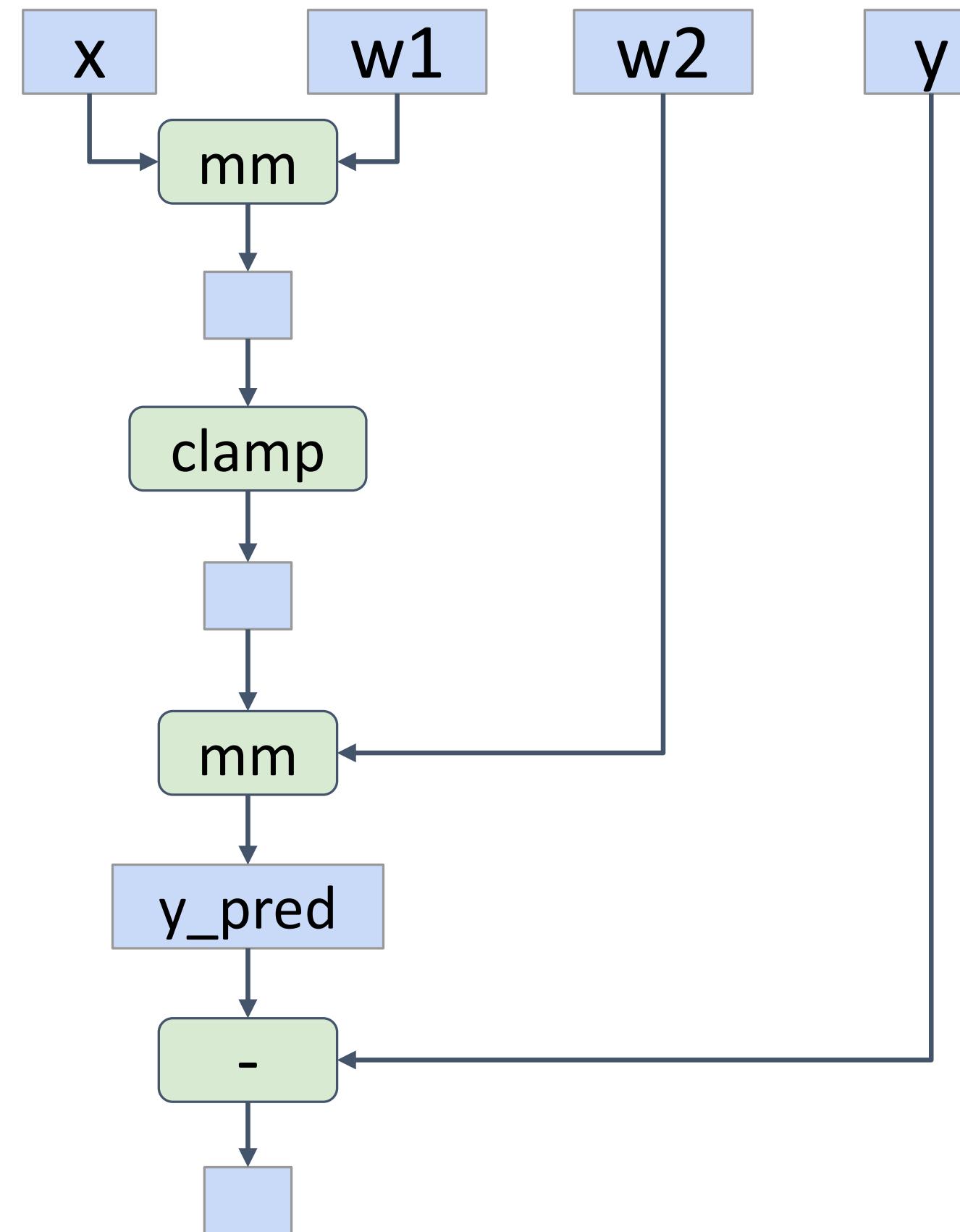
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

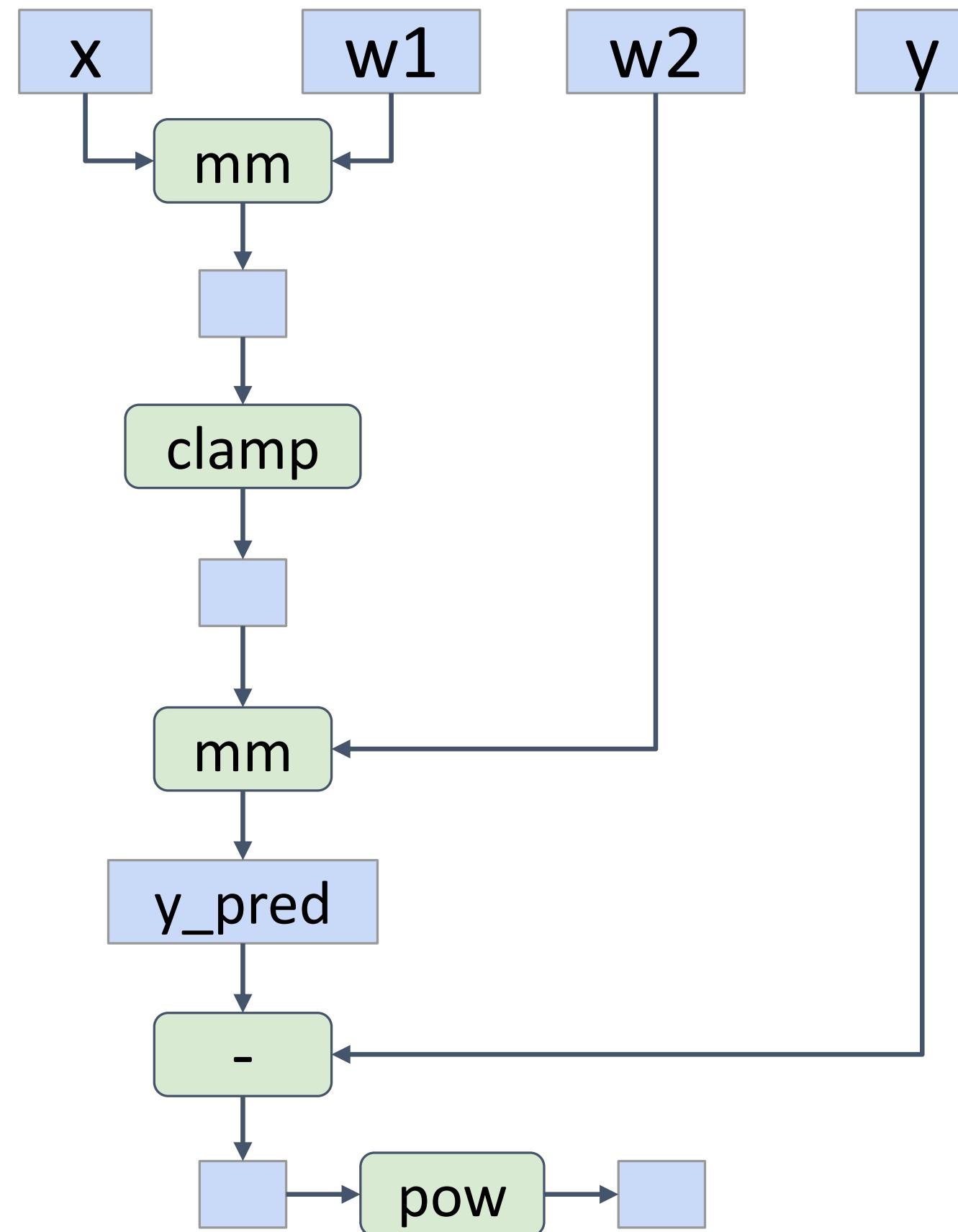
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

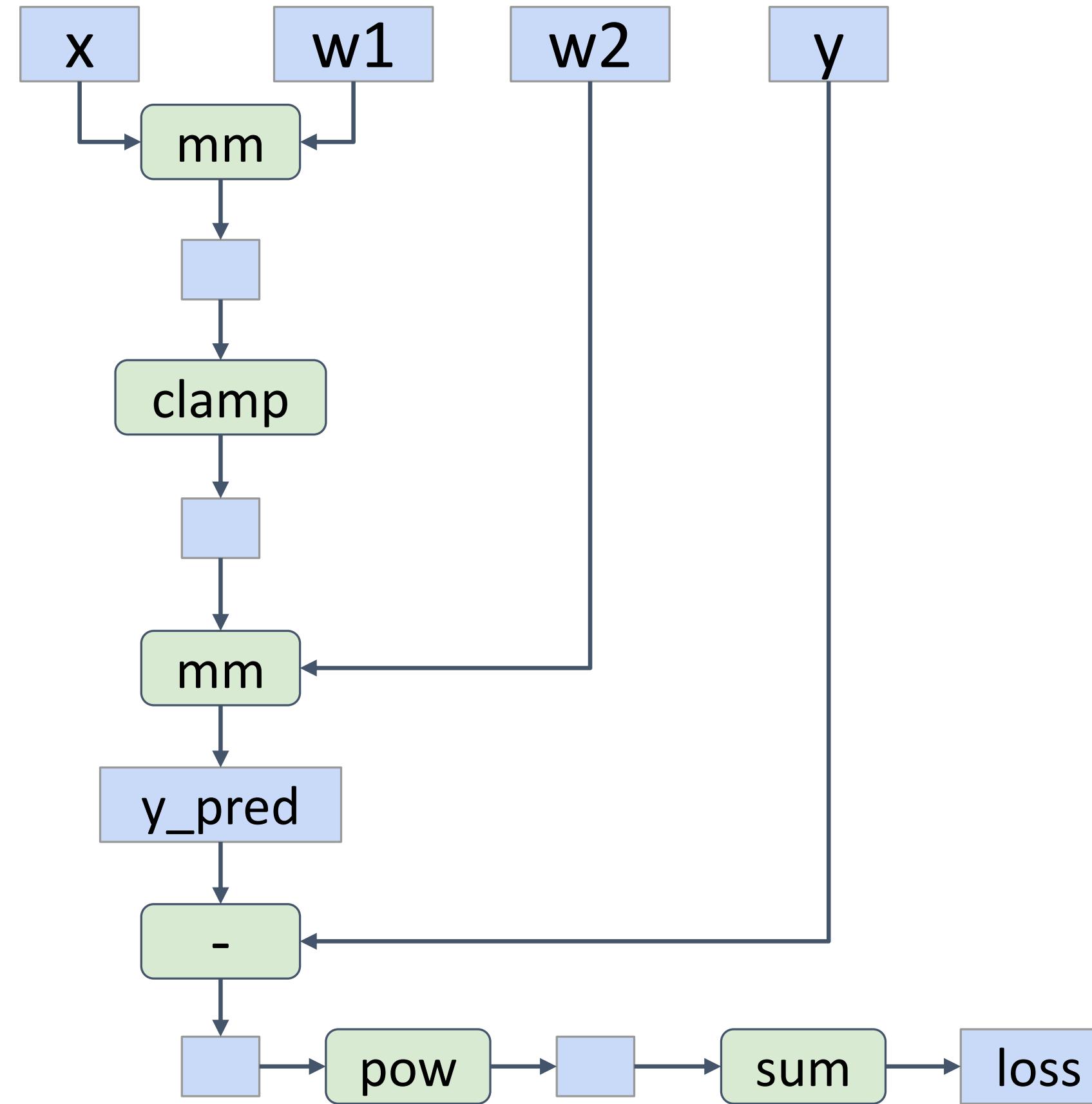
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

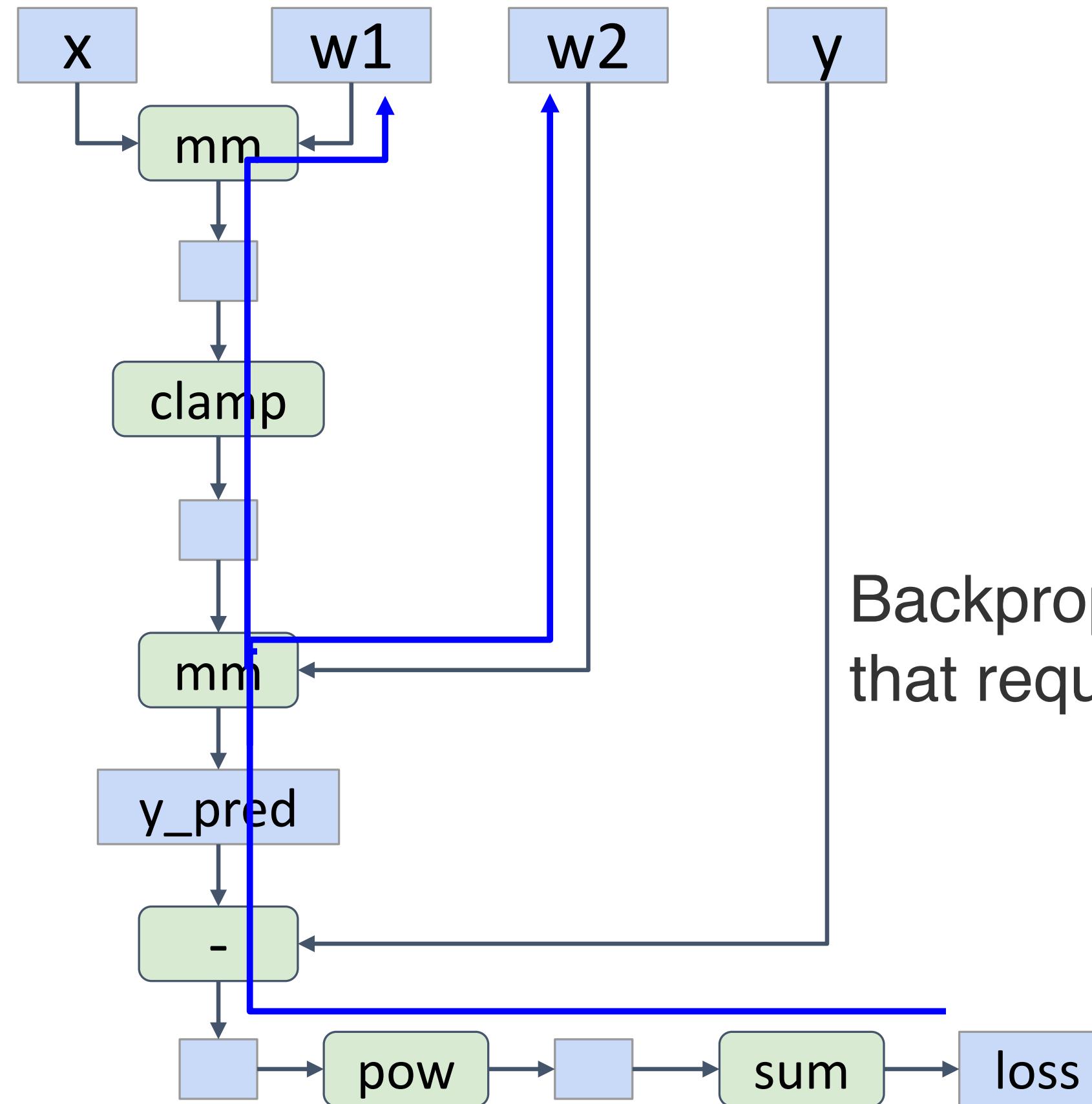
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

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())
```

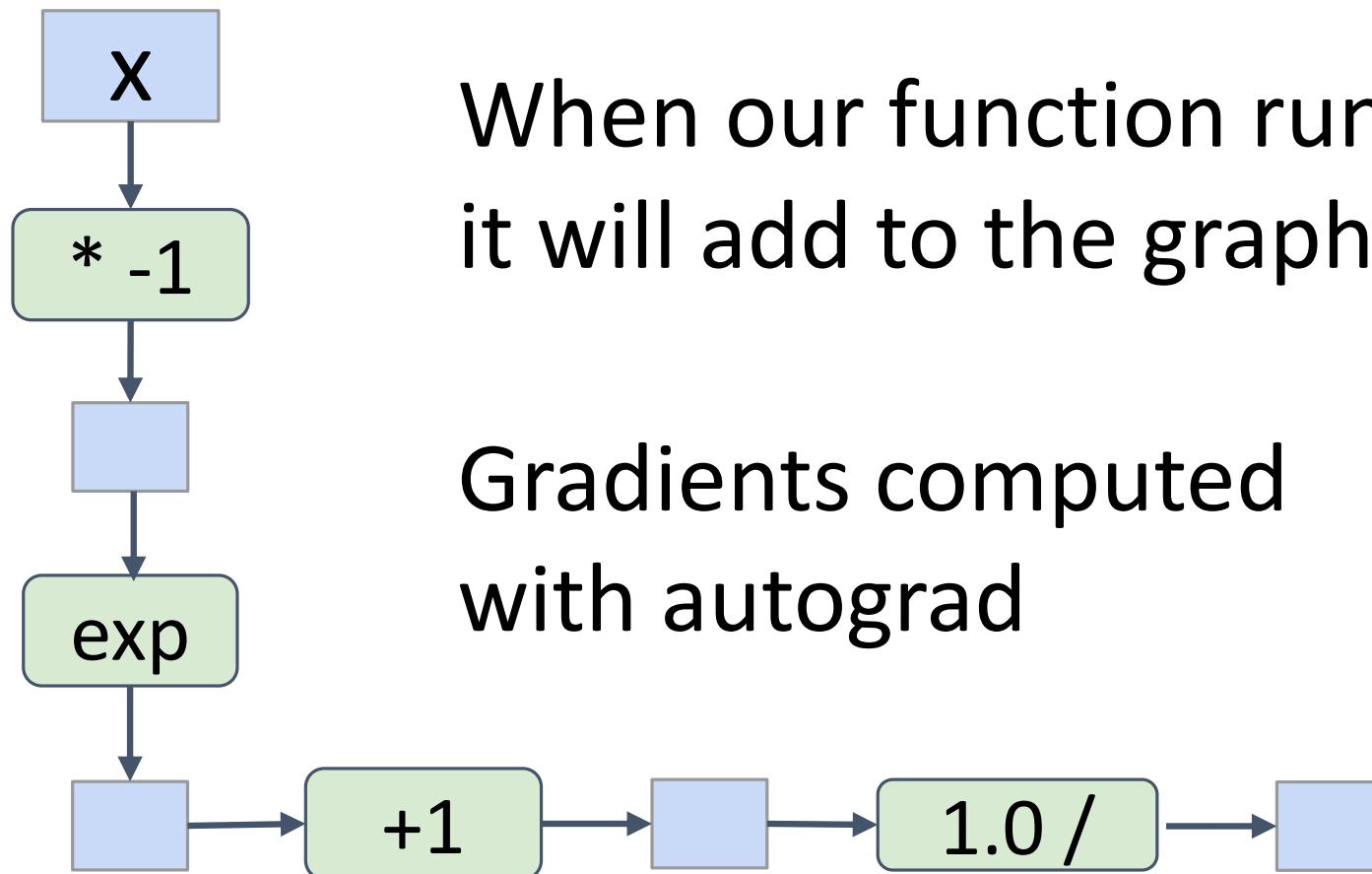
```
import torch  
  
N, D_in, H, D_out = 64, 1000, 100, 10  
  
x = torch.randn(N, D_in)  
y = torch.randn(N, D_out)  
y = torch.randn(N, D_out)  
w1 = torch.randn(D_in, H, requires_grad=True)  
w2 = torch.randn(H, D_out, requires_grad=True)  
  
learning_rate = 1e-6  
for t in range(500):  
    y_pred = 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())
```



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()

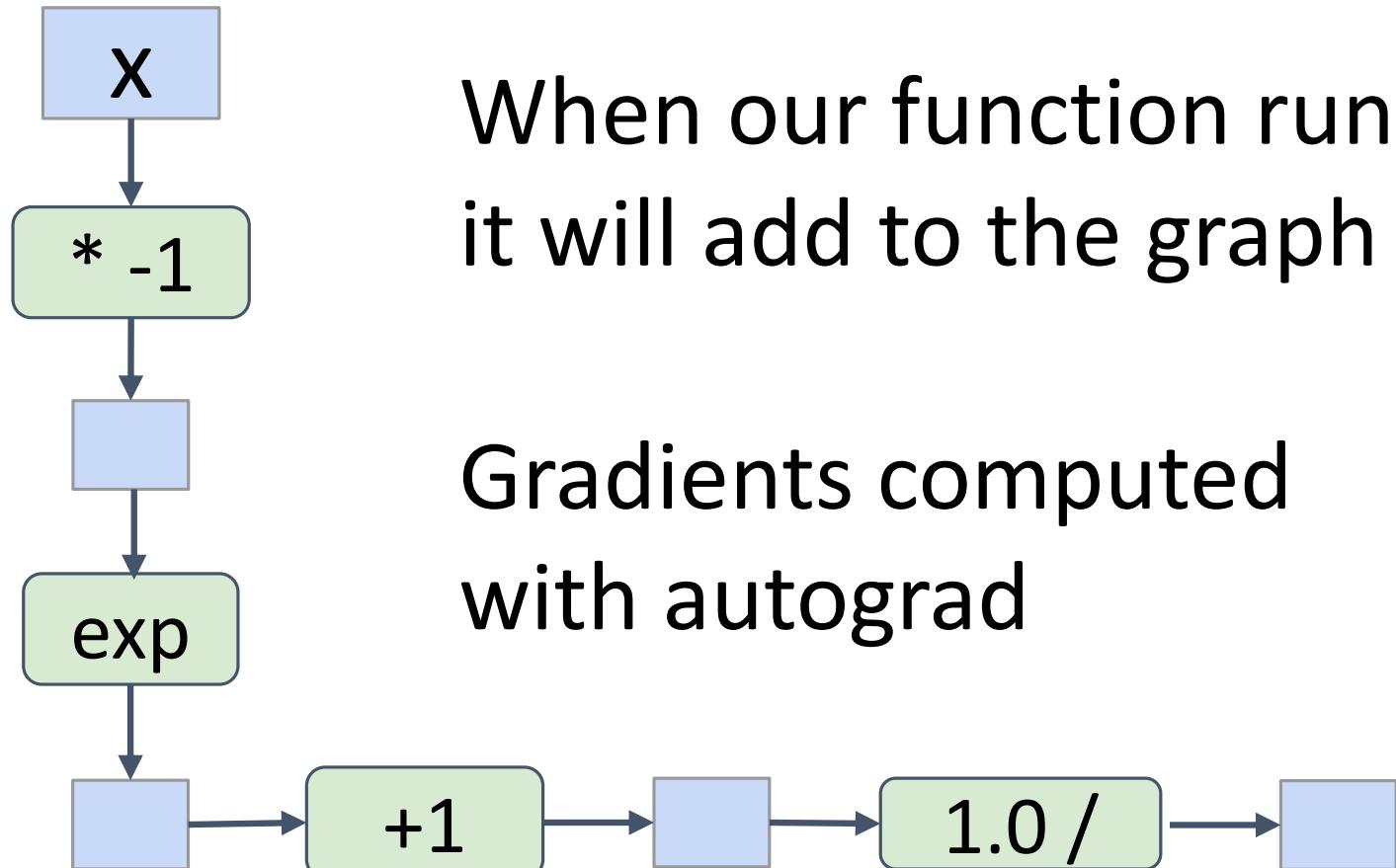
    loss.backward()
    if t % 50 == 0:
        print(t, loss.item())

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

PyTorch: New Functions

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

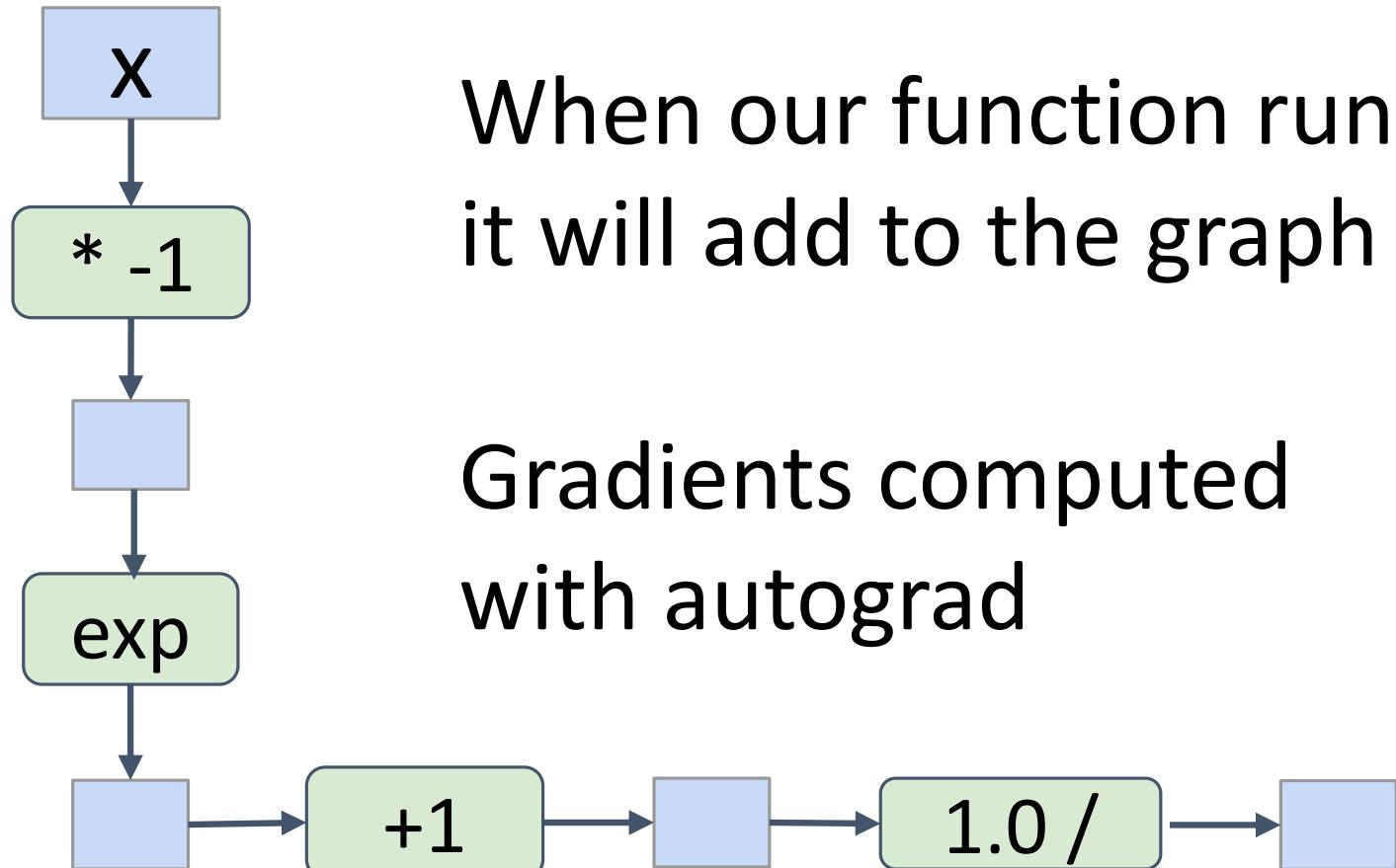
def sigmoid(x):
    return Sigmoid.apply(x)
```

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())
```



When our function runs,
it will add to the graph

Gradients computed
with autograd

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

```
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)
```

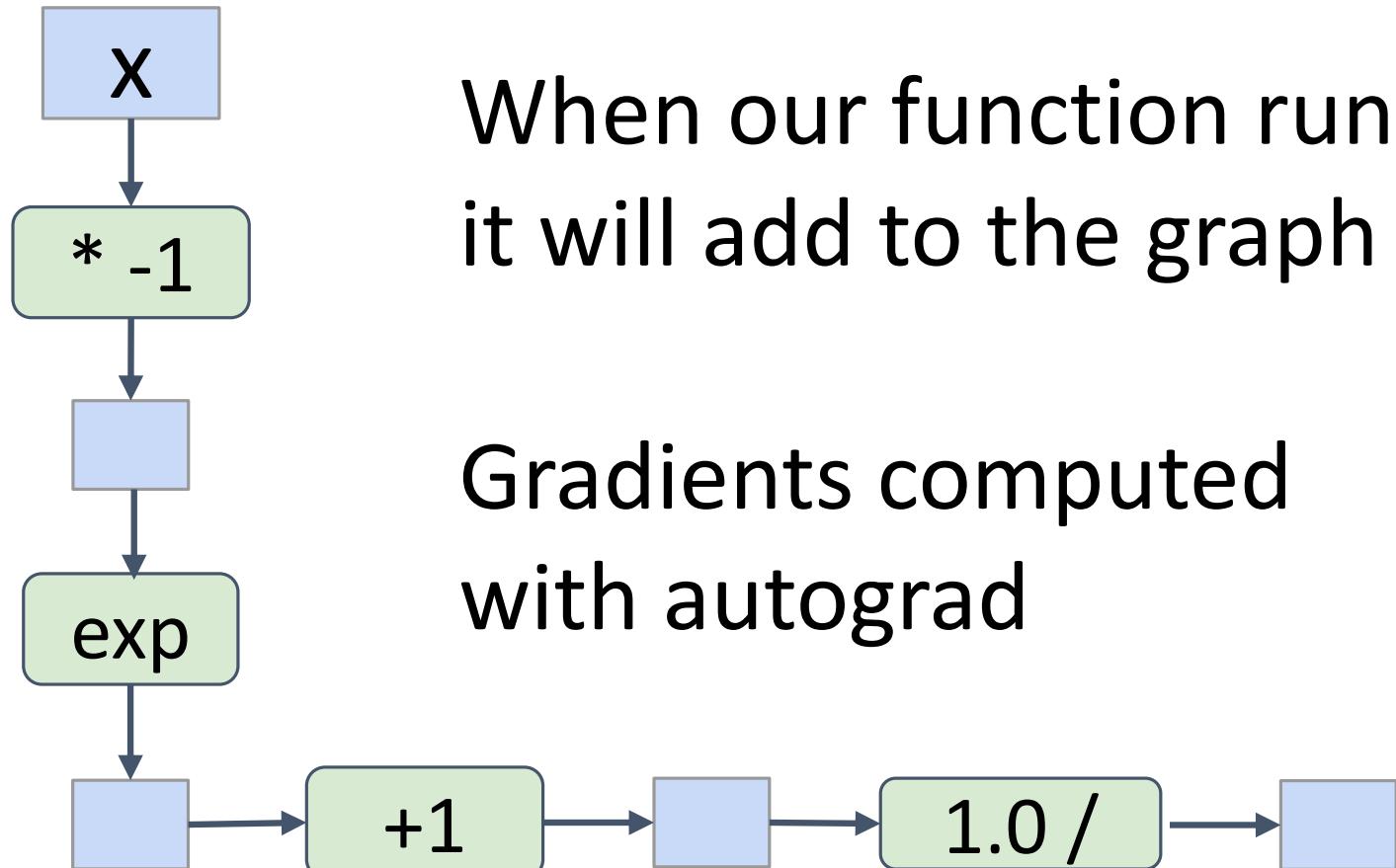
Now when our function runs,
it adds one node to the graph!



PyTorch: 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

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)
```

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



PyTorch: nn

Higher-level wrapper for working with neural nets

Use this! It will make your life easier

```
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)

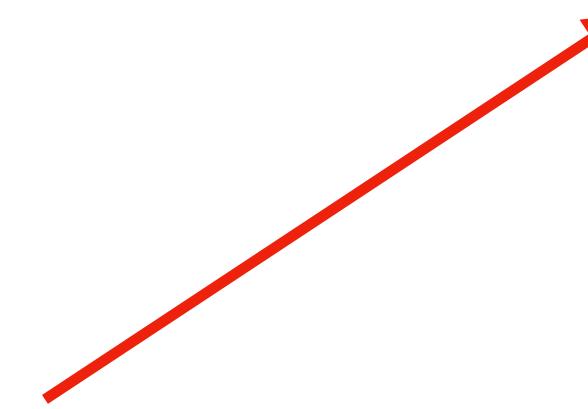
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```





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)

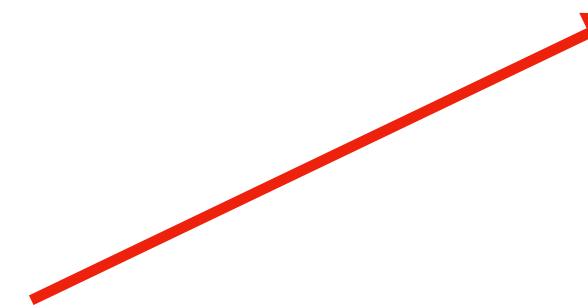
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```





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)

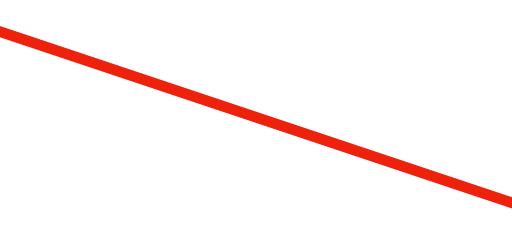
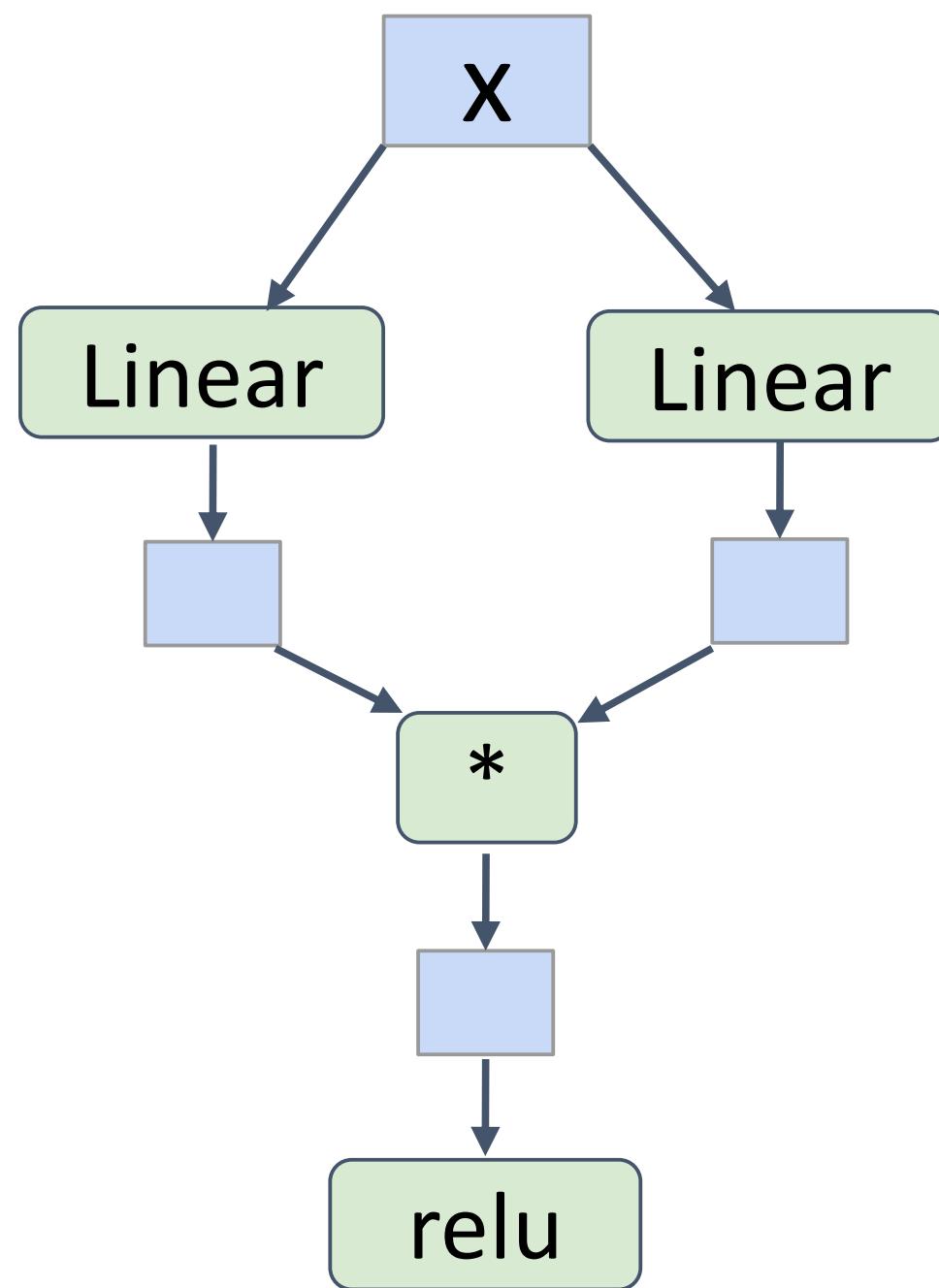
model = torch.nn.Sequential(
    ParallelBlock(D_in, H),
    ParallelBlock(H, H),
    torch.nn.Linear(H, D_out))

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)

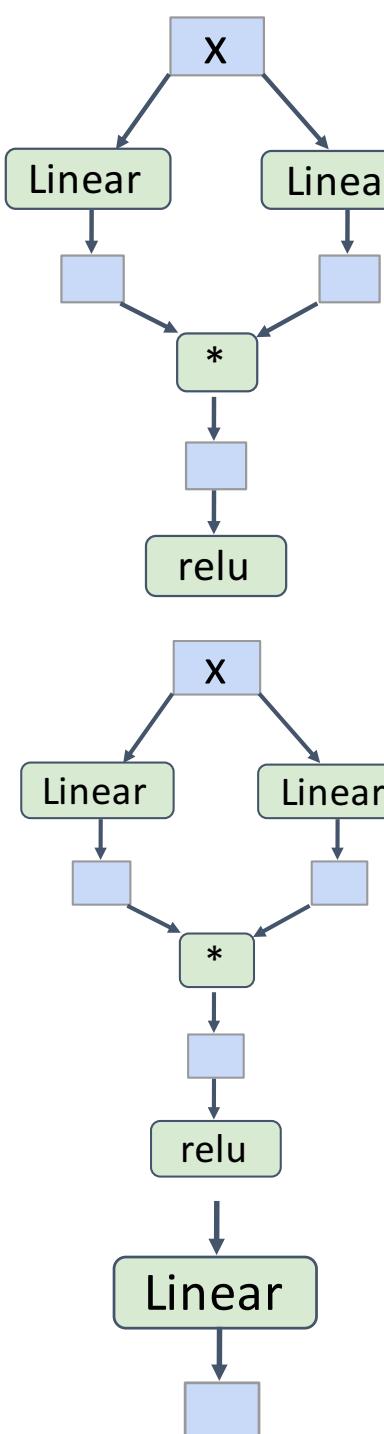
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

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: 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

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

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

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

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

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```



PyTorch: DataLoaders

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 Models

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)

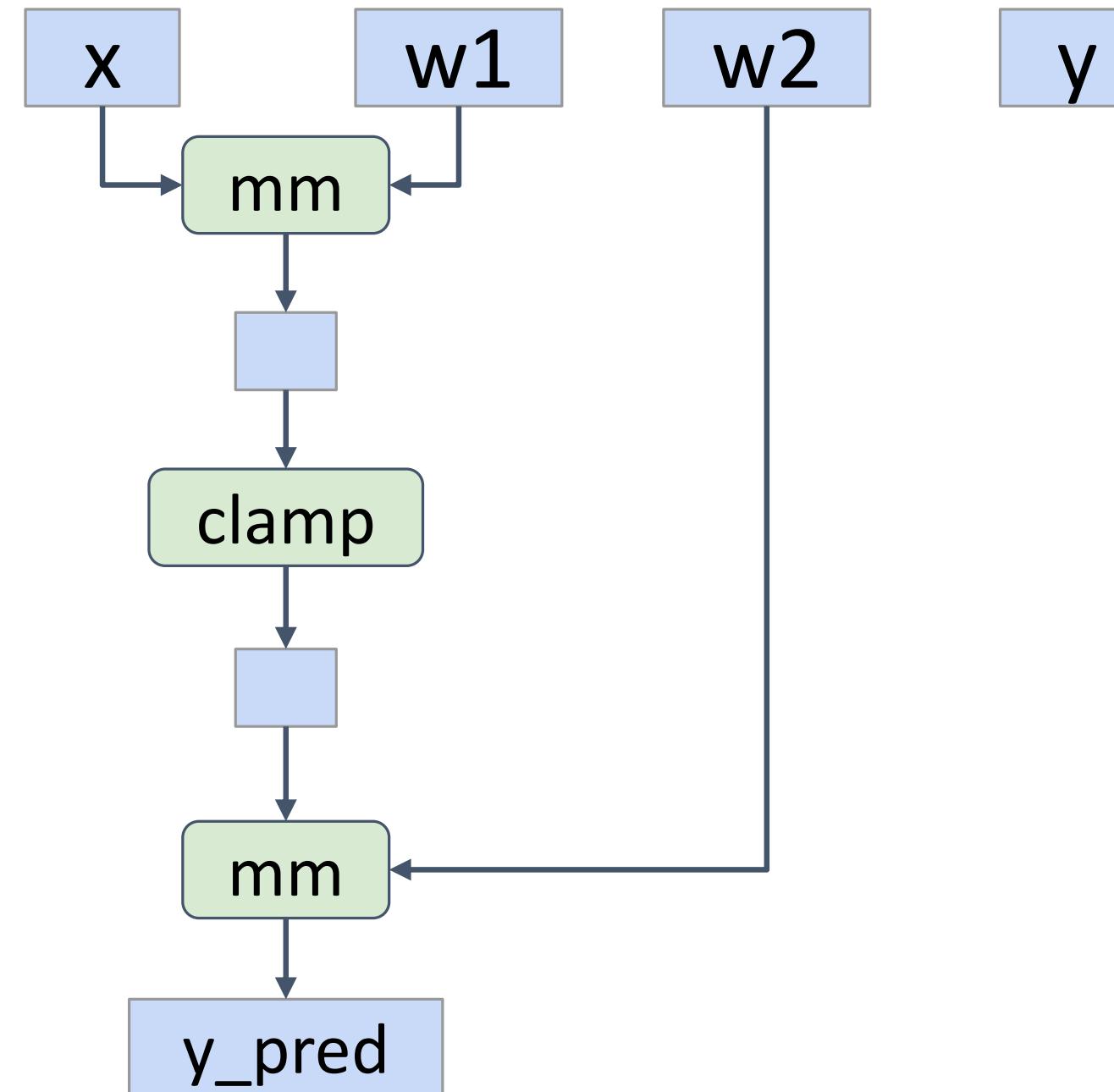
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Create Tensor objects



PyTorch: Dynamic Computation Graphs



```
import torch

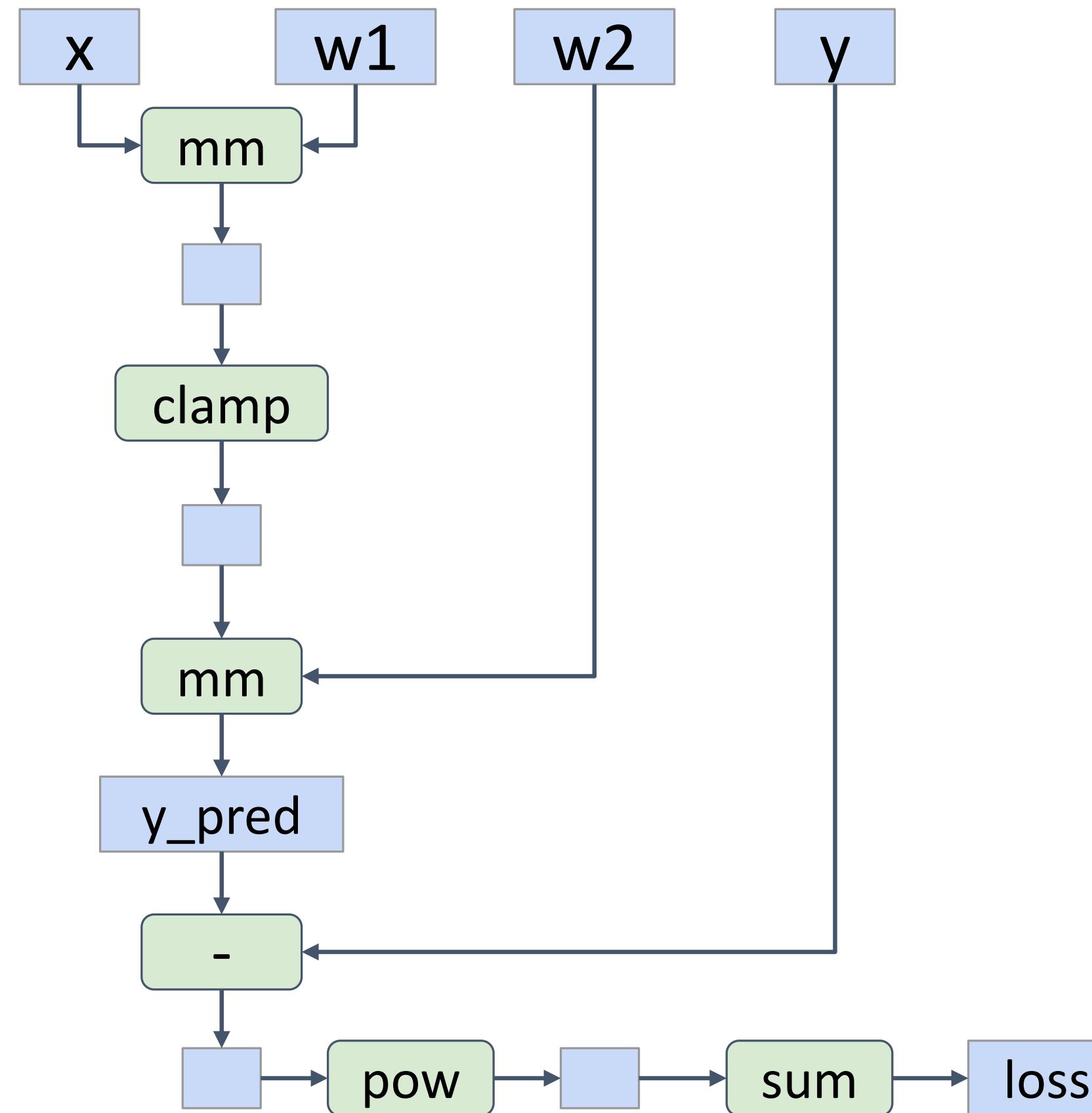
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

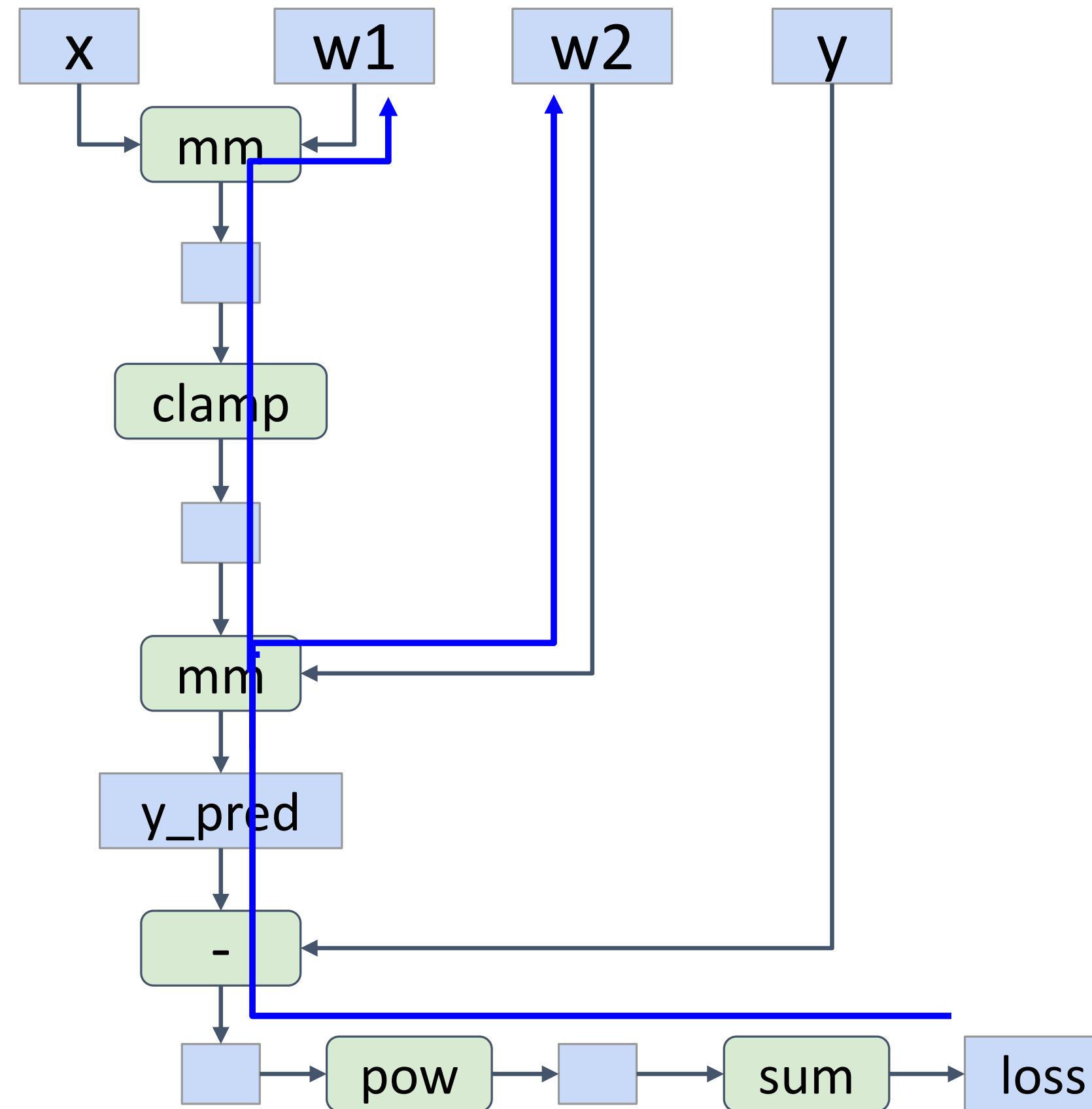
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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)

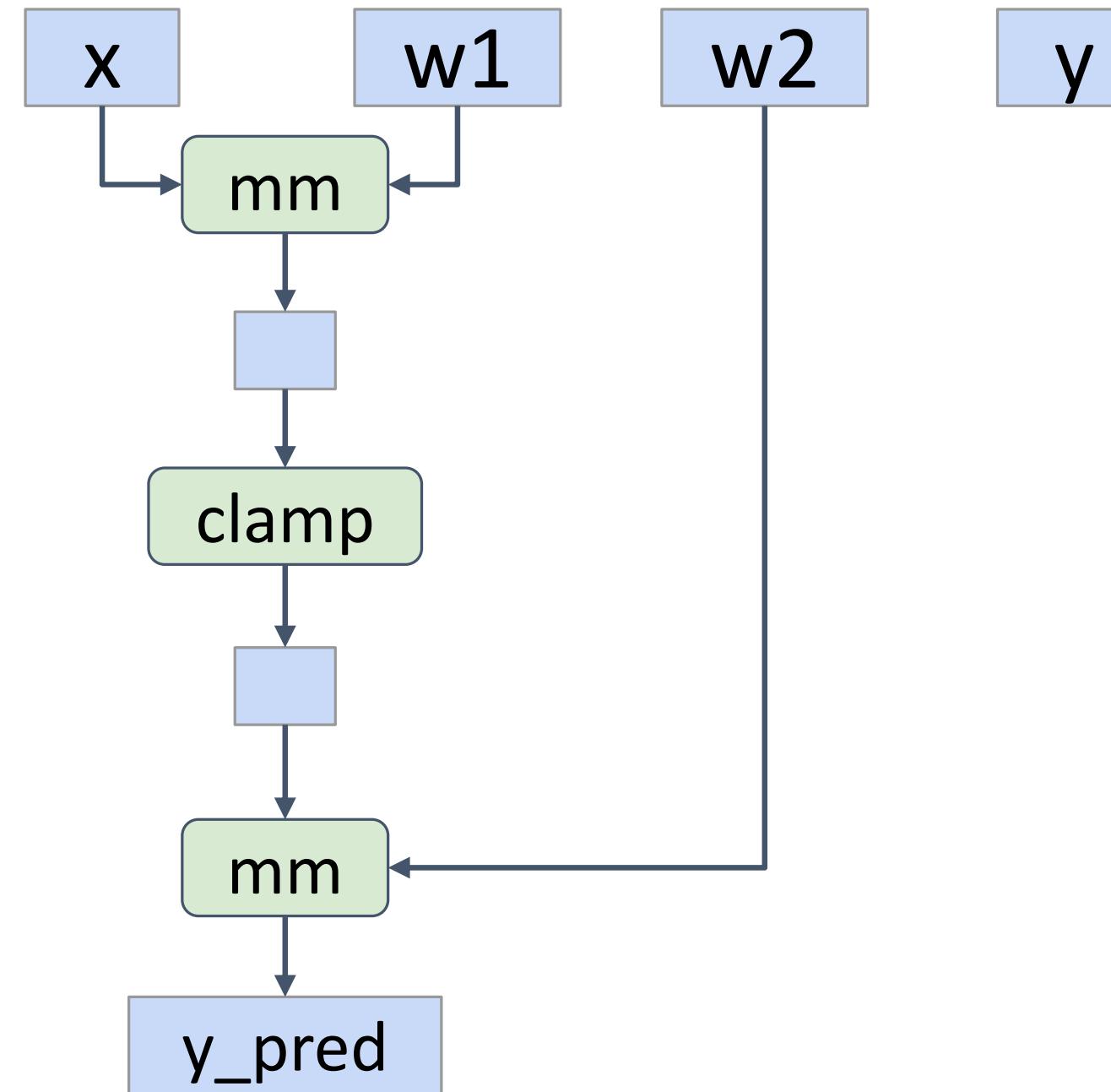
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Perform backprop,
throw away graph



PyTorch: Dynamic Computation Graphs



```
import torch

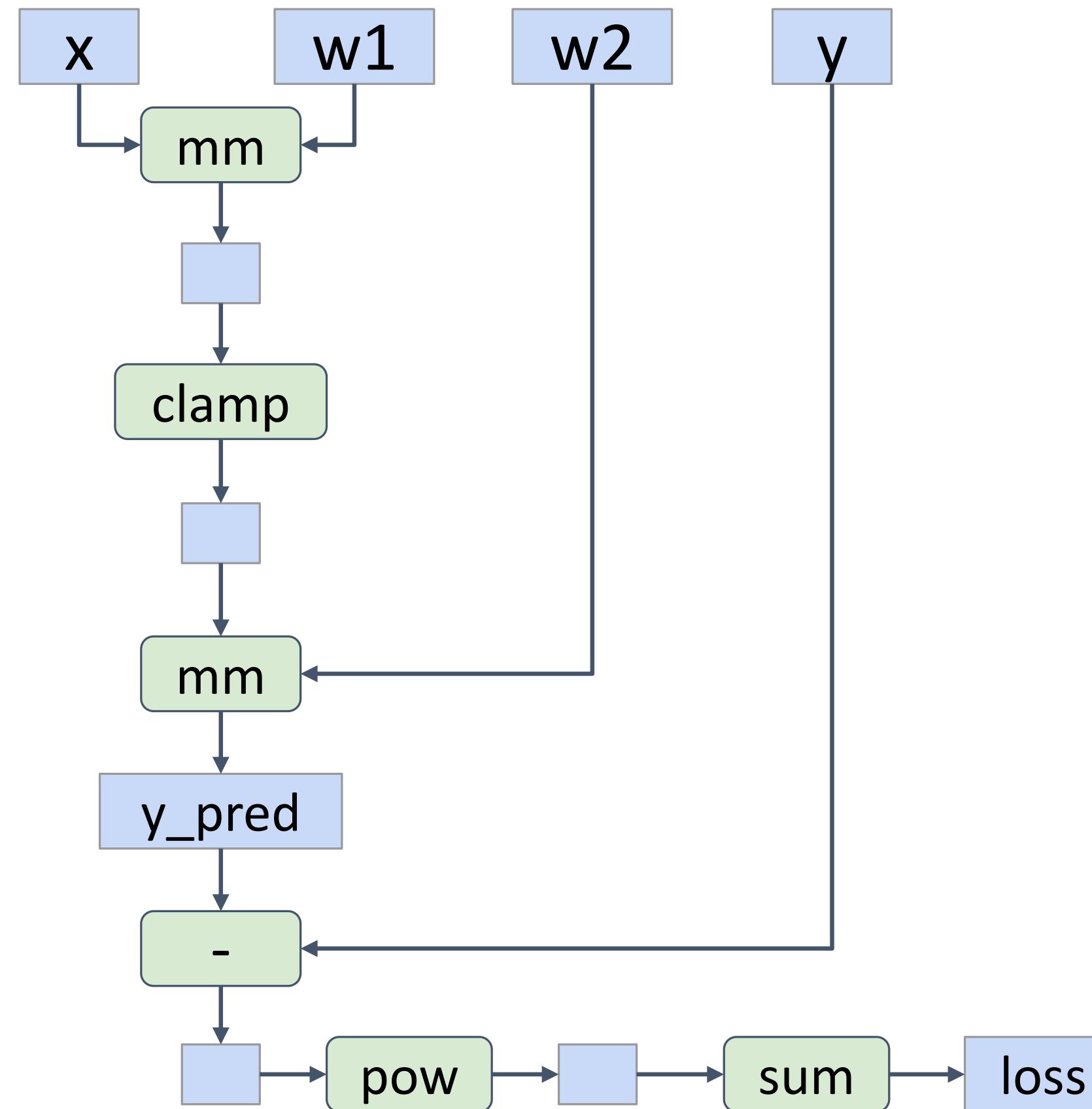
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

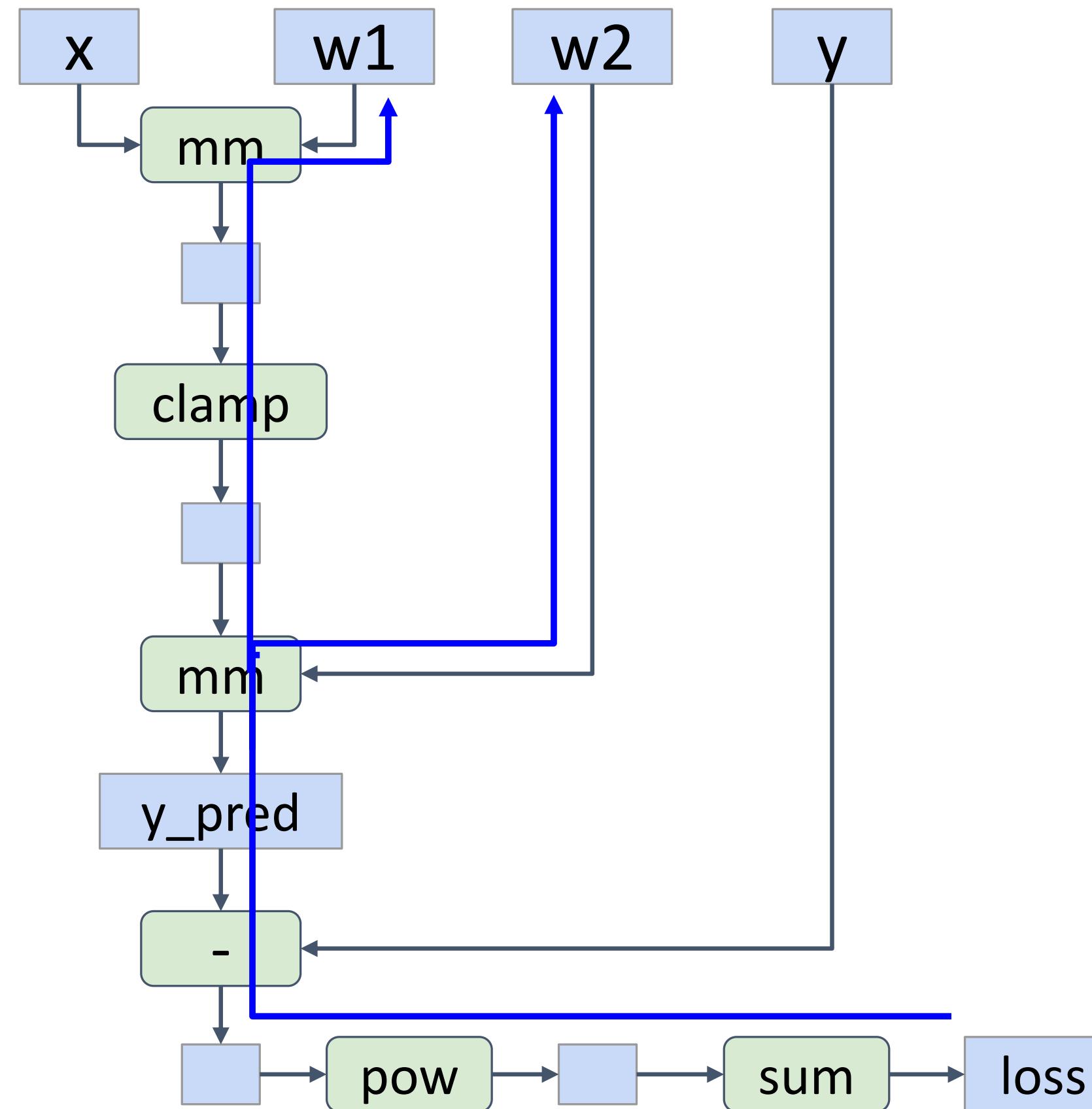
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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



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()
```

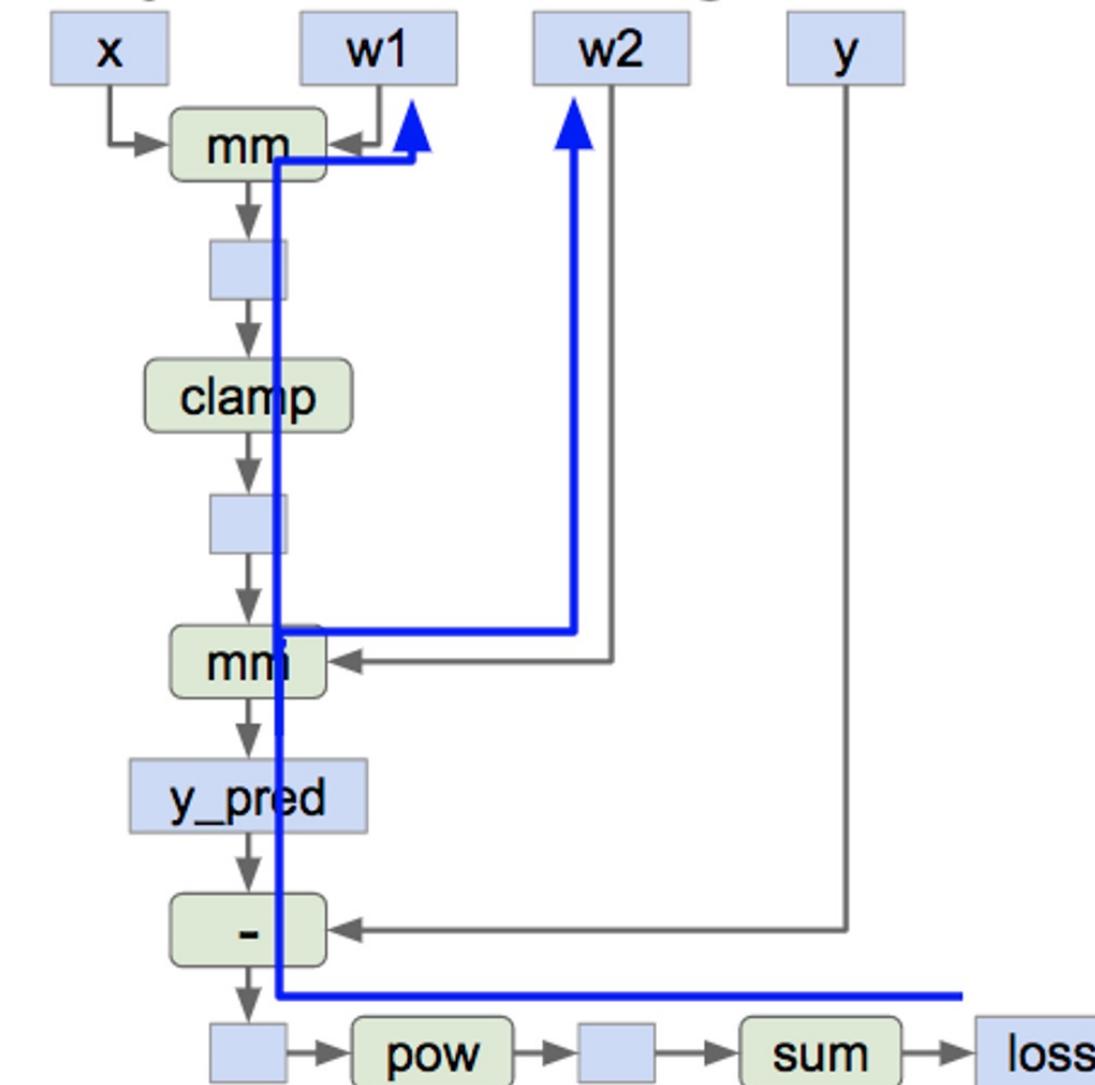


Alternative: Static Computation Graphs

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()
```





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)

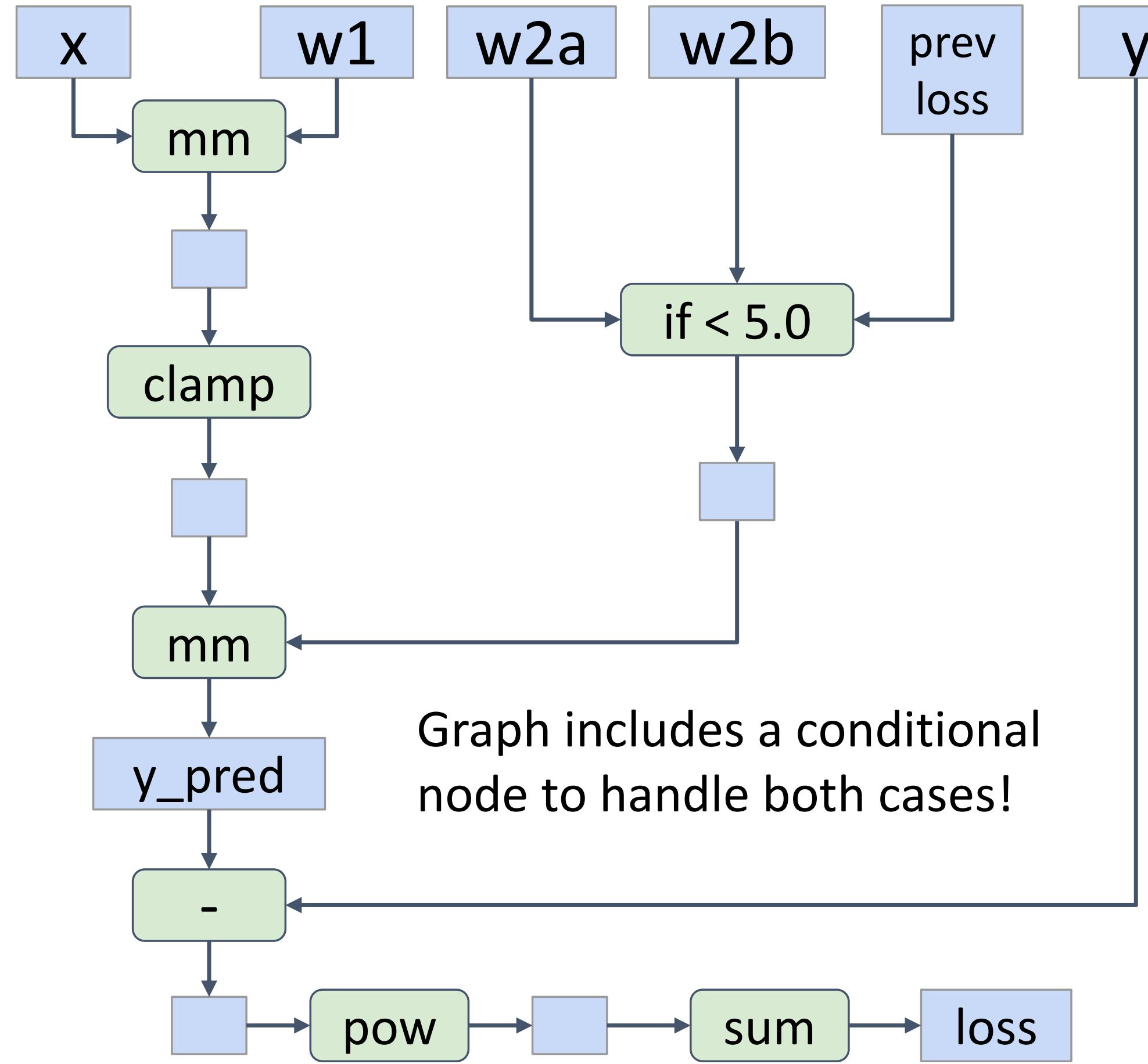
graph = torch.jit.script(model)

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()
```



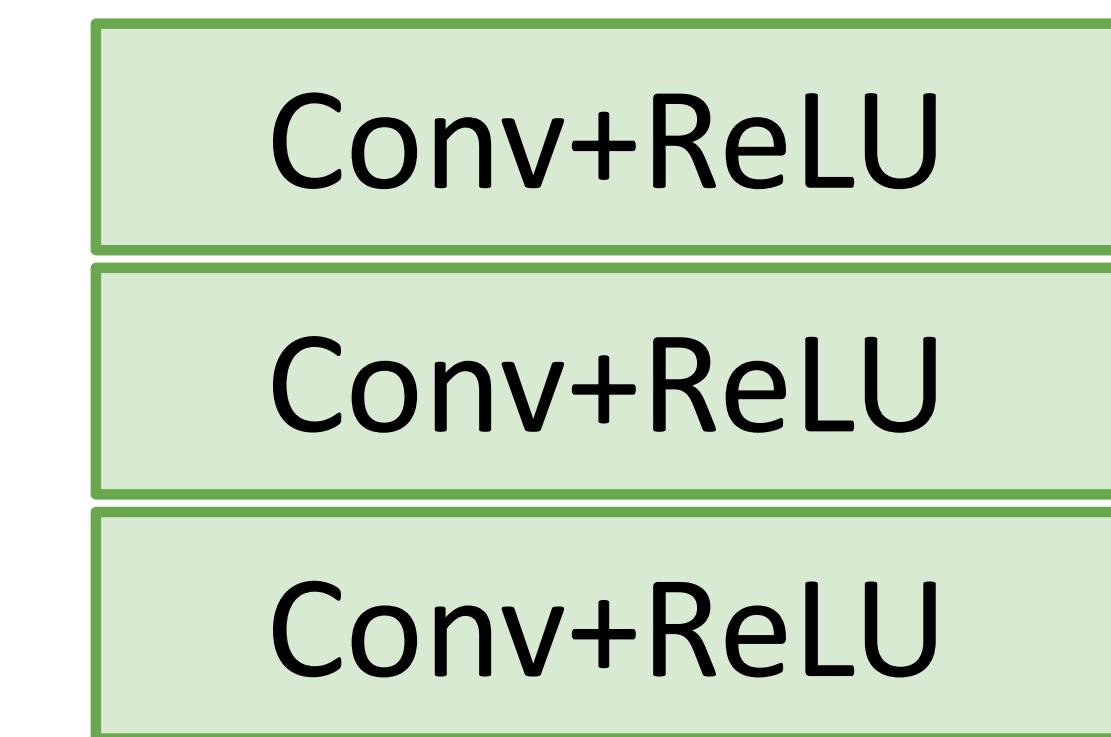
Static vs Dynamic Graphs: Optimization

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

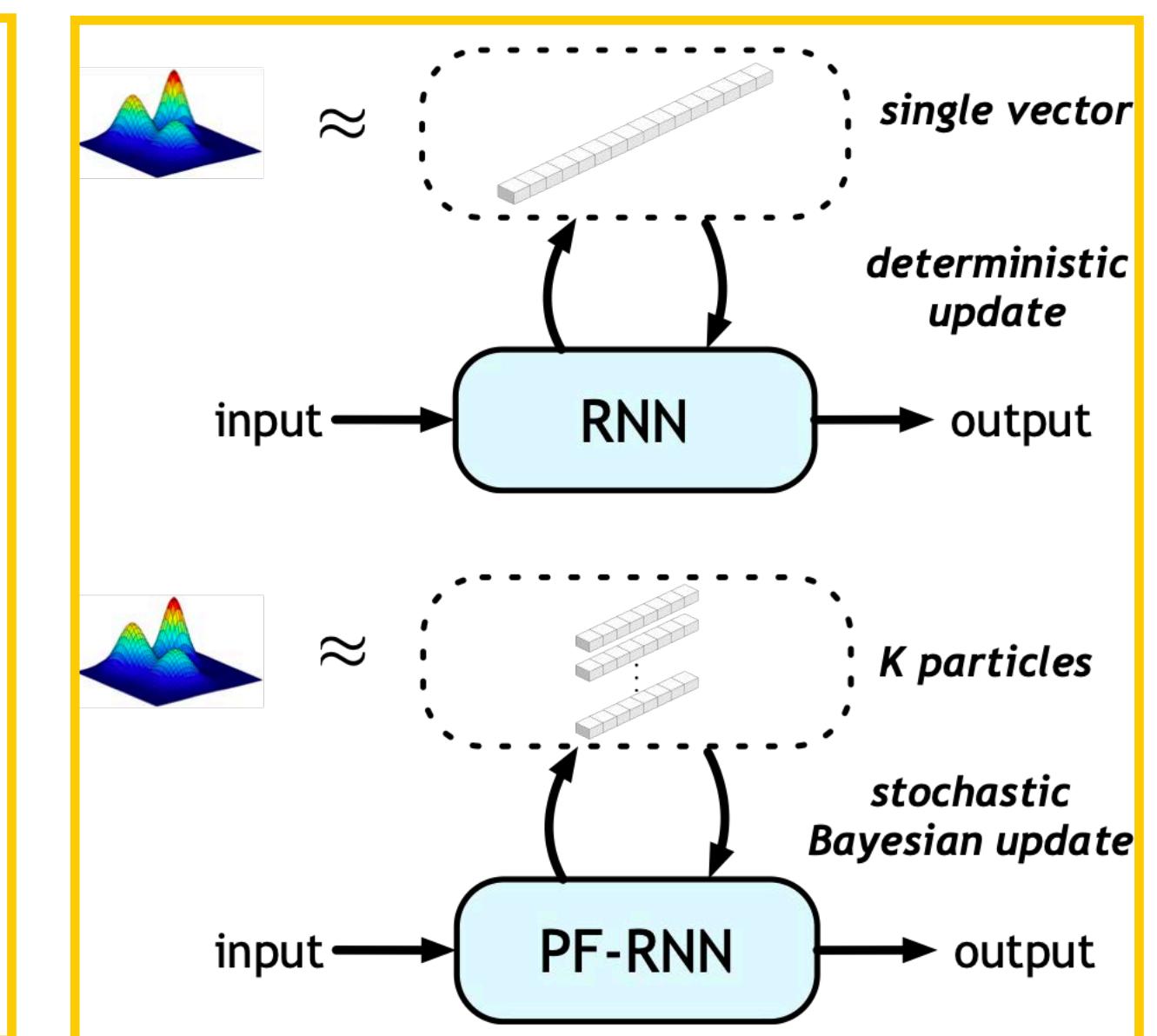
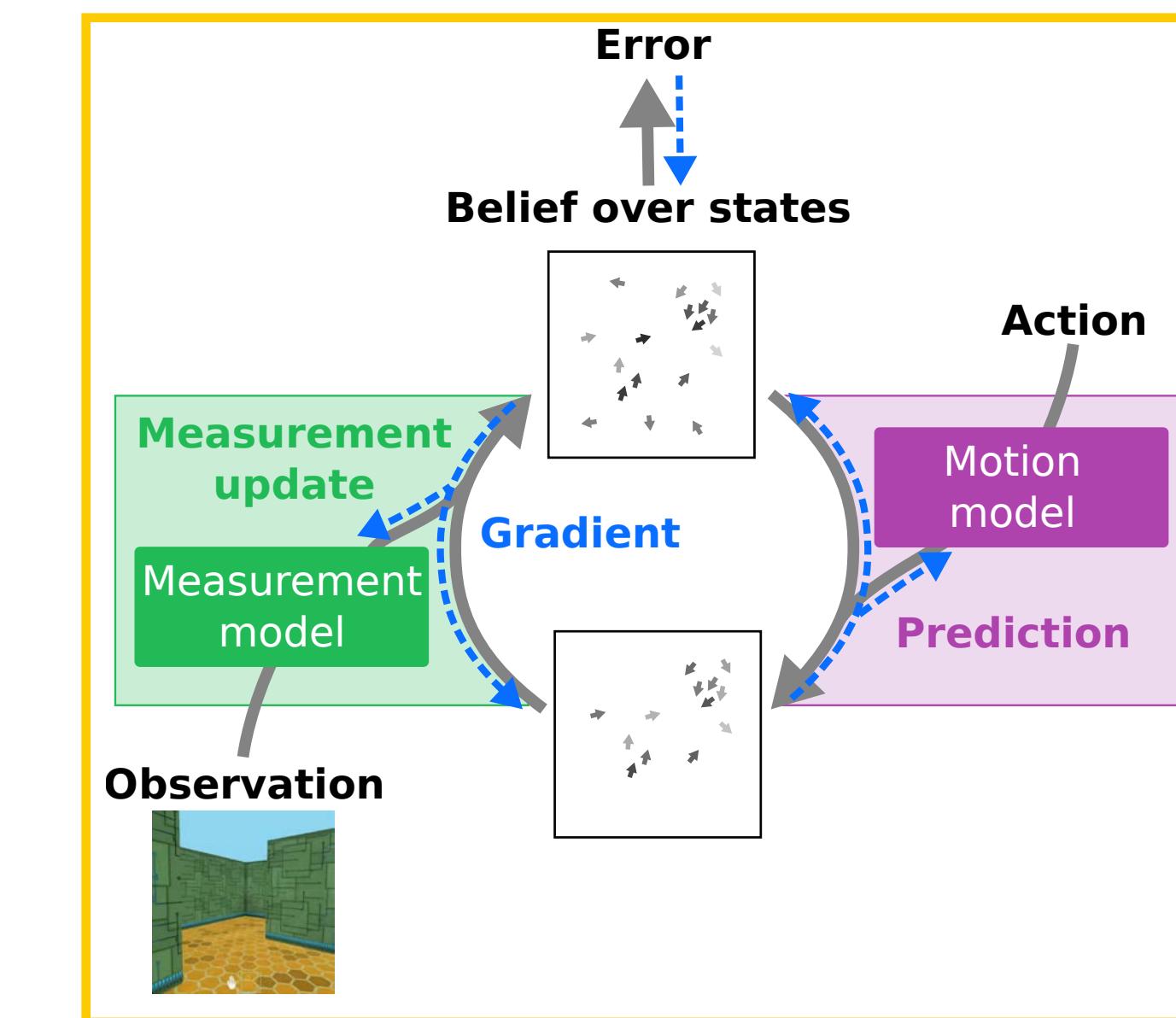
The code you write is the code that runs! Easy to reason about, debug, profile, etc



Dynamic Graph Applications

Model structure
depends on the input:

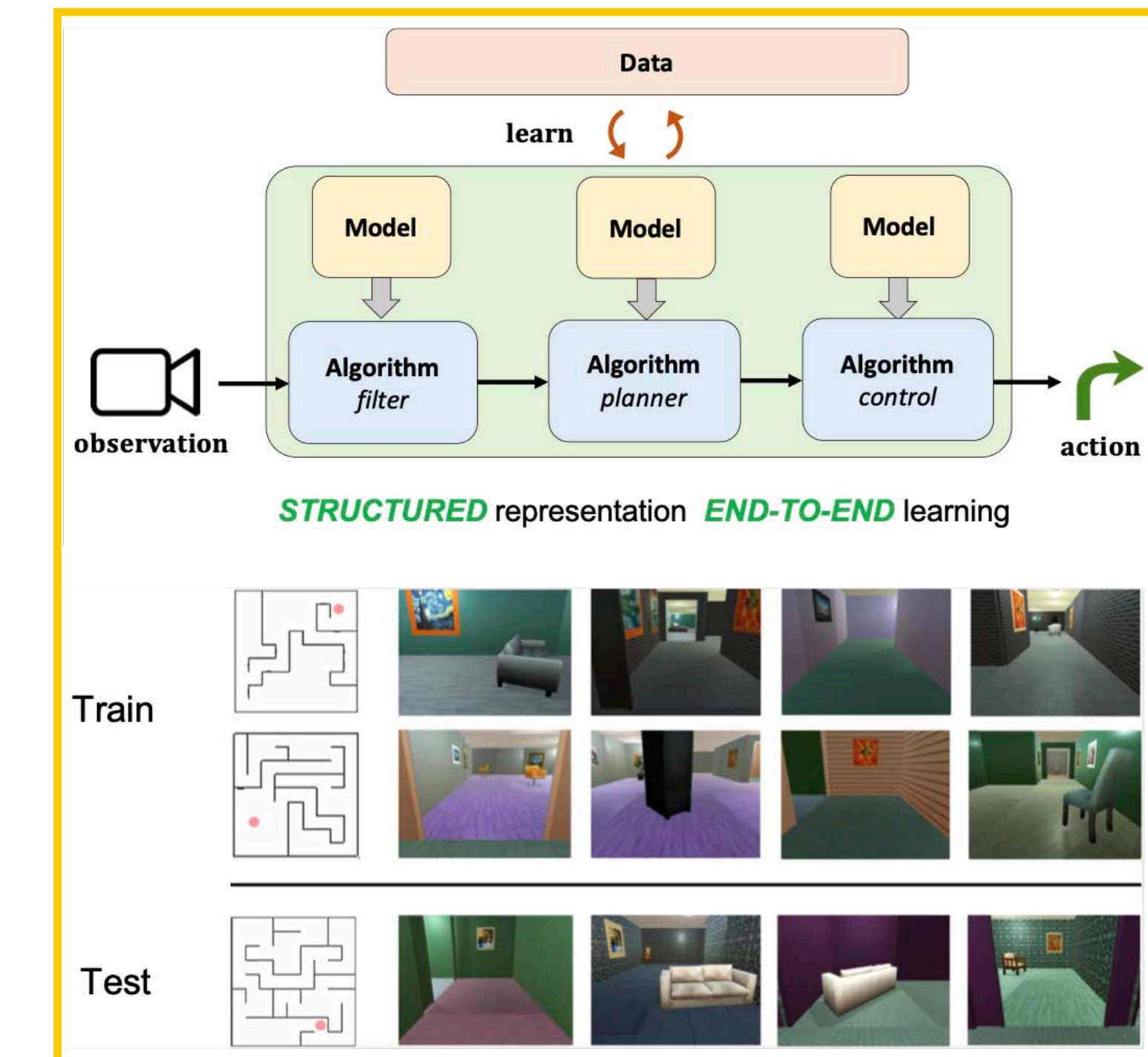
- Recurrent Networks
- Recursive Networks



Dynamic Graph Applications

Model structure
depends on the input:

- Recurrent Networks
- Recursive Networks
- Modular Networks



[1] Karkus et al., RSS 2019





Dynamic Graph Applications

Model structure
depends on the input:

- Recurrent Networks
- Recursive Networks
- Modular Networks
- (Your idea here!)

Final Project!





TensorFlow





TensorFlow: Versions

TensorFlow 1.0

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

TensorFlow 2.0

- Current release: 2.8.0
 - Released 2/2/2022
- 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



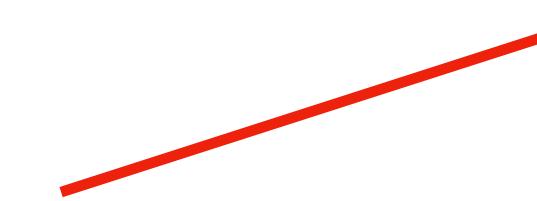
```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val,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

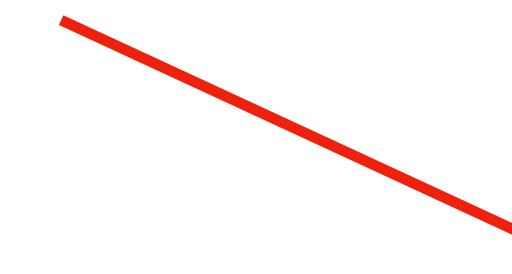
    w1.assign(w1 - learning_rate * grad_w1)
    w2.assign(w2 - learning_rate * grad_w2)
```





TensorFlow 2.0: Dynamic Graphs

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])

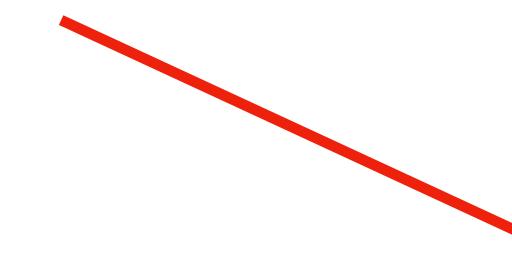
    w1.assign(w1 - learning_rate * grad_w1)
    w2.assign(w2 - learning_rate * grad_w2)
```





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)
```

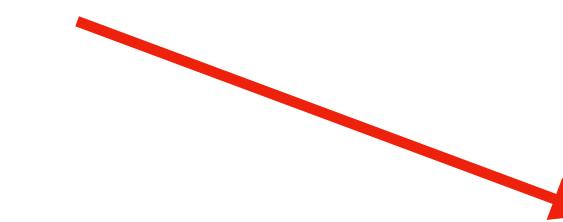
In PyTorch, all ops build graph by default; **opt out** via `torch.no_grad`
In Tensorflow, ops do not build graph by default; **opt in** via `GradientTape`





TensorFlow 2.0: Dynamic Graphs

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])

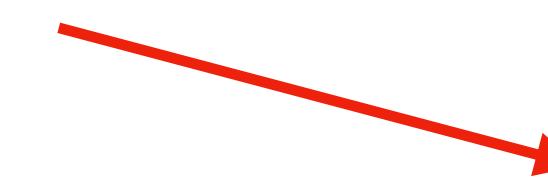
    w1.assign(w1 - learning_rate * grad_w1)
    w2.assign(w2 - learning_rate * grad_w2)
```





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])

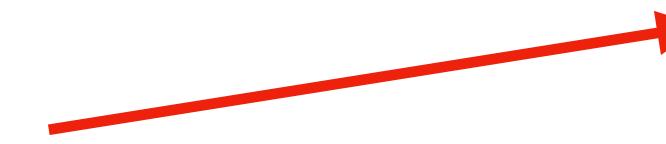
    w1.assign(w1 - learning_rate * grad_w1)
    w2.assign(w2 - learning_rate * grad_w2)
```



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: 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))
```





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))

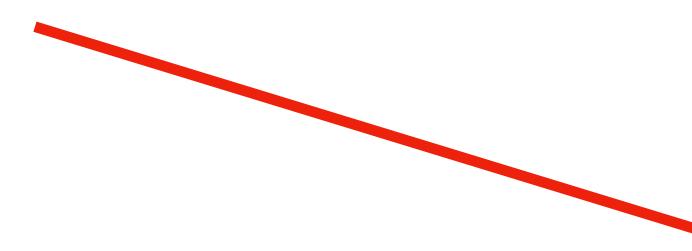
for t in range(1000):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        grads = tape.gradient(loss, params)
        opt.apply_gradients(zip(grads, params))
```



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)

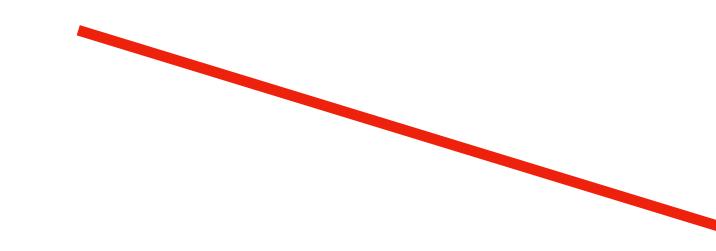
x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))

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

Optimizer object
updates parameters



```
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))

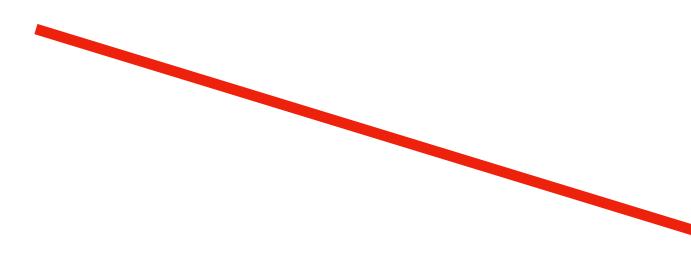
for t in range(1000):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = loss_fn(y_pred, y)
        grads = tape.gradient(loss, params)
    opt.apply_gradients(zip(grads, params))
```





Keras: High-level API

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

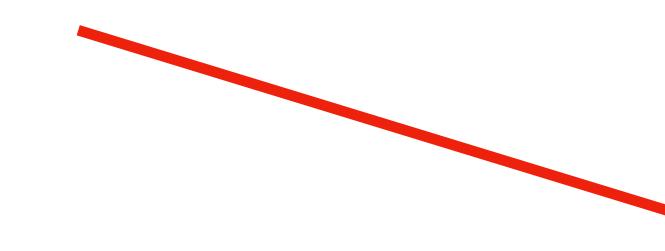
for t in range(1000):
    opt.minimize(step, params)
```





Keras: High-level API

Optimizer computes
gradients and
updates parameters



```
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))

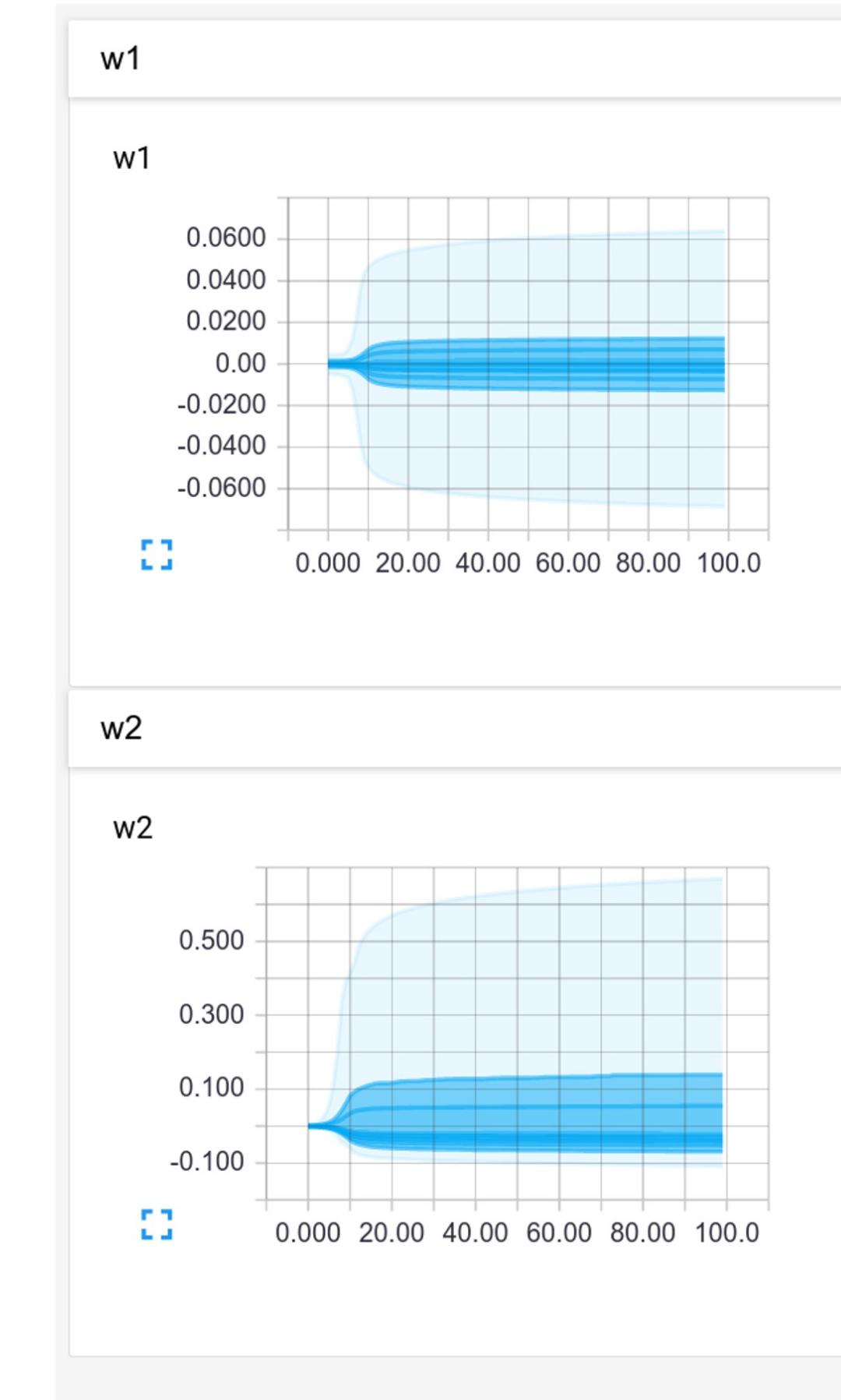
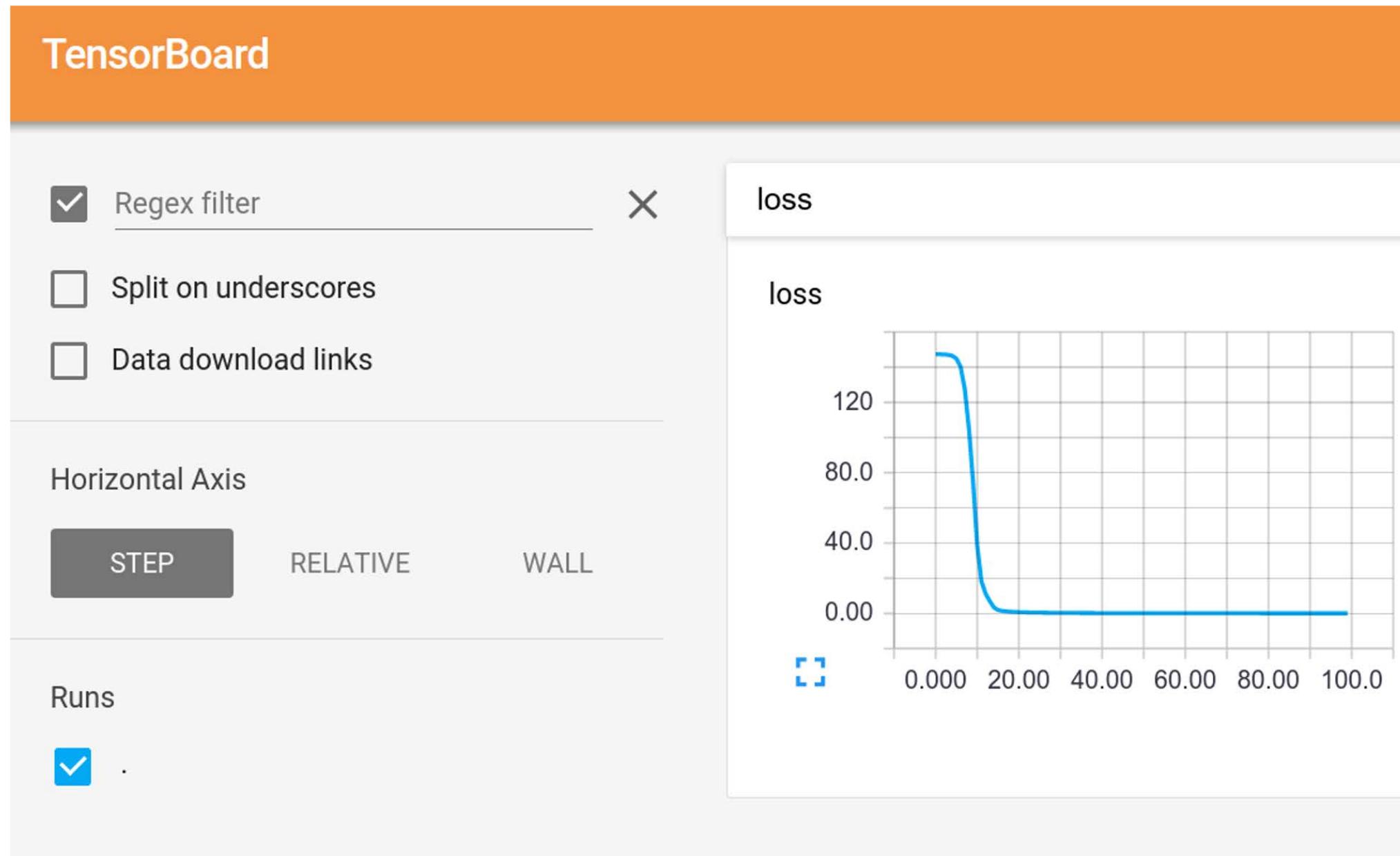
def step():
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    return loss

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



TensorBoard

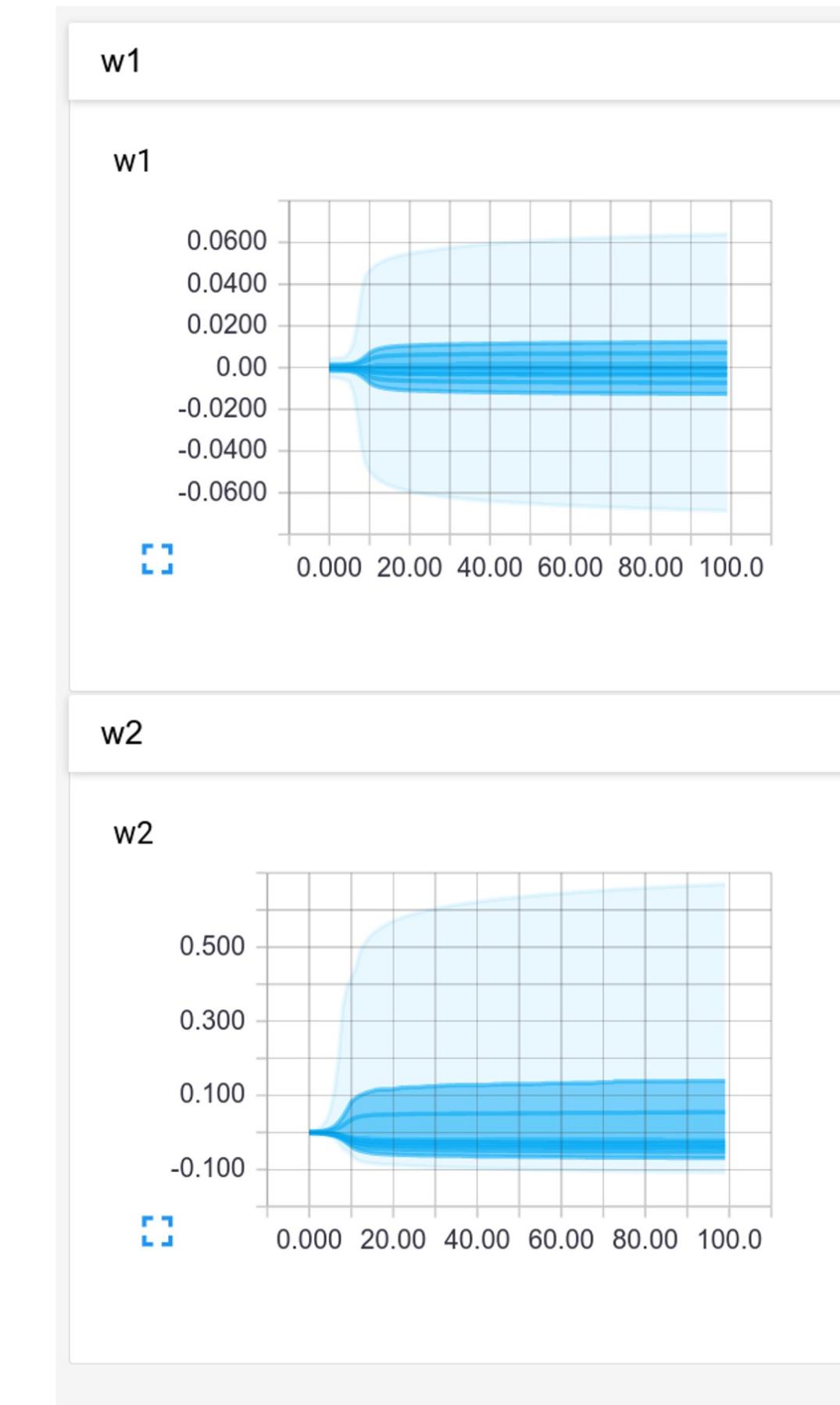
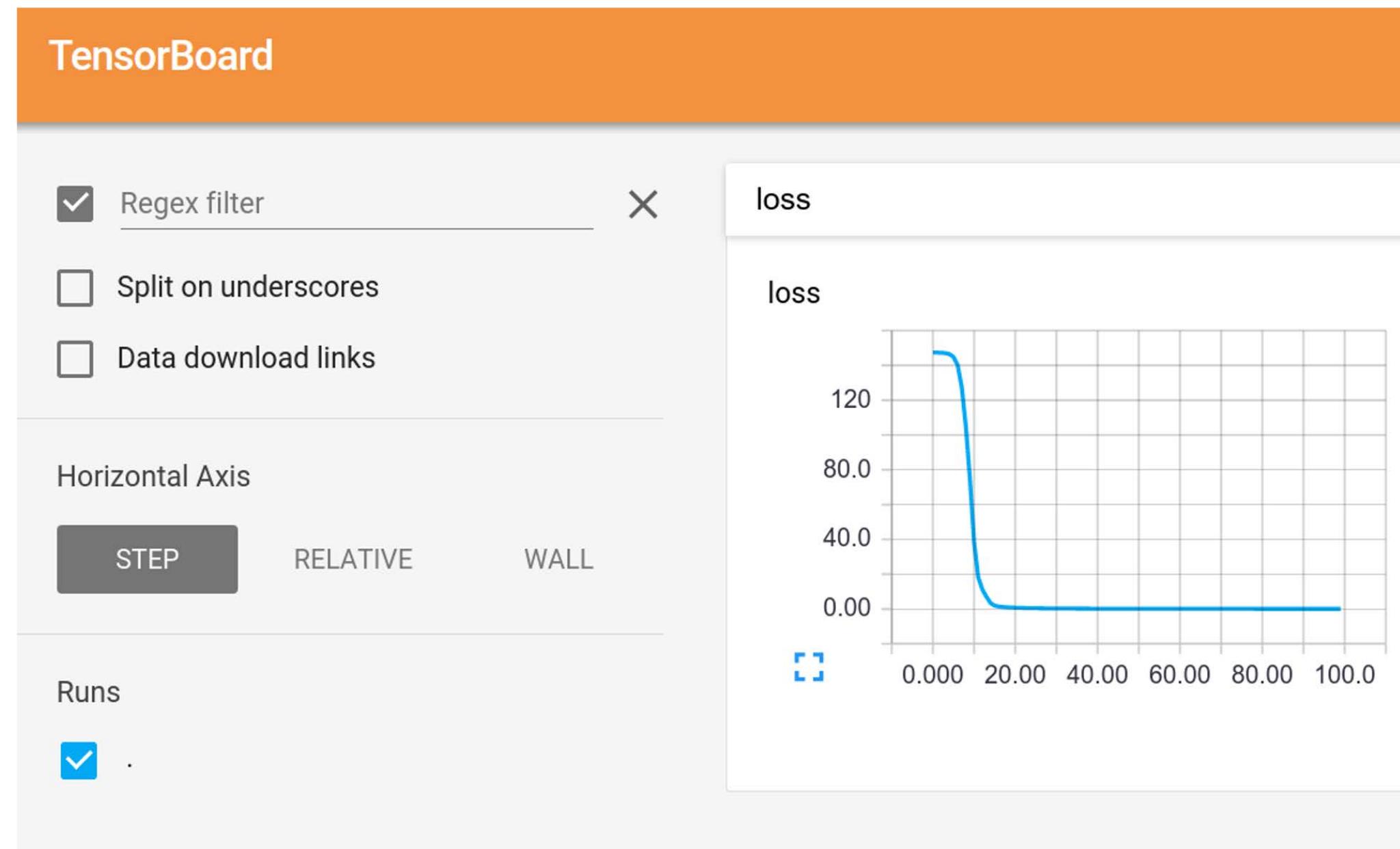
Add logging to code to record loss, stats, etc
Run server and get pretty graphs!





TensorBoard

Also works with [PyTorch!](#)



PyTorch vs TensorFlow

PyTorch

- My personal favorite
- 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



Summary: Deep Learning Software

Static Graphs vs Dynamic Graphs

PyTorch vs TensorFlow



DeepRob

Deep Learning Software
University of Minnesota

