











### A zoo of frameworks!





**CNTK** (Microsoft)

(Facebook)

**TensorFlow** 

PaddlePaddle (Google) (Baidu)

(Facebook)

Darknet (Redmon)

Chainer

JAX

MXNet (Amazon)

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







### A zoo of frameworks!



**CNTK** (Microsoft)

(Facebook)

**TensorFlow** 

(Facebook)

#### Darknet (Redmon)

Chainer

JAX

PaddlePaddle (Google) (Baidu)

#### MXNet (Amazon)

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

We'll focus on these





### Recall: Computational Graphs





**L**



### 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



6



# 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

People building crazy models







### Introduced to further optimize models (torch.compile) Intended to be backwards compatible with 1.x Expected stable release in March 2023





Video credit: [PyTorch](https://pytorch.org/get-started/pytorch-2.0/)



# 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

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

#### **Tensor**: Like a numpy array, but can run on GPU **P0, P1, P2**

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









#### Train a two-layer ReLU network andom data with I 2 k  $\cdots$ Running example: on random data with L2 loss



```
import torch
device = <b>torch.device('cpu')</b>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-6for 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] = 0grad_w1 = x.t() .mm(grad_h)w1 - = learning_rate * grad_wl
```
 $w2 = learning_rate * grad_w2$ 



#### Create random tensors for data and weights



```
import torch
```

```
device = <b>torch.device('cpu')</b>
```

```
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)
wl = <b>torch.random(D_in, H, device=device)</b>w2 = torch.randn(H, D out, device=device)
```

```
learning_rate = 1e-6for 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] = 0grad_w1 = x.t() .mm(grad_h)w1 - = learning rate * grad w1w2 = learning_rate * grad_w2
```


#### Running example: Train a there had need in the control of the contr dictions and los Forward pass: compute predictions and loss



```
import torch
device = <b>torch.device('cpu')</b>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-6for 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] = 0grad_w1 = x.t() .mm(grad_h)w1 - = learning rate * grad w1
```
 $w2 = learning_rate * grad_w2$ 



#### Backward pass: manually compute gradients



```
import torch
device = <b>torch.device('cpu')</b>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-6for 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())}
```
 $grad_h = grad_h_{relu,clone}$ 

 $w1 -$  = learning rate \* grad  $w1$ 

 $w2 = learning_rate * grad_w2$ 

 $grad_w1 = x.t() .mm(grad_h)$ 

 $grad_h[h < 0] = 0$ 



#### Gradient descent step on weights



```
import torch
device = <b>torch.device('cpu')</b>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-6for 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] = 0grad_w1 = x.t() .mm(grad_h)w1 - = learning rate * grad w1
```
 $w2 = learning$  rate \* grad w2



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



```
import torch
```

```
device = <b>torch.device('cpu')</b>
```

```
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)
wl = <b>torch.random(D_in, H, device=device)</b>w2 = torch.randn(H, D out, device=device)
```

```
learning_rate = 1e-6for 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] = 0grad_w1 = x.t() .mm(grad_h)w1 - = learning rate * grad w1w2 = learning_rate * grad_w2
```
#### Creating Tensors with Creating Tensors with requires\_grad=True enables autograd

PyTorch: Autograd



## PyTorch: Autograd

Operations on Tensors with requires\_grad=True cause PyTorch to build a computational graph



```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
wl = <i>torch.random(D_in, H, requires grad=True</i>)w2 = torch.randn(H, D out, requires grad=True)
learning rate = 1e-6for 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.gradw2 == learning rate * w2.getwl.grad. zero ()
        w2.getad{\cdot}zero ()
```
#### PyTorch: Autograd (of loss) with respect to data We will not want gradients



### PyTorch: Autograd

#### Operations on Tensors with radients with build a computation of the computation<br>The computation of the computat Do want gradients with respect to weights



```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
wl = <i>torch.random(D_in, H, requires grad=True</i>)w2 = torch.randn(H, D out, requires grad=True)
learning_rate = 1e-6for 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.gradw2 == learning rate * w2.getwl.grad. zero ()
        w2. grad. zero ()
```
#### requires\_grad=True cause PyTorch to Compute gradients with respect to all inputs that have requires\_grad=True!



#### import torch



loss.backward()

```
with torch.no\_grad():
    w1 - = learning rate * wl.grad
    w2 == learning_rate * w2.getwl.grad. zero ()
    w2.getad.zero ()
```




#### requires\_grad=True will add to the requires and the  $t$ s will also have requires Every operation on a tensor with 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 = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
wl = <i>torch.random(D in, H, requires grad=True</i>)w2 = torch.randn(H, D_out, requires_grad=True)
learning rate = 1e-6for 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.gradw2 == learning rate * w2.getwl.grad. zero ()
        w2.getad.zero ()
```
PyTorch: Autograd





#### requires\_grad=True will add to the requires and the  $t$ s will also have requires Every operation on a tensor with 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 = <i>torch</i>.<i>randn</i>(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-6for 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.gradw2 = learning rate * w2.grad
        wl.grad. zero ()
        w2.getad.zero ()
```
PyTorch: Autograd







#### import torch

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(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-6for 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 * wl.grad
        w2 = learning_rate * w2.gradwl.grad. zero ()
        w2.\text{grad. zero} ()
```








```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(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-6for 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 * wl.grad
        w2 = learning_rate * w2.gradwl.grad. zero ()
        w2.\text{grad. zero} ()
```






```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(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-6for 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 * wl.grad
        w2 = learning_rate * w2.gradwl.grad. zero ()
        w2.\text{grad. zero} ()
```






```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(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-6for 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 * wl.grad
        w2 = learning_rate * w2.gradwl.grad. zero ()
        w2.\text{grad. zero} ()
```




#### import torch N, D\_in, H, D\_out =  $64$ , 1000, 100, 10  $x = *torch*.*randn*(N, D_in)$  $y =$  torch.randn(N, D\_out)  $w1 = *torch.random(D_in, H, requires_grad=True*)$  $w2 =$  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 \* wl.grad  $w2 = learning_rate * w2.grad$  $wl.grad. zero$  ()  $w2.getad.zero$  ()





#### Perumulated into wit ared are **accumulated** into w1.grad and w2.grad and the graph is destroyed w2.grad and the graph is destroyedAfter backward finishes, gradients build a computational graph are accumulated into w1.grad and



#### import torch



 $w2.getad.zero$  ()





#### Perumulated into wit ared are **accumulated** into w1.grad and w2.grad and the graph is destroyed w2.grad and the graph is destroyed After backward finishes, gradients build a computational graph are accumulated into w1.grad and



### PyTorch: Autograd

Make gradient step on weights







#### Perumulated into wit ared are **accumulated** into w1.grad and w2.grad and the graph is destroyed w2.grad and the graph is destroyed After backward finishes, gradients build a computational graph are accumulated into w1.grad and



### PyTorch: Autograd

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









#### After backward finishes, gradients are **accumulated** into w1.grad and w2.grad and the graph is destroyed w2.grad and the graph is destroyed After backward finishes, gradients build a computational graph are accumulated into w1.grad and

### PyTorch: Autograd

#### import torch



Tell PyTorch not to build a graph for these operations



```
PyTorch: New functions
```

```
N, D in, H, D out = 64, 1000, 100, 10
    x = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
    y = torch.randn(N, D_out)
    w1 = torch.randn(D_in, H, requires_grad=True)
                                                      True)w2 = torch.randn(H, D_out, requires_grad=True) r = r\cdot ( )
    learning_rate = 1e-6for t in range(500):
      y pred = sigmoid(x.mm(w1)).mm(w2)
      loss = (y_{pred} - y) . pow(2) .sum()loss.backward()
      if t \frac{1}{2} 50 == 0:
        print(t, loss.time() )with torch.no grad():
        w1 -= learning rate * wl.grad
        w2 == learning rate * w2.getw1.grad.zero ()
        w2. grad. zero ()
```


## PyTorch: New Functions

efine new operations Python functions Can define new operations using Python functions

PyTorch: New functions

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



```
PyTorch: New functions
```

```
N, D in, H, D out = 64, 1000, 100, 10
    x = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
    y = torch.randn(N, D_out)
    w1 = torch.randn(D_in, H, requires_grad=True)
                                                         True)w2 = <i>torch</i>.<br>randn(H, D.out, requires <i>grad</i>=True)=True)
\cdot ( )
    learning_rate = 1e-6for t in range(500):
      y pred = sigmoid(x.mm(w1)).mm(w2)
      loss = (y_{pred} - y) . pow(2) .sum()loss.backward()
       if t \frac{1}{2} 50 == 0:
         print(t, loss.time() )with torch.no grad():
         w1 -= learning rate * wl.grad
         w2 == learning_rate * w2w1.grad.zero ()
         w2.\text{grad}.\text{zero} ()
```
efine new operations Python functions Can define new operations using Python functions

PyTorch: New functions

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







## PyTorch: New Functions

PyTorch: New functions

When our function runs, it will add to the graph The and to the graph  $\overline{\phantom{a}}$ 

Can define new operations using Python functions

 $\mathbf{u}$  is the python functions of  $\mathbf{u}$ 

Gradients computed with autograd it will add to the graph n autogradients computed to the computed of th

 $\sqrt{10}$ 

 $+1$   $\rightarrow$   $1.0/$ 

x

 $* -1$ 

 $\overline{a}$ 

 $\rightarrow$ 

exp

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

```
import torch
N, D_by subclassing Function, define 
X = \begin{cases} X = \frac{1}{1} \\ Y = \frac{1}{1} \end{cases} forward and backward
       class Sigmoid(torch.autograd.Function):
w1 =@staticmethod
w2 =def forward(ctx, x):
           y = 1.0 / (1.0 + (-x).exp())learn
           ctx.save for backward(y)
for t
           return y
  Y_{I}\log@staticmethod
  \logdef backward(ctx, grad y):
  if
           y_i = ctx.\text{saved} tensors
            grad_x = grad_y * y * (1.0 - y)return grad x
  wit
       def sigmoid(x):
         return Sigmoid.apply(x)
     Recall:
```


# PyTorch: New Functions

 $\mathbf{C} \mathbf{C}$ 



```
PyTorch: New functions
  import torch<br>Define new autograd operators
   Pefine new autograd operators<br>Per define new subclassing Eunction define
                                                  by subclassing Function, define 
                                                  by subclassing Function, define 
                                                  forward and backward
     ng Python functions \begin{array}{ccc} x & = & 1 \\ y & = & 1 \end{array} forward and backward
                                               class Sigmoid(torch.autograd.Function):
                                         w1 =@staticmethod
                                         w2 =def forward(ctx, x):
                                                   y = 1.0 / (1.0 + (-x).exp())learn
                                                   ctx.save for backward(y)
                                         for t
                                                   return y
                                           Y_{I}\log@staticmethod
                                                 def backward(ctx, grad_y):
                                           \logy_i = ctx.\text{saved} tensors
                                           if
                                                   grad_x = grad_y * y * (1.0 - y)return grad x
                                           wit
                                               def sigmoid(x):
                                                 return Sigmoid.apply(x)
```
Can define new operations using Python functions ng Python functions

34





 $\left|\sigma(x)\right| = \frac{e}{1-\phi(x)}$  it adds one node to the graph  $\left| \begin{array}{c} x \\ \end{array} \right|$  Sigmoid Now when our function runs, it adds one node to the graph!



# PyTorch: New Functions

\* -1

exp

 $\rightarrow$ 

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

> it will add to the graph When our function runs, The and to the graph  $\overline{\phantom{a}}$

with autograd Gradients computed it will add to the graph n autogradients computed to the computed of th

 $\frac{1}{\sigma(x)}\Big|_{-\frac{1}{\sigma(x)}}$ 

with a strong control and the strong c

 $\sqrt{10}$ 

 $+1$   $\rightarrow$   $1.0/$ 

```
PyTorch: New functions
  import torch<br>Define new autograd operators
   Pefine new autograd operators<br>Per define new subclassing Eunction define
                                                            by subclassing Function, define 
                                                            by subclassing Function, define 
      \frac{x}{x} = \frac{1}{x} forward and backward
       ng Python functions \begin{array}{ccc} x & = & 1 \\ y & = & 1 \end{array} forward and backward
                                                         class Sigmoid(torch.autograd.Function):
                                                 w1 =@staticmethod
                                                 w2 =def forward(ctx, x):
                                                             y = 1.0 / (1.0 + (-x).exp())learn
                                                             ctx.save for backward(y)
                                                 for t
                                                             return y
                                                   Y_I\log@staticmethod
                                                   \logdef backward(ctx, grad y):
                                                   if
                                                             y_i = ctx.\text{saved} tensors
                                                              grad_x = grad_y * y * (1.0 - y)return grad x
                                                   wit
                                                         def sigmoid(x):
                                                           return Sigmoid.apply(x)
                                                        Tוו\Psiועך \partialווז\Gammaן אין ק\Gammaוני וועטע
        -\sigma(x)\big| = \frac{1}{(1 - x)^2} = \left(\frac{1}{1 - x^2}\right)^{1/2}cases Plythan functions are good enough
J_{\text{out}} Justin J
                                                     \overline{a}
```


using Python functions and the python functions of the Can define new operations using Python functions

 $\mathbf{u}$  is the python functions of  $\mathbf{u}$ 

 $\overline{\phantom{a}}$ 

 $\overline{a}$ 

x

 $* -1$ 

exp



# PyTorch: New Functions





#### . . .<br>Heilral nets with the contract of the contra<br>  $\sim$  0.000 m since the contract of the contra Higher-level wrapper for working with neural nets

IT ITIANE YOUR Use this! It will make your life easier



### PyTorch: nn

```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
model = <b>torch.nn.S</b> equivalenti\text{torch.nn.Linear}(D_in, H),torch.nn. ReLU(),
           torch.nn.Linear(H, D.out))learning_rate = 1e-2for t in range(500):
    y pred = model(x)loss = <i>torch.nn.functional.mse_loss(y_pred, y)</i>loss.backward()
    with torch.no_grad():
        for param in model.parameters():
             param -= learning rate * param.grad
    model.zero grad()
```
PyTorch: nn
#### objective API: Define API: D uence <u>\_\_\_\_\_\_\_\_\_\_</u> of layers objects, each of of layers objects, each of which holds weight tensors which holds weight tensors Object-oriented API: Define model object as sequence



#### import torch

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

```
model = <b>torch.nn.S</b> equivalenti\text{torch.nn.Linear}(D_in, H),torch.nn. ReLU(),
           torch.nn.Linear(H, D out))
```

```
learning_rate = 1e-2for t in range(500):
   y pred = model(x)loss = <i>torch.nn.functional.mse_loss(y_pred, y)</i>loss.backward()
   with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero grad()
```


# PyTorch: nn



# PyTorch: nn

Forward pass: Feed data to Forward pass: Feed data to model and compute loss model and compute loss

PyTorch: nn



```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
model = <b>torch.nn.S</b> equivalentitorch.nn.Linear(D_in, H),torch.nn. ReLU(),
           torch.nn.Linear(H, D.out))learning_rate = 1e-2for t in range(500):
    y pred = model(x)loss = <i>torch.nn.functional.mse_loss(y_pred, y)</i>loss.backward()
    with torch.no_grad():
        for param in model.parameters():
            param -= learning rate * param.grad
    model.zero grad()
```
Forward pass: Feed data to model and compute loss model and compute loss



# PyTorch: nn

PyTorch: nn

helners like loss functions helpers like loss functions torch.nn.functional has useful helpers like loss functions



```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
model = <b>torch.nn.S</b> equivalentitorch.nn.Linear(D_in, H),torch.nn. ReLU(),
          torch.nn.Linear(H, D.out))learning_rate = 1e-2for t in range(500):
    y pred = model(x)loss = <b>torch.nn.functional.mse_loss(y pred, y)</b>loss.backward()
    with torch.no_grad():
        for param in model.parameters():
             param -= learning_rate * param.grad
    model.zero_grad()
```
#### Speed to an model weights (they have  $\text{tr}\left(\mathbf{r}\right)$ ruc<sub>)</sub> Backward pass: compute gradient with respect to all requires\_grad=True)



```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D out)
model = <b>torch.nn.S</b> equivalentitorch.nn.Linear(D_in, H),torch.nn. ReLU(),
           torch.nn.Linear(H, D_out))
learning_rate = 1e-2for t in range(500):
    y pred = model(x)loss = <i>torch.nn.functional.mse_loss(y_pred, y)</i>
```


loss.backward()

# PyTorch: nn

```
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 each model parameter  $\log$ Make gradient step on (with gradients disabled)



```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
model = <b>torch.nn.S</b> equivalentitorch.nn.Linear(D_in, H),torch.nn. ReLU(),
          torch.nn.Linear(H, D.out))learning_rate = 1e-2for t in range(500):
    y pred = model(x)loss = <i>torch.nn.functional.mse_loss(y_pred, y)</i>loss.backward()
    with torch.no_grad():
        for param in model.parameters():
             param -= learning_rate * param.grad
    model.zero_grad()
```
#### Use an **optimizer** for Use an **optimizer** for different update rules different update rules



```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D out)
model = <b>torch.nn.S</b> equivalentitorch.nn.Linear(D in, H),
           torch.nn.ReLU(),
           torch.nn.Linear(H, D out))
learning rate = 1e-4
optimize r = torch.optim. Adam(model.parameters(),
                               lr=learning_rate)
for t in range(500):
    y pred = model(x)loss = <i>torch.nn.functional.mse_loss(y_pred, y)</i>loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```


#### PyTorch: optim PyTorch: optim

#### and computed computed to the computation of the computation of the computation of the contribution of update and zero gradients After computing gradients, use optimizer to



```
import torch
N, D in, H, D out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D out)
model = <b>torch.nn.S</b> equation <math display="inline">1</math>torch.nn.Linear(D_in, H),torch.nn.ReLU(),
           torch.nn.Linear(H, D out))
learning_rate = 1e-4optimize r = torch.optim. Adam(model.parameters(),
                                 lr=learning rate)
for t in range(500):
    y pred = model(x)loss = <i>torch.nn.functional.mse_loss(y_pred, y)</i>loss.backward()
```
optimizer.step() optimizer.zero\_grad()



# PyTorch: optim



### PyTorch: nn Defining Modules <u>Defining Modules Modul</u>

#### A PyTorch **Module** is a neural net layer; it inputs and outputs Tensors

on to define idyers ds custom iviouules 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 ()
        selfuinearl = torch.nn.Linear(D_in, H)
        selfu. Thear2 = torch.nn. Linear(H, D out)
    def forward(self, x):
        h_{relu} = selfu.linearl(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 = <i>torch.nn.functional.mse_loss</i>(<math>y</math> pred, <math>y</math>)loss.backward()
    optimizer.step()
    optimizer.zero grad()
```
A PyTorch **Module** is a neural net an contain weights or Modules can contain weights or other modules

#### A PyTorch **Module** is a neural net Define our whole model as a single Module



### PyTorch: nn Defining Modules

PyTorch: nn



import torch

```
class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D.out):super(TwoLayerNet, self). __init__()
        selfuinearl = torch.nn.Linear(D_in, H)
        selfu. The 2 = torch.nn. Linear (H, D out)
    def forward(self, x):
        h_{relu} = selfu.linearl(x).clamp(min=0)
        y pred = self.linear2(h relu)
        return y_pred
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(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 = <i>torch.nn.functional.mse loss</i>(<i>y pred</i>, <i>y</i>)loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```
PyTorch: nn



```
import torch
class TwoLayerNet(torch.nn.Module):
    def __init_(self, D_in, H, D_out):
        super(TwoLayerNet, self). init ()
        selfuinearl = torch.nn.Linear(D_in, H)
        selfu. Thear2 = torch.nn. Linear(H, D out)
    def forward(self, x):
        h_{relu} = selfu.linearl(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 = <i>torch.nn.functional.mse_loss(y pred, y)</i>loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```


### PyTorch: nn Defining Modules

#### Initializer sets up two children (Modules can contain modules)

No need to define backward autograd will handle it define backward -



```
import torch
class TwoLayerNet(torch.nn.Module):
    def _init_(self, D_in, H, D_out):
        super(TwoLayerNet, self). _init ()
        selfuinearl = torch.nn.Linear(D_in, H)
        selfu. Thear2 = torch.nn. Linear(H, D out)
    def forward(self, x):
        h_{relu} = selfu.linearl(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 = <code>torch.nn.functional.mse loss(y pred, y)</code>loss.backward()
    optimizer.step()
    optimizer.zero grad()
```
#### Define forward pass using child modules and tensor operations a tensor operations layer; it inputs and outputs Tensors



### PyTorch: nn Defining Modules

#### $\mathbf{S}$  sequential containers  $\mathbf{S}$ 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 ()
        selfu. Thearl = torch.nn. Linear (D in, D out)
        selfu. Thear2 = torch.nn. Linear(D in, D out)
    def forward(self, x):
        h1 = selfu1inear1(x)h2 = selfuinear2(x)
        return (h1 * h2).clamp(min=0)N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
model = <b>torch.nn.S</b> equivalent(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 = <code>torch.nn.functional.mse_loss(y_pred, y)</code>loss.backward()
    optimizer.step()
    optimizer.zero grad()
```


### PyTorch: nn Defining Modules

#### Define network component Denne network component



### Defining Modules Defining ModulesPyTorch: nn





```
import torch
class ParallelBlock(torch.nn.Module):
    def init (self, D in, D out):
         super(ParallelBlock, self). _init ()
         selfu. Thearl = torch.nn. Linear (D in, D out)
         selfuinear2 = torch.nn.Linear(D in, D out)
    def forward(self, x):
         h1 = selfu1inear1(x)h2 = selfuinear2(x)
         return (h1 * h2).clamp(min=0)
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
model = <b>torch.nn.S</b> equivalent(ParallelBlock(D_in, H),
             ParallelBlock(H, H),
             torch.nn.Linear(H, D out))
optimizer = <i>torch</i>.<i>optim</i>.<i>Adam</i>(model.<i>parameters</i>(), 1r=1e-4)for t in range(500):
    y pred = model(x)loss = <code>torch.nn.functional.mse_loss(y_pred, y)</code>loss.backward()
    optimizer.step()<br>optimizer.zero grad()
```
#### Stack multiple instances of the component in a sequential Itipie instances<br>Intin a conventi  $\begin{array}{cc} \text{Equation} \ & \text{Equation$

Very easy to quickly build complex network architectures!



### Defining Modules Defining ModulesPyTorch: nn





```
import torch
class ParallelBlock(torch.nn.Module):
    def init (self, D in, D out):
        super(ParallelBlock, self). init ()
        selfu. Thearl = torch.nn. Linear (D in, D out)
        selfu. Thear2 = torch.nn. Linear(D in, D out)
    def forward(self, x):
        h1 = selfu1inear1(x)h2 = selfu = n2(x)return (h1 * h2).clamp(min=0)
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
model = <b>torch.nn.S</b> equivalent(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 = <code>torch.nn.functional.mse_loss(y_pred, y)</code>loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```


# PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorc<br>Loaders PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchist<br>L

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



## PyTorch: DataLoaders

```
from torch.utils.data import TensorDataset, DataLoader
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D out)
loader = DataLoader(TensorDatabase(x, y), batch_size=8)model = TwoLayerNet(D_in, H, D.out)optimize r = 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 = <i>torch.nn.functional.mse_loss(y pred, y_batch)</i>loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```
## PyTorch: DataLoaders

```
from torch.utils.data import TensorDataset, DataLoader
N, D in, H, D out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
loader = DataLoader(TensorDatabase(x, y), batch_size=8)model = TwoLayerNet(D_in, H, D.out)optimize r = torch.optim.SGD (model. parameters(), 1r=1e-2)for epoch in range(20):
    for x_batch, y_batch in loader:
        y pred = model(x batch)
        loss = <i>torch.nn.functional.mse_loss(y pred, y_batch)</i>loss.backward()
        optimizer.step()
```

```
optimizer.zero_grad()
```
# PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorc<br>Loaders PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchistan PyTorchist<br>L







Iterate over loader to

form minibatches



# PyTorch: Pretrained Models

## <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= $True$ )



Super easy to use pertained models with torch vision



### PyTorch: Dynamic Computation Graphs PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
wl = <i>torch.random(D_in, H, requires grad=True</i>)w2 = <i>torch.random(H, D.out, requires grad=True</i>)learning_rate = 1e-6for 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 PyTorch: Dynamic Computation Graphs

#### x | w1 | w2 | y



#### Create Tensor objects

import torch

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
wl = <i>torch.random(D_in, H, requires<sub>grad</sub>=True</i>)w2 = <i>torch.random(H, D.out, requires grad=True</i>)
```

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





```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = <i>torch</i>.<i>randn</i>(N, D_out)w1 = torch.randn(D_in, H, requires_grad=True)
w2 = <i>torch.random(H, D.out, requires_grad=True</i>)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







```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = <i>torch</i>.<i>randn</i>(N, D_out)w1 = <i>torch.random(D_in, H, requires_grad=True</i>)w2 = <i>torch.random(H, D.out, requires_grad=True</i>)learning_rate = 1e-6for t in range(500):
    y pred = x . mm(w1) . claimp(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 = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
wl = <i>torch.random(D_in, H, requires<sub>grad</sub>=true</i>)w2 = <i>torch.random(H, D.out, requires_grad=True</i>)learning_rate = 1e-6for 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 PyTorch: Dynamic Computation Graphs

#### $x \mid w1 \mid w2 \mid y$



Perform backprop, throw away graph

```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
wl = <i>torch.random(D_in, H, requires grad=True</i>)w2 = <i>torch.random(H, D.out, requires grad=True</i>)learning_rate = 1e-6for t in range(500):
    y_{pred} = x . mm(w1) . clamp(min=0) . mm(w2)loss = (y pred - y).pow(2).sum()
```
loss.backward()





```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = <i>torch</i>.<i>randn</i>(N, D_out)w1 = torch.randn(D_in, H, requires_grad=True)
w2 = <i>torch.random(H, D.out, requires_grad=True</i>)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







```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = <i>torch</i>.<i>randn</i>(N, D_out)w1 = <i>torch.random(D_in, H, requires_grad=True</i>)w2 = <i>torch.random(H, D.out, requires_grad=True</i>)learning_rate = 1e-6for t in range(500):
    y pred = x . mm(w1) . claimp(min=0) . mm(w2)loss = (y_{pred} - y) . pow(2) .sum()
```

```
loss.backward()
```
Build graph data structure AND perform computation







```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D_out)
w1 = <i>torch.random(D_in, H, requires_grad=True</i>)w2 = <i>torch.random(H, D.out, requires_grad=True</i>)learning_rate = 1e-6for t in range(500):
    y_{pred} = x . mm(w1) . clamp(min=0) . mm(w2)loss = (y pred - y).pow(2).sum()
```
loss.backward()

Perform backprop, throw away graph



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





#### import torch

### PyTorch: Dynamic Computation Graphs PyTorch: Dynamic Computation Graphs

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2a = torch.random(H, Dout, requires grad=True)w2b = <i>torch</i>.randn(H, D out, requires grad=True)learning rate = 1e-6prev loss = 5.0for t in range(500):
  w2 = w2a if prev_loss < 5.0 else w2b
  y pred = x . mm(w1) . clamp(min=0) . mm(w2)loss = (y pred - y).pow(2).sum()loss.backward()
  prev loss = loss.item()
```
Dynamic graphs let you use regular Python control flow during the forward pass!



### PyTorch: Dynamic Computation Graphs PyTorch: Dynamic Computation Graphs

Initialize two different weight matrices for second layer



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2a = torch.random(H, D out, requires grad=True)w2b = <i>torch.random(H, D out, requires grad=True</i>)
```

```
learning rate = 1e-6prev loss = 5.0for t in range(500):
 w2 = w2a if prev_loss < 5.0 else w2b
 y pred = x . mm(w1) . clamp(min=0) . mm(w2)loss = (y pred - y).pow(2).sum()loss.backward()
```
 $prev loss = loss.item()$ 

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



### PyTorch: Dynamic Computation Graphs PyTorch: Dynamic Computation Graphs

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

(this model doesn't makes sense! Just a simple dynamic example)



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = <i>torch</i>.<i>randn</i>(N, D_in)y = torch.randn(N, D out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2a = torch.random(H, D out, requires grad=True)w2b = <i>torch</i>.randn(H, D out, requires grad=True)learning rate = 1e-6prev loss = 5.0for 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** graphs



### Alternative: Static Computation Graphs (Static Computation Graphs) Alternative: Static Computation 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)



#### PyTorchistan Graphs with Julian Static Graphs with Julian Station Stat Alternative: Static Graphs with JIT

Define model as a Python function



#### import torch



![](_page_67_Picture_0.jpeg)

#### PyTorchistan Graphs with Julian Static Graphs with Julian Station Stat 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!

![](_page_67_Picture_4.jpeg)

#### 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 = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
```

```
w1 = torch.randn(D_in, H, requires grad=True)
w2a = torch.random(H, Dout, requires grad=True)w2b = <i>torch.random(H, D out, requires grad=True</i>)
```
 $graph = **torch.** *jit*. **script (model)**$ 

```
prev loss = 5.0learning rate = 1e-6for t in range(500):
 loss = graph(x, y, w1, w2a, w2b, prev loss)loss.backward()
 prev loss = loss.item()
```
![](_page_68_Picture_0.jpeg)

#### PyTorchistan Graphs with Julian Static Graphs with Julian Station Stat Alternative: Static Graphs with JIT

![](_page_68_Figure_2.jpeg)

![](_page_68_Picture_3.jpeg)

#### 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 = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
w1 = torch.randn(D_in, H, requires grad=True)
w2a = torch.random(H, D out, requires grad=True)w2b = <i>torch.random(H, D out, requires grad=True</i>)
```
 $graph = **torch.** *jit*. **script (model)**$ 

```
prev loss = 5.0learning rate = 1e-6for t in range(500):
 loss = graph(x, y, w1, w2a, w2b, prev loss)loss.backward()
 prev_loss = loss.item()
```
![](_page_69_Picture_0.jpeg)

#### PyTorchists of Graphs with Jit<br>Production and Jittle Graphs with Jittle Graphs with Jittle Graphs with Jittle Graphs with Jittle Graphs with<br>Production and Jittle Graphs with Jittle Graphs with Jittle Graphs with Jittle Gr Alternative: Static Graphs with JIT

Use our compiled graph object at each forward pass

![](_page_69_Picture_3.jpeg)

#### 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 = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2a = torch.random(H, D.out, requires grad=True)w2b = <i>torch</i>.randn(H, Dout, requires <i>grad=True</i>)graph = <b>torch.</b> <i>jit</i>. <b>script (model)</b>prev loss = 5.0learning rate = 1e-6for t in range(500):
 \begin{bmatrix} \text{loss} = \text{graph}(x, y, w1, w2a, w2b, prev loss) \end{bmatrix}loss.backward()
```

```
prev loss = loss.item()
```
![](_page_70_Picture_0.jpeg)

#### PyTorchists of Graphs with Jit<br>Production and Jittle Graphs with Jittle Graphs with Jittle Graphs with Jittle Graphs with Jittle Graphs with<br>Production and Jittle Graphs with Jittle Graphs with Jittle Graphs with Jittle Gr 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

![](_page_70_Picture_4.jpeg)

#### 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 = <i>torch</i>.<i>randn</i>(N, D in)y = torch.randn(N, D out)
w1 = torch.randn(D_in, H, requires grad=True)
w2a = torch.random(H, D out, requires grad=True)w2b = <i>torch.random(H, D out, requires grad=True</i>)prev loss = 5.0learning rate = 1e-6
for t \times n range(500):
  loss = model(x, y, w1, w2a, w2b, prev loss)loss.backward()
  prev_loss = loss.titlem()
```
With static graphs, framework can **optimize** the graph for you before it runs!

Conv ReLU Conv ReLU Conv

![](_page_71_Picture_0.jpeg)

### Static vs Dynamic Graphs: Optimization Static vs Dynamic Graphs: Optimization

![](_page_71_Figure_4.jpeg)

![](_page_71_Picture_5.jpeg)

The graph you wrote

Conv+ReLU

Equivalent graph with **fused operations**

Conv+ReLU

Conv+ReLU
Once graph is built, can **serialize** it and run it without the code that built the graph!



### $S_n$ Static vs Dynamic Graphs: Optimization

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



Graph building and execution are intertwined, so always need to keep code around

## **Static Dynamic**

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





### $S_n$ Static vs Dynamic Graphs: Optimization

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

## **Static Dynamic**









### Dynamic Graph Applications End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-<br>End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-to-End-<br>End-to-End-to-End-to-End-to-En Dynamic Graph Applications

#### *Abstract*—We present differentiable particle filters (DPFs): a differentiable implementation of the particle filter algorithm depends on the input: Model structure

- end-to-end differentiable, we can efficiently train their models by optimizing end-to-end-to-end-to-end-to-end-to-end-to-end-to-end-to-endthat provide the property such as model and the such as model and the three three three theories of the theories of the theories of the three th - Recurrent Networks
- structure of  $\mathbb{R}^n$  recovered to  $\mathbb{R}^n$  and means  $\mathbb{R}^n$  and means  $\mathbb{R}^n$ surement update that on a probability of the probab states. This structure represents an algorithmic prior that im-- Recursive Networks



End (2) Xiao Ma, Peter Karkus, David Hsu, Wee Sun Lee. "Particle Filter Recurrent Neural Networks" AAAI, 2020. [1] Rico Jonschkowski, Divyam Rastogi, Oliver Brock. "Differentiable Particle Filters: End-to-End Learning with Algorithmic Priors" RSS, 2018 with models of the specific task. By making the specific task of the specific tasks and the specific these algorithms. <sub>the two bonson tows in Divyann Rastogi, Onver Brock. Binerentiable Particle Pilters. End-to-End Eearning with Algontmine Phors Troo, 2010<br>[2] Xiao Ma, Peter Karkus, David Hsu, Wee Sun Lee. "Particle Filter Recurrent Neura</sub> algorithms into network architectures. This approach enables are the state of the state of the state of the st<br>This approach enables are the state of the st

### Model structure depends on the input:

- Recurrent Networks
- Recursive Networks
- Modular Networks



11 Feter Kankus, Xiao wa, David Hisu, Leslie Fack Kaelbillig, wee Suit Lee, Tomas Lozano-Ferez.<br>"Differentiable Algorithm Networks for Composable Robot Learning" RSS, 2019 [1] Peter Karkus, Xiao Ma, David Hsu, Leslie Pack Kaelbling, Wee Sun Lee, Tomas Lozano-Perez.



[1] Karkus et al., RSS 2019



### Dynamic Graph Applications Dynamic Graph Applications

### Model structure depends on the input:

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





### Dynamic Graph Applications Dynamic Graph Applications

### **Final Project!**

78



# TensorFlow





## TensorFlow: Versions 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 <u>Tensor 1.0:00 in the set of the s</u><br>Tensor 1.0:00 in the set of the



tf

## import numpy as np import tensorflow as tf

top of each snippet) (Assume imports at the





```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))y = tf.placeholder(tf.float32, shape=(N, D))wl = 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 - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, axis=1))grad_w1, grad_w2 = tf.gradients(loss, [wl, w2])with tf. Session() as sess:
    values = \{x: np.random.random(N, D),\}wl: np.random.random(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 Tensor 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1<br>Tensor 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1<br>





Then run the granh many Then **run** the graph many times



```
N, D, H = 64, 1000, 100x = 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 - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, axis=1))grad_w1, grad_w2 = tf.gradients(loss, [wl, w2])
```

```
with tf. Session() as sess:
    values = \{x: np.random.random(N, D),\}wl: np.random.random(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_val, grad_w2_val = out
```




### TensorFlow 2.0: TensorFlow 2.0: TensorFlow 2.0: Dynamic Graphs









```
N, Di N, Din, H, Dout = 16, 1000, 100, 10
x = t x = tf.random.normal((N, Din))y = t y = tf.random.normal((N, Dout))
W^{1} = tf.Variable(tf.random.normal((Din, H)))
w2 = w2 = tf.Varable(tf.random.normal((H, Dout)))
```

```
learning rate = 1e-6for t 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 - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, axis=1))grad w1, grad w2 = \text{tape.gradient}(\text{loss}, [w1, w2])wl. assign(w1 - learning_rate * grad_w1)w2 \text{.} assign(w2 - learning rate * grad w2)
```




### impor TensorFlow 2.0: Dynamic Graphs TensorFlow 2.0: Dynamic Graphs





```
y = tf.random.normal((N, Dout))wl = tf.Variable(tf.random.normal((Din, H)))w2 = tf.Variable(tf.random.normal((H, Dout)))learning rate = 1e-6for 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 - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, axis=1))grad w1, grad w2 = tape.gradient(loss, [w1, w2])
  wl. assign(w1 - learning_rate * grad_w1)w2 \text{.} assign(w2 - learning rate * grad w2)
          wl \cdot assign(w1 - learning rate * grad w1)w2 \text{.}assign(w2 - learning rate * grad w2)
```


### impor TensorFlow 2.0: Dynamic Graphs TensorFlow 2.0: Dynamic Graphs





In Tensorflow, ops do not build graph by default; **opt in** via GradientTape In PyTorch, all ops build graph by default; **opt out** via torch.no\_grad



```
N, Din, H, Dout = 16, 1000, 100, 10<br>x = tf.random.normal((N, Din))
y = tf.random.normal((N import tensorflow as tf<br>w1 = tf.Variable(tfran<br>w2 = tf.Variable(tfran<sub>N, Din, H, Dout = 16, 1000, 100, 10learning rate = 1e-6x = tf.random.normal((N, Din))y = tf.random.normal((N, Dout))for t in range(1000):
  with \text{tf.g.} \text{find}(\text{p.v.}, \text{f.})w2 = tf.Variable(tf.random.normal((H, Dout)))h = tf.maximum(tf.m)y pred = tf. matmul(<br>diff = y pred - y
                              learning_rate = 1e-6for t in range(1000):
     loss = tf.readuce mewith tf.GradientTape() as tape:
                                   h = tf.maximum(tf.matmul(x, w1), 0)y pred = tf.matmul(h, w2)grad w1, grad w2 = tadiff = y pred - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, ax)w2 \texttt{.assign}(w2 - learning)grad_w1, grad_w2 = tape.gradient(loss, [w1, w2])w1.\text{assign}(w1)wl. assign(w1 - learning rate * grad w1)w2 \text{.}assign(w2)w2 \text{.}assign(w2 - learning rate * grad w2)
```


#### TensorFlow 2.0: TensorFlow 2. Cimport tensorflow as tf TensorFlow 2.0: TensorFlow Z



#### ask the tradigate cico diametrico Ask the tape to compute gradients





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-6for 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 - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, a)grad w1, grad w2 = \text{tape.gradient}(loss, [w1, w2])wl. assign(w1 - learning rate * grad w1)w2 \text{.}assign(w2 - learning rate * grad w2)
```


### TensorFlow 2.0: Dynamic Graphs TensorFlow







```
x = tf.random.normal((N, Din))y = tf.random.normal((N, Dout))wl = tf.Variable(tf.random.normal((Din, H)))w2 = tf.Variable(tf.random.normal((H, Dout)))learning rate = 1e-6for 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 - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, axis=1))grad_w1, grad_w2 = tape.gradient(loss, [wl, w2])w1.assign(wl - learning rate * grad wl)
  w2 \text{.} assign(w2 - learning rate * grad w2)
```


TensorFlow 2.0: ction that<br>forward. backward, and update Define a function that implements forward,

Annotating with ction into a grai  $\cdot$  to torch iit script) the function is the function into  $\frac{1}{2}$ 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 - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, axis=1))grad_w1, grad_w2 = tape.gradient(loss, [w1, w2])w1.assign(w1 - learning rate * grad w1)
  w2 \text{.} 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))wl = tf.Variable(tf.random.normal((Din, H)))w2 = tf.Variable(tf.random.normal((H, Dout)))learning rate = 1e-6for t in range(1000):
```

```
loss = step(x, y, w1, w2)
```


## TensorFlow 2.0: Static Graphs TensorFlow 2.0: Static Graphs



TensorFlow 2.0: ction that<br>forward. backward, and update Define a function that implements forward, Define a function that implements forward,

Annotating with ction into a grai  $\cdot$  to torch iit script)  $\sim$  to function ip  $\sim$ tf.function will compile tf.function will compile the function into a graph! the function into a graph! (similar to torch.jit.script) (similar to torch.jit.script)

 $\Gamma$  aroph.com  $r_{\rm F}$ include gradient computation and update, the computation and update,  $\frac{1}{2}$ (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 - yloss = tf.reduce_mean(tfىreduce_sum(diff ** 2, axis=1))grad w1, grad w2 = tape.gradient(loss, [w1, w2])
  w1.assign(w1 - learning rate * grad w1)
  w2 \text{.} 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-6for t in range(1000):
  loss = step(x, y, w1, w2)
```


#### TensorFlow 2.0: Static Graphs TensorFlow 2.0: Static Graphs LensorFlo Static Graphs



### TensorFlow 2.0: Static Graphs TensorFlow 2.0:



Static Graphs  $W1$ return loss Call the compiled step function in the training Call the compiled step



loop

```
y pred = tf.matmul(h, w2)diff = y pred - yloss = tf.readuce mean(tf.readuce sum(diff ** 2, axis=1))grad w1, grad w2 = tape.gradient(loss, [w1, w2])
  wl. assign(w1 - learning rate * grad w1)w2 \text{.} assign(w2 - learning rate * grad w2)
N, Din, H, Dout = 16, 1000, 100, 10
x = tf.random.normal((N, Din))y = tf.random.normal((N, Dout))wl = tf.Variable(tf.random.normal((Din, H)))w2 = tf.Variable(tf.random.normal((H, Dout)))learning_rate = 1e-6for t in range(1000):
  loss = step(x, y, w1, w2)learning rate = 1e-6for 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. Mean Squared Error()
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





```
rt tensorflow as tf
tensorflow.keras.models import Sequential
tensorflow.keras.layers import InputLayer, Dense ase
in, H, Dout = 16, 1000, 100, 10
\mathbf{r} = Sequential()
l.add(InputLayer(input shape=(Din,)))
l.add(Dense(units=H, activation='relu'))
l.add(Dense(units=Dout))
ms = model.trainable variables
fn = tf.keras.losses.MeanSquaredError()= tf.keras.optimizers.SGD(learning rate=1e-6)
tf.random.normal((N, Din))
tf.random.normal((N, Dout))
t in range(1000):
th tf.GradientTape() as tape:
y pred = model(x)loss = loss_fn(y_pred, y)ads = tape.gradient(loss, params)
t.apply_gradients(zip(grads, params))
```




# Keras: High-level API

Keras gives you Keras gives you non loss  $f$ ions and optimization  $i$ thms common loss functions and optimization algorithms



```
N, Din, H, Dout = 16, 1000, 100, 10
model = Sequential()model.add(InputLayer(input shape=(Dim,)))model.add(Dense(units=H, activation='relu'))
model.add(Dense(units=Dout))
params = model.trainable variables
loss fn = tf. keras. losses. Mean Squared Error ()
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 Kerasian Ke<br>Kerasian Kerasian Ke

Compute loss, build graph Forward pass:







```
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 fin(y pred, y)grads = tape.gradient(loss, params)
  opt.apply_gradients(zip(grads, params))
```
updates parameters Optimizer object



 $\frac{f}{\lambda}$ 

٠



# Keras: High-level API

Keras: High-level API



```
import tensorflow as tf
from tensorflow.keras.models import Sequential
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 fin = 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 fin(y pred, y)grads = tape.gradient(loss,params)opt.apply_gradients(zip(grads, params))
    opt.apply_gradients(zip(grads, params))
```


### Keras: High-level API 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)
```
Optimizer computes gradients and updates parameters Prinsipality of the computer ales paramele



Justin Johnson Lecture 12 - February 12 - February 16, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2<br>Justin Johnson Lecture 12 - February 16, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 2022, 20



Keras: High-level API

100



```
Keras: High-level API
                              from tensorflow.keras.models import Sequential
                              fr
                                 import tensorflow as tf
Keras: High-level API
                                  from tensorflow.keras.layers import InputLayer, Dense
                              mo
                                 N, Din, H, Dout = 16, 1000, 100, 10
                              mo
                              mo
                                 model = Sequential()momodel.add(InputLayer(input_shape=(Din,)))
                                 model.add(Dense(units=H, activation='relu'))
                              pa
                                 model.add(Dense(units=Dout))
                              lo
                                 params = model.trainable_variableop
                                 loss fin = tf.keras.losses.MeanSquaredError()x
                                 opt = tf.keras.optimizers.SGD(learning_rate=1e-6)y
                                 x = tf.random.normal((N, Din))de
                                 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! dd logging to code Add logging to code to record loss, stats Run server and get pretty gra











TensorBoard



## TensorBoard

#### AISO WOrks With **Pylorch!** Also works with [PyTorch!](https://pytorch.org/docs/stable/tensorboard.html)

### Add logging to code to record loss, stats, etc







TensorBoard



### PyTorch vs TensorFlow PyTorch vs Tensor<br>FlowTorch vs Tensor<br>FlowTorch vs Tensor

### **PyTorch**

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



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

#### **TensorFlow 1.0**

#### **TensorFlow 2.0**



# Summary: Deep Learning Software

Summary: Software



### **Static Graphs** vs **Dynamic Graphs**

### **PyTorch** vs **TensorFlow**





