

Lecture 6: Hardware and Software

Hardware Computation Units, Dynamic & Static Computational
Graph, PyTorch & TensorFlow

Today

- Deep learning hardware
 - CPU, GPU
- Deep learning software
 - PyTorch and TensorFlow
 - Static and Dynamic computation graphs

Deep Learning Hardware

CPU vs GPU

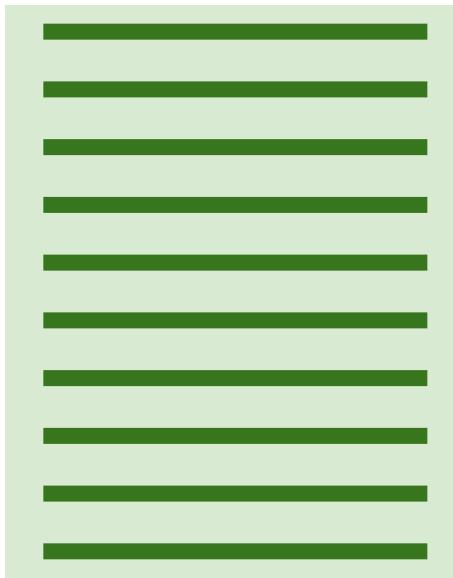
	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$385	~540 GFLOPs FP32
GPU (NVIDIA RTX 2080 Ti)	3584	1.6 GHz	11 GB GDDR6	\$1199	~13.4 TFLOPs FP32

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

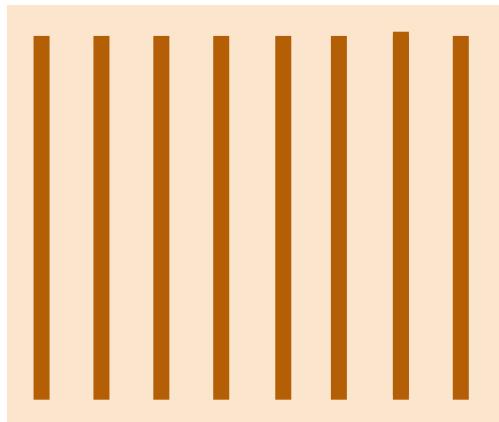
GPU: More cores, but each core is much slower and “dumber”; great for parallel tasks

Example: Matrix Multiplication

$A \times B$

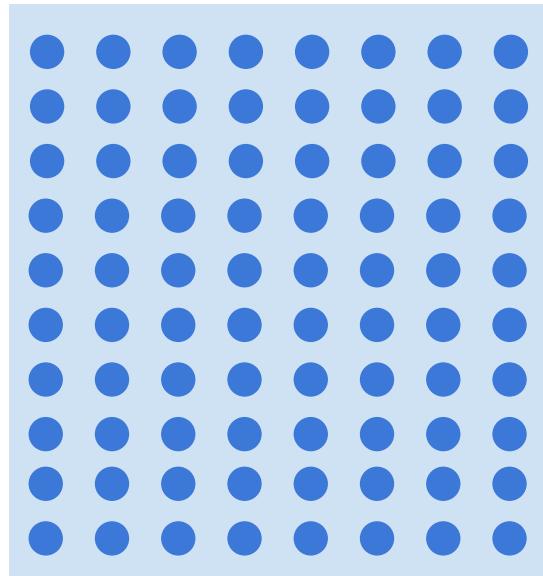


$B \times C$



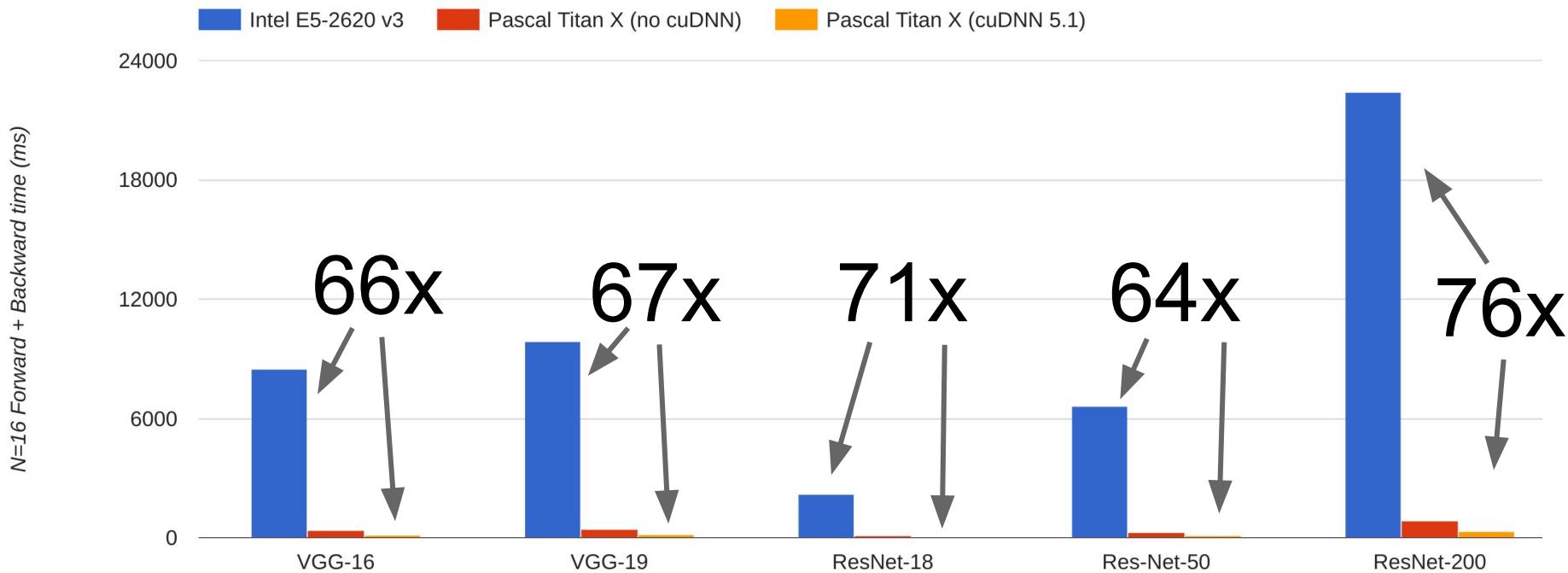
=

$A \times C$



CPU vs GPU in practice

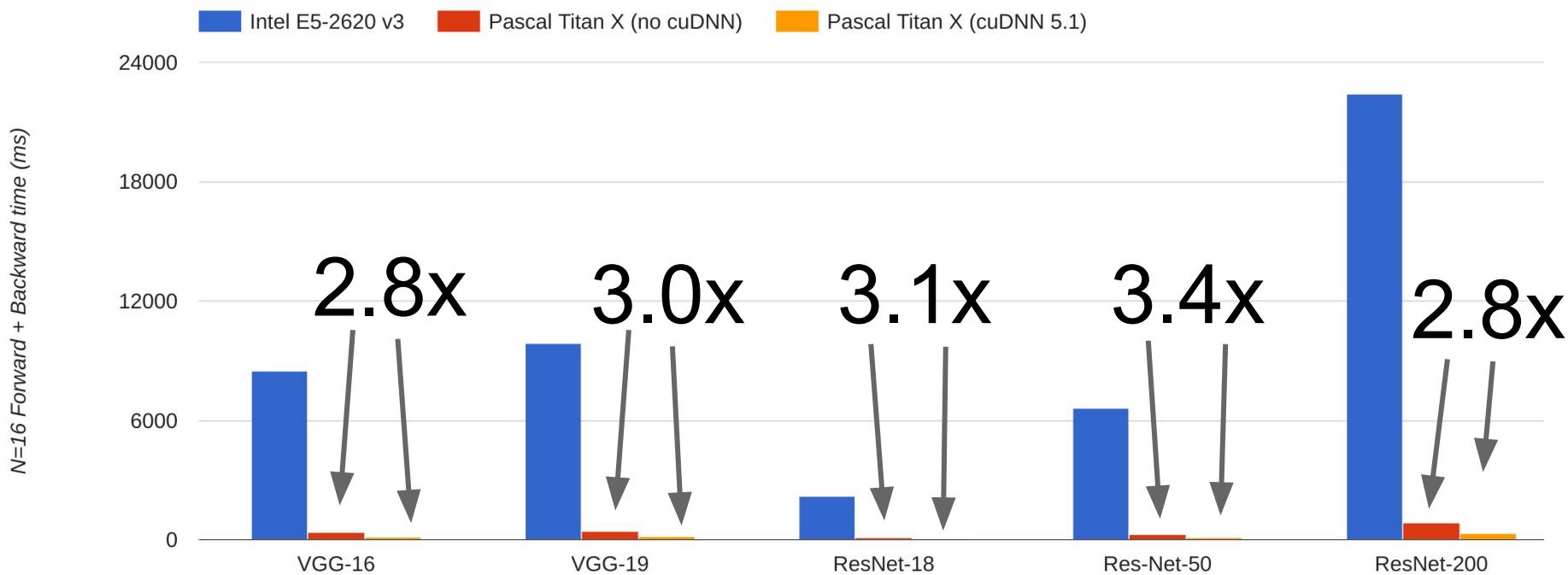
(CPU performance not well-optimized, a little unfair)



Data from <https://github.com/jcjohnson/cnn-benchmarks>

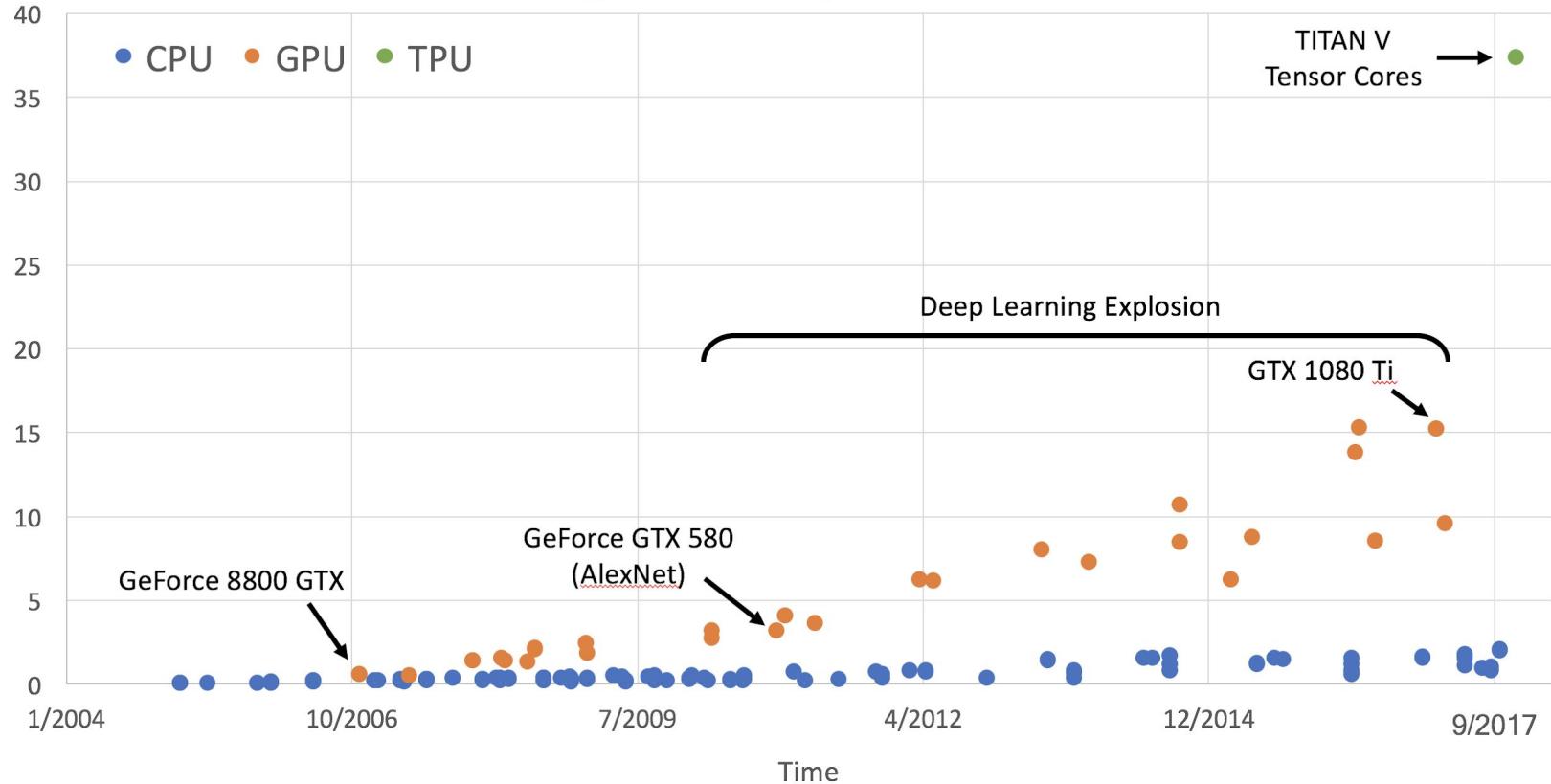
CPU vs GPU in practice

cuDNN much faster than
“unoptimized” CUDA



Data from <https://github.com/jcjohnson/cnn-benchmarks>

GigaFLOPs per Dollar



NVIDIA

vs

AMD

CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$385	~540 GFLOPs FP32
GPU NVIDIA RTX 2080 Ti	3584	1.6 GHz	11 GB GDDR 6	\$1099	~13 TFLOPs FP32 ~114 TFLOPs FP16
GPU (Data Center) NVIDIA V100	5120 CUDA, 640 Tensor	1.5 GHz	16/32 GB HBM2	\$2.5/hr (GCP)	~8 TFLOPs FP64 ~16 TFLOPs FP32 ~125 TFLOPs FP16
TPU Google Cloud TPUv3	2 Matrix Units (MXUs) per core, 4 cores	?	128 GB HBM	\$8/hr (GCP)	~420 TFLOPs (non-standard FP)

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and “dumber”; great for parallel tasks

TPU: Specialized hardware for deep learning

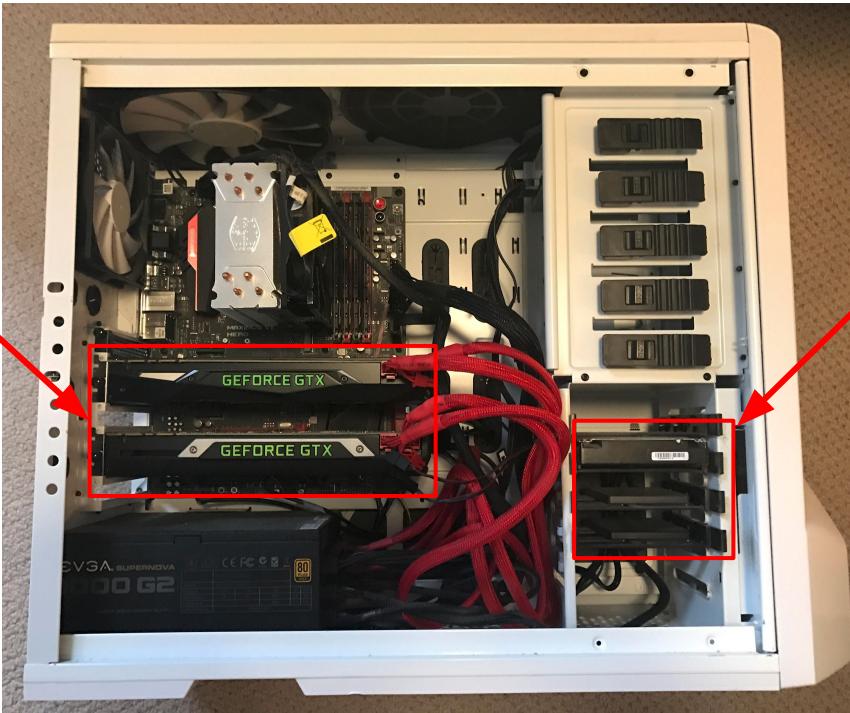
Programming GPUs

- CUDA (NVIDIA only)
 - Write C-like code that runs directly on the GPU
 - Optimized APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
 - Similar to CUDA, but runs on anything
 - Usually slower on NVIDIA hardware
- HIP <https://github.com/ROCm-Developer-Tools/HIP>
 - New project that automatically converts CUDA code to something that can run on AMD GPUs
- Stanford CS 149: <http://cs149.stanford.edu/fall19/>

CPU / GPU Communication

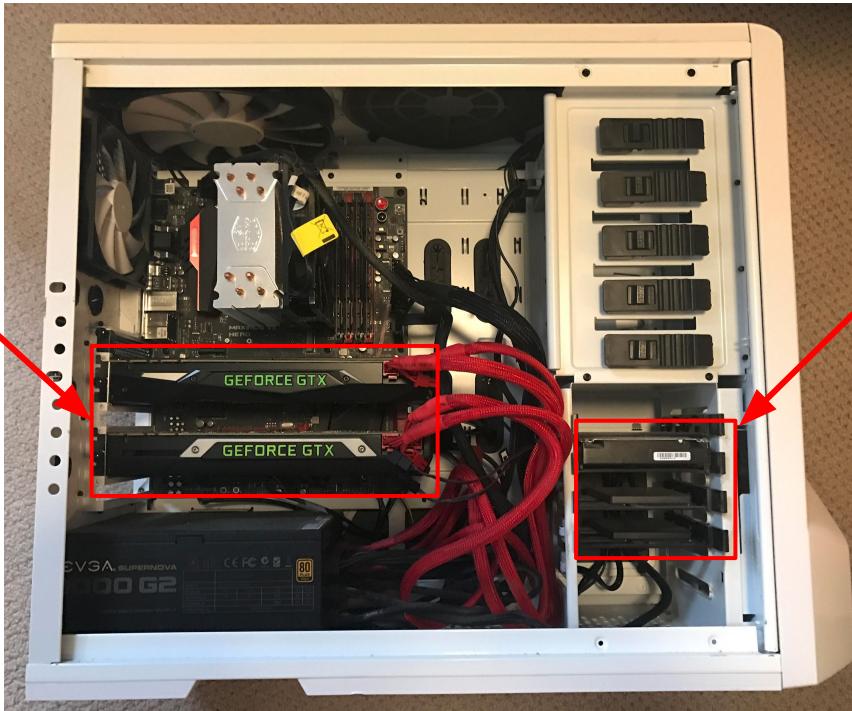
Model
is here

Data is here



CPU / GPU Communication

Model
is here



Data is here

If you aren't careful, training can bottleneck on reading data and transferring to GPU!

Solutions:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data

Deep Learning Software

A zoo of frameworks!

Caffe
(UC Berkeley)



Caffe2
(Facebook)
mostly features absorbed
by PyTorch

Torch
(NYU / Facebook)



PyTorch
(Facebook)

Theano
(U Montreal)



TensorFlow
(Google)

We'll focus on these

PaddlePaddle
(Baidu)

Chainer
(Preferred Networks)
The company has officially migrated its research
infrastructure to PyTorch

MXNet
(Amazon)

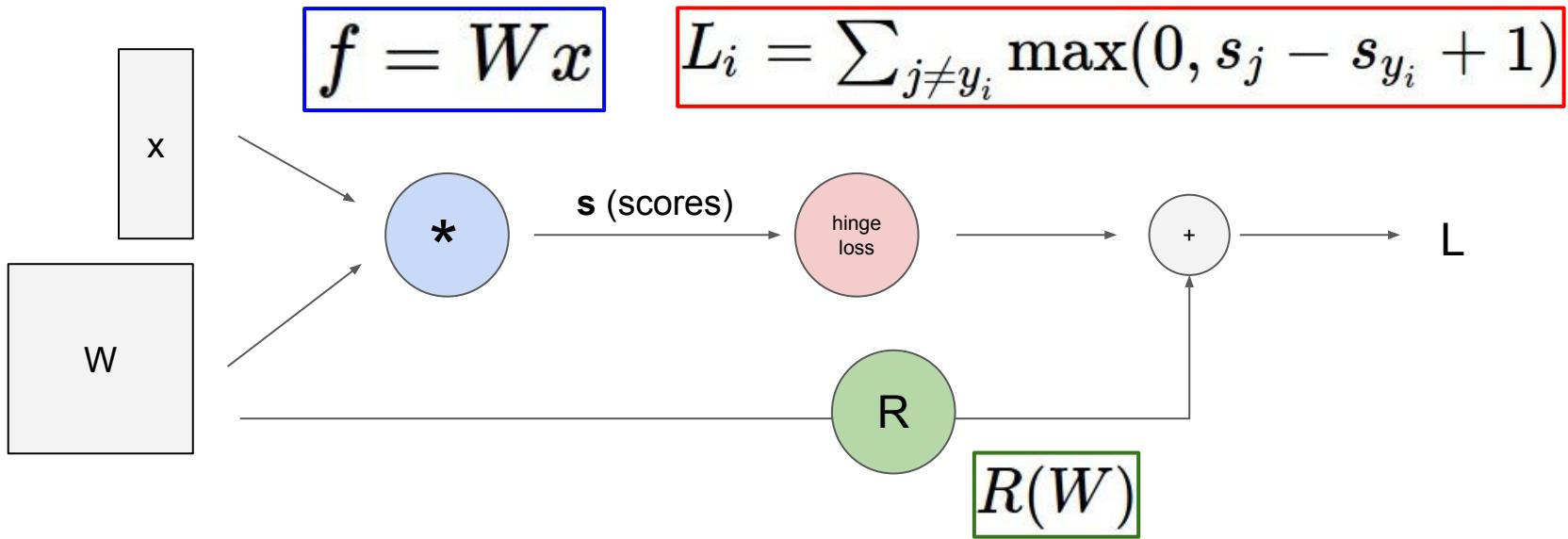
Developed by U Washington, CMU, MIT,
Hong Kong U, etc but main framework of
choice at AWS

CNTK
(Microsoft)

JAX
(Google)

And others...

Recall: Computational Graphs



Recall: Computational Graphs

input image

weights

loss

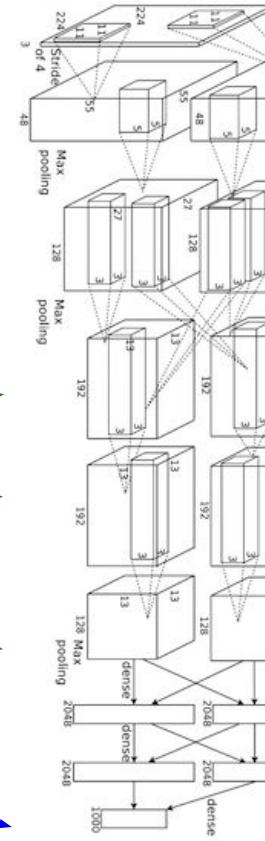


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Recall: Computational Graphs

input image

loss

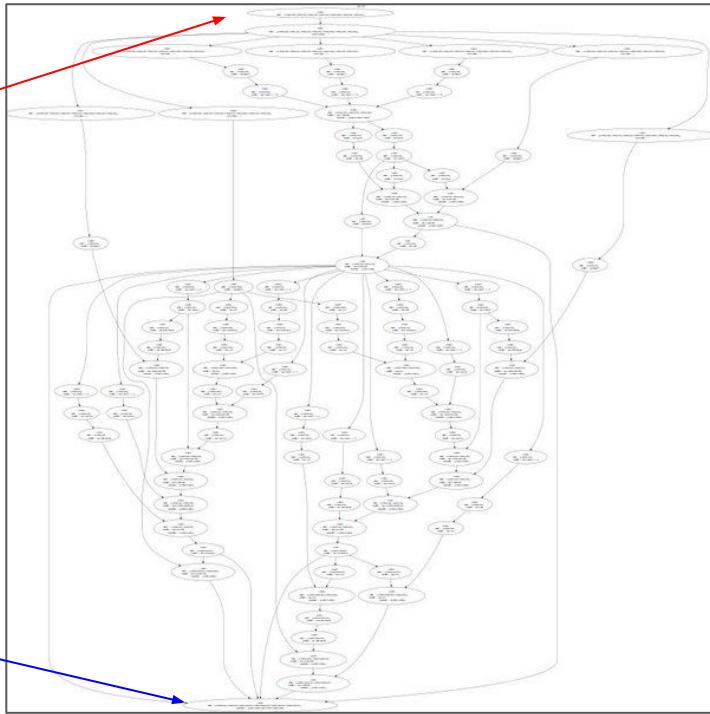


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

The point of deep learning frameworks

- (1) Quick to develop and test new ideas
- (2) Automatically compute gradients
- (3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, OpenCL, etc)

Computational Graphs

Numpy

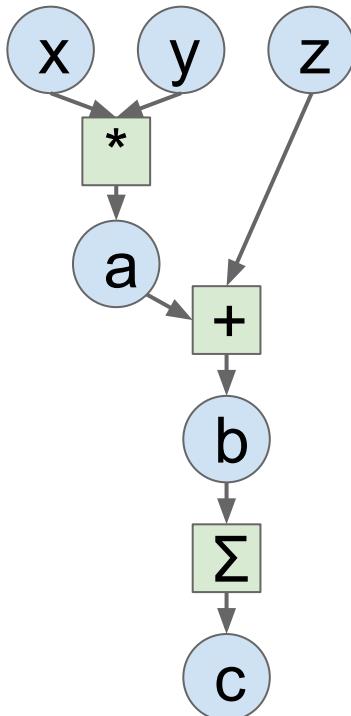
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Good:

Clean API, easy to write numeric code

Bad:

- Have to compute our own gradients
- Can't run on GPU

Computational Graphs

Numpy

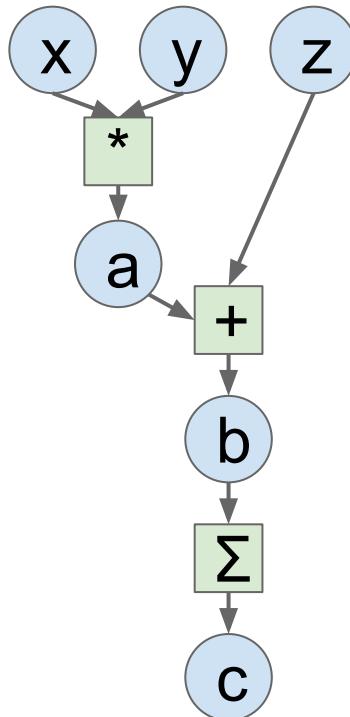
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!

Computational Graphs

Numpy

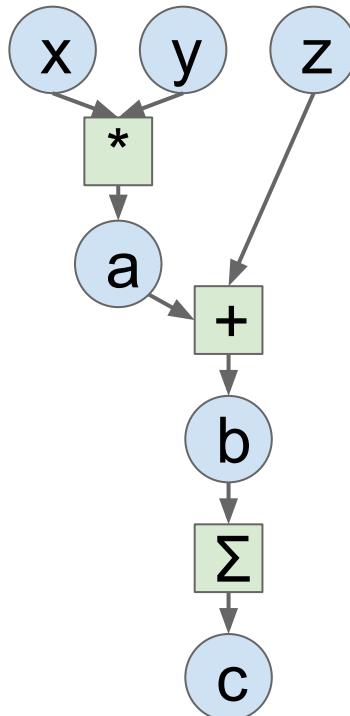
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

PyTorch handles gradients for us!

Computational Graphs

Numpy

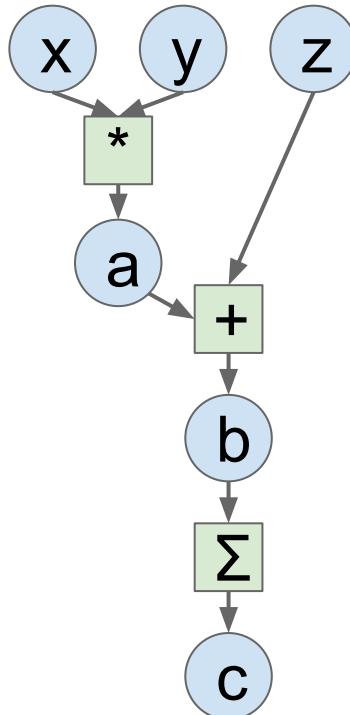
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch
device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!

PyTorch

(More details)

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: Versions

For this class we are using **PyTorch version 1.4**
(Released January 2020)

Major API change in release 1.0

Be careful if you are looking at older PyTorch code (<1.0)!

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('cuda:0')

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

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

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

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

PyTorch: Autograd

Creating Tensors with
requires_grad=True enables
autograd

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

```
import torch

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

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

    loss.backward()

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

PyTorch: Autograd

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

Do want gradients with
respect to weights

```
import torch

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

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

    loss.backward()

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

PyTorch: Autograd

Forward pass looks exactly the same as before, but we don't need to track intermediate values - PyTorch keeps track of them for us in the graph

```
import torch

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

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

    loss.backward()

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

PyTorch: Autograd

Compute gradient of loss
with respect to w1 and w2

```
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

Make gradient step on weights, then zero them. `Torch.no_grad` means “don’t build a computational graph for this part”

```
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

PyTorch methods that end in underscore modify the Tensor in-place; methods that don't return a new Tensor

```
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 Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Use ctx object to “cache” values for the backward pass, just like cache objects from A2

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Use ctx object to “cache” values for the backward pass, just like cache objects from A2

Define a helper function to make it easy to use the new function

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)
```

PyTorch: New Autograd Functions

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)
```

Can use our new autograd
function in the forward pass

```
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 = my_relu(x.mm(w1)).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 Autograd Functions

```
def my_relu(x):
    return x.clamp(min=0)
```

In practice you almost never need to define new autograd functions! Only do it when you need custom backward. In this case we can just use a normal Python function

```
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 = my_relu(x.mm(w1)).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: 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

Define our model as a sequence of layers; each layer is an object that holds learnable 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)

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

```
import torch

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

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

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
```

```
with torch.no_grad():
    for param in model.parameters():
        param -= learning_rate * param.grad
model.zero_grad()
```

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



PyTorch: optim

Use an **optimizer** for different update rules

```
import torch

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

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

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

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

    loss.backward()

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

PyTorch: optim

```
import torch

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

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

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

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

After computing gradients, use
optimizer to update params
and zero gradients



PyTorch: nn

Define new Modules

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

Modules can contain weights or other modules

You can define your own Modules using autograd!

```
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

Define new 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

Define new 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

Define new Modules

Define forward pass using child modules

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

Define new Modules

Construct and train an instance of our model

```
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

Define new 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

Define new 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

Define new Modules

Stack multiple instances of the component in a sequential



```
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: Pretrained Models

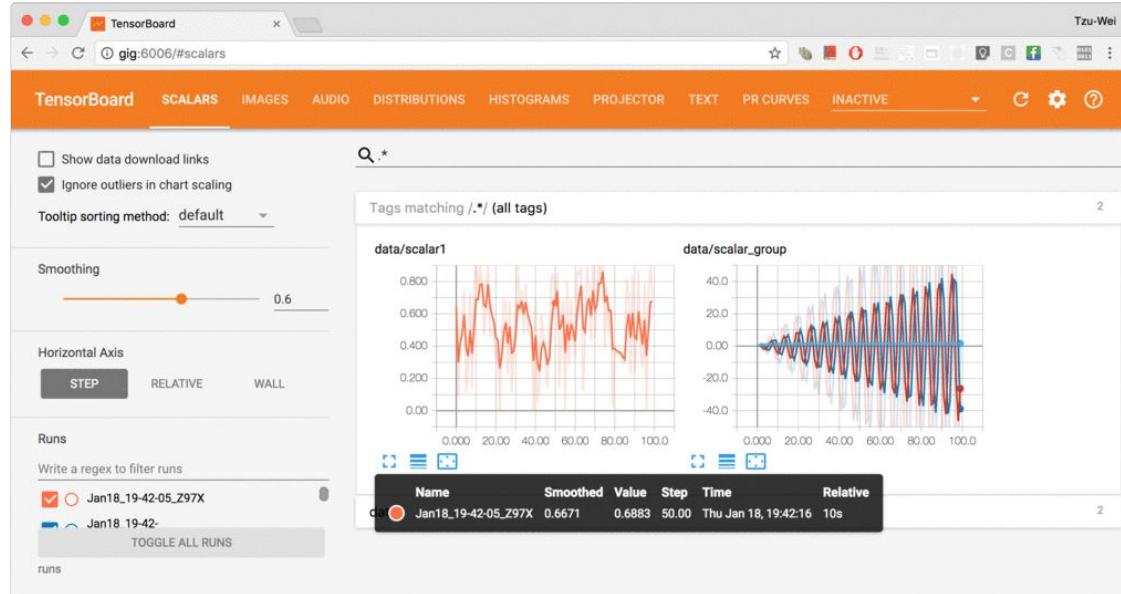
Super easy to use pretrained models with torchvision
<https://github.com/pytorch/vision>

```
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)
```

PyTorch: torch.utils.tensorboard

A python wrapper around
Tensorflow's web-based
visualization tool.



This image is licensed under CC-BY 4.0; no changes were made to the image

PyTorch: Computational Graphs

input image

loss

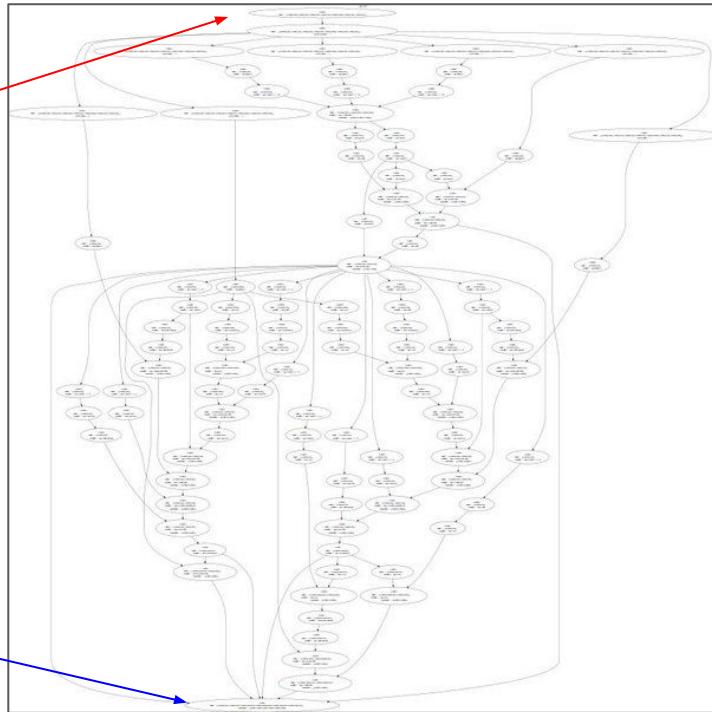


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

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

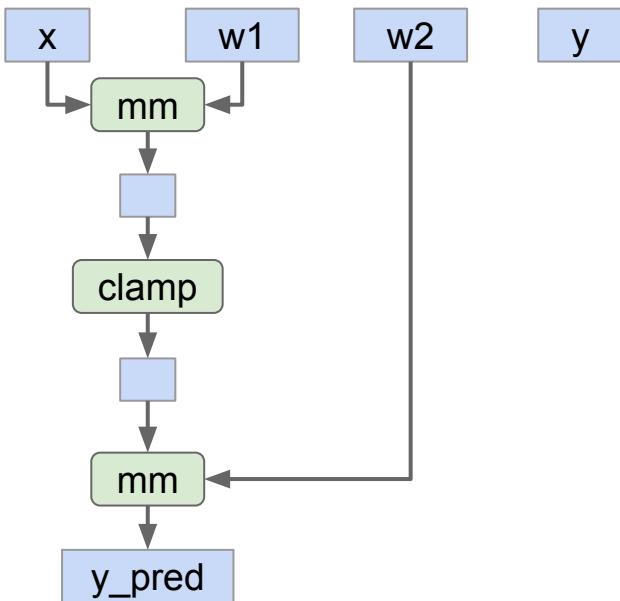
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

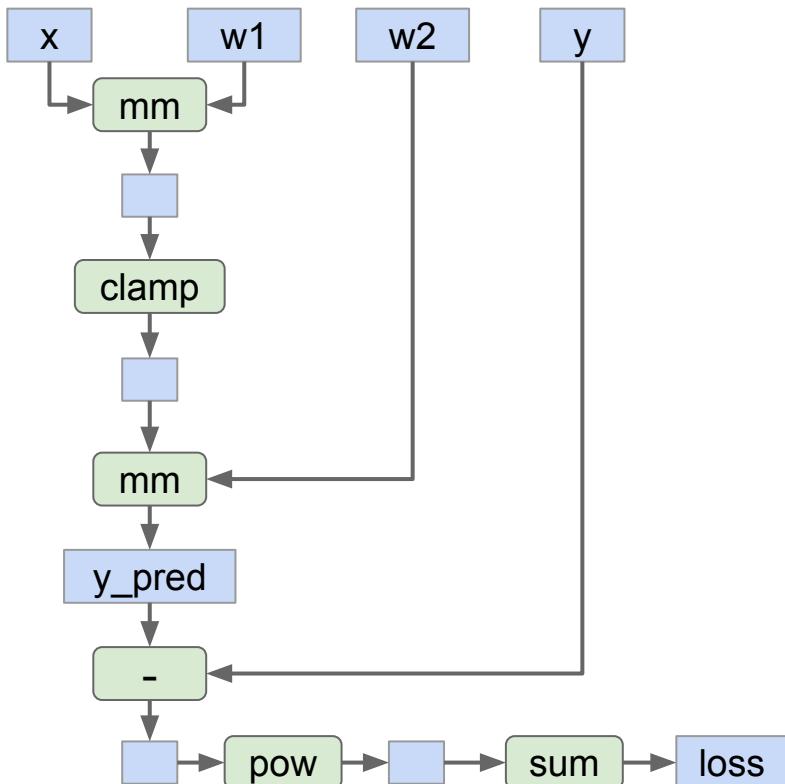
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

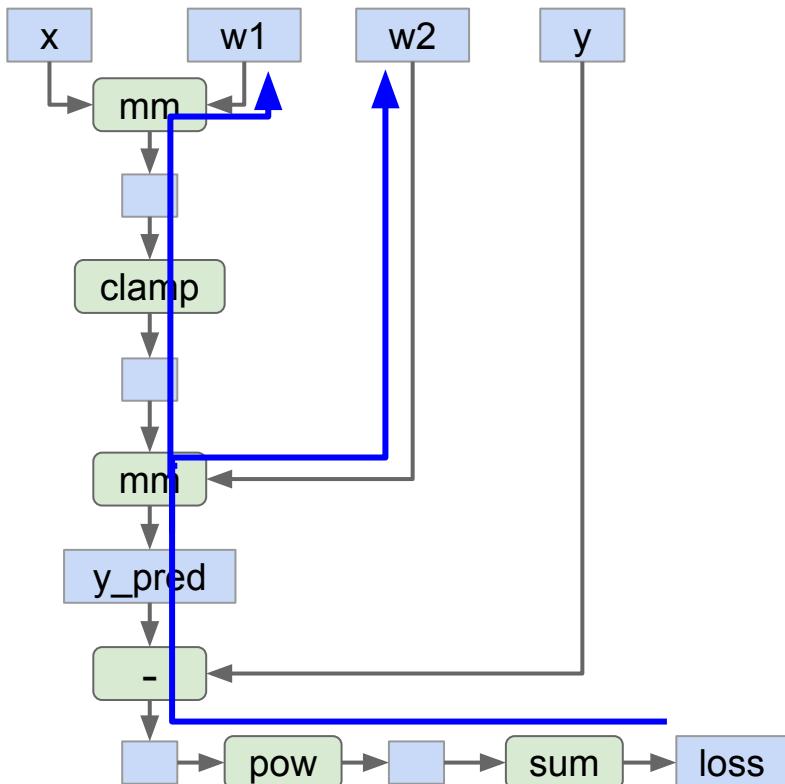
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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()
```

Search for path between loss and w1, w2
(for backprop) AND perform computation

PyTorch: Dynamic Computation Graphs

x

w1

w2

y

```
import torch

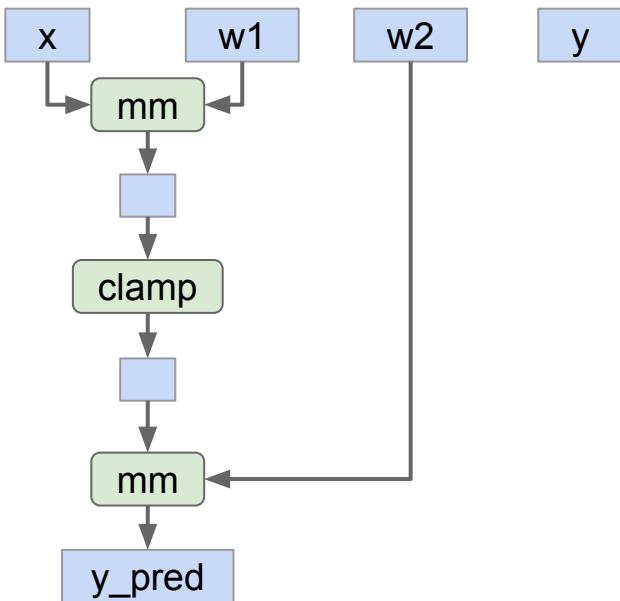
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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()
```

Throw away the graph, backprop path, and rebuild it from scratch on every iteration

PyTorch: Dynamic Computation Graphs



```
import torch

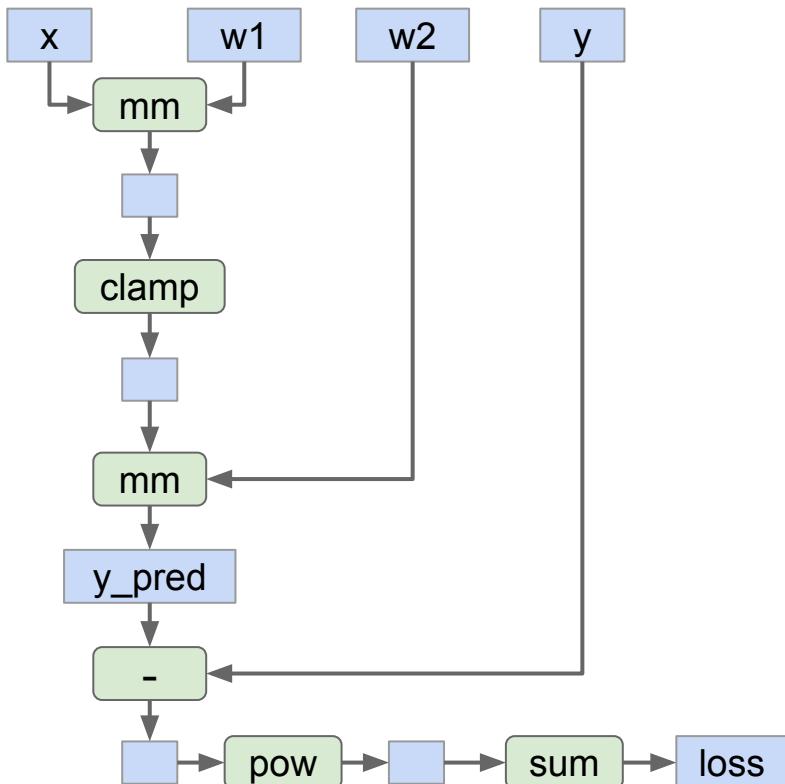
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

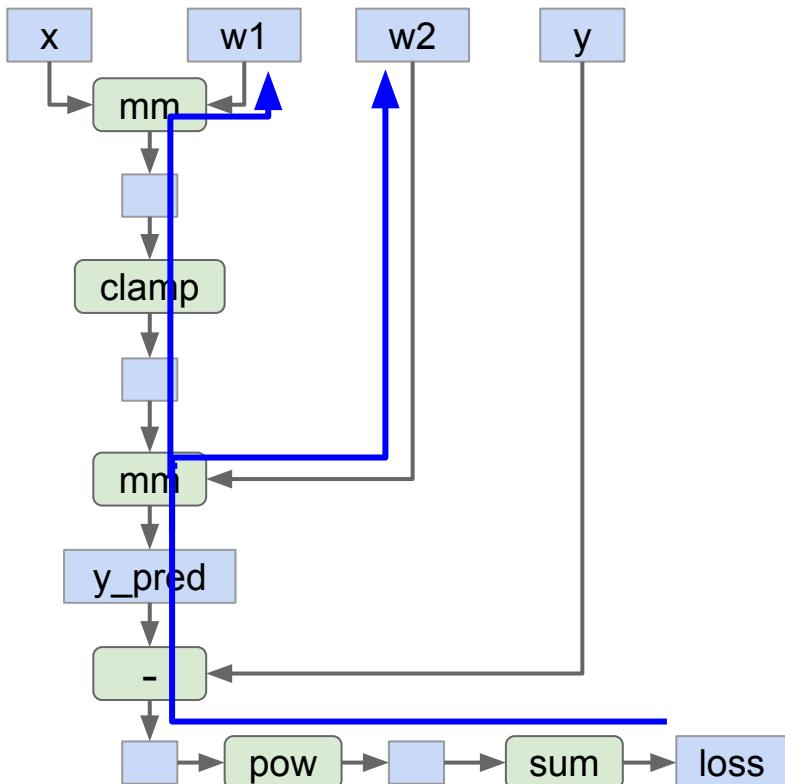
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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()
```

Search for path between loss and w1, w2
(for backprop) AND perform computation

PyTorch: Dynamic Computation Graphs

Building the graph and
computing the graph happen at
the same time.

Seems inefficient, especially if we
are building the same graph over
and over again...

```
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()
```

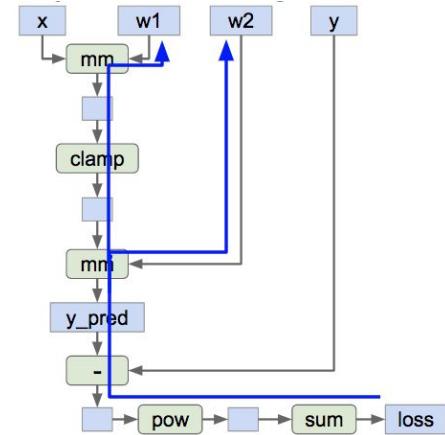
Static Computation Graphs

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)
```



TensorFlow

TensorFlow Versions

Pre-2.0 (1.14 latest)

Default static graph,
optionally dynamic
graph (eager mode).

2.1 (March 2020)

Default dynamic graph,
optionally static graph.
We use 2.1 in this class.

TensorFlow: Neural Net (Pre-2.0)

```
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: Neural Net (Pre-2.0)

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+ vs. pre-2.0

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

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))
gradients = tape.gradient(loss, [w1, w2]).
```

Tensorflow 2.0+:
“Eager” Mode by default
`assert(tf.executing_eagerly())`

```
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 1.13

TensorFlow: 2.0+ vs. pre-2.0

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

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))
gradients = tape.gradient(loss, [w1, w2]).
```

Tensorflow 2.0+:
“Eager” Mode by default

```
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 1.13

TensorFlow: 2.0+ vs. pre-2.0

```
N, D, H = 64, 1000, 100
```

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

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))
gradadients = tape.gradient(loss, [w1, w2]).
```

Tensorflow 2.0+:
“Eager” Mode by default

```
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 1.13

TensorFlow: Neural Net

Convert input numpy arrays to TF **tensors**.
Create weights as tf.Variable

N, D, H = 64, 1000, 100

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

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))
gradients = tape.gradient(loss, [w1, w2]).
```

copy value between numpy and Tensorflow are expensive and slow, especially on GPU with large weights and gradients. We need fix it.

Using tf.Variable instead. tf.Variable is something a value that lives inside the computational graph and it's going to persist inside the computational graph across different times when you run the same graph.

TensorFlow: Neural Net

Use `tf.GradientTape()`
context to build
dynamic computation
graph.

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
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))
gradients = tape.gradient(loss, [w1, w2]).
```

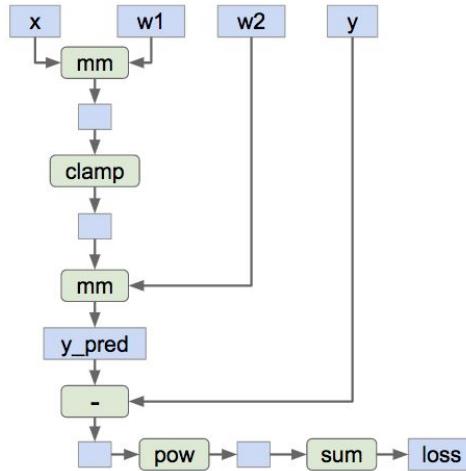
TensorFlow: Neural Net

All forward-pass operations in the contexts (including function calls) gets traced for computing gradient later.

```
N, D, H = 64, 1000, 100  
  
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)  
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)  
w1 = tf.Variable(tf.random.uniform((D, H))) # weights  
w2 = tf.Variable(tf.random.uniform((H, D))) # weights  
  
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))  
    gradients = tape.gradient(loss, [w1, w2]).
```



TensorFlow: Neural Net



Forward pass

$N, D, H = 64, 1000, 100$

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

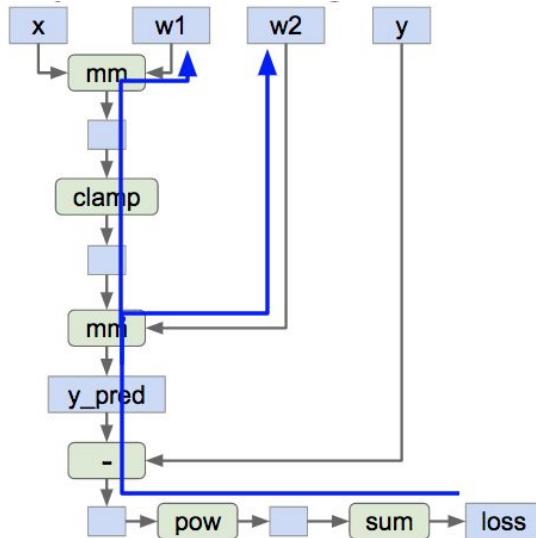
with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradientes = tape.gradient(loss, [w1, w2]).
```

TensorFlow: Neural Net

`tape.gradient()` uses the traced computation graph to compute gradient for the weights

```
N, D, H = 64, 1000, 100  
  
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)  
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)  
w1 = tf.Variable(tf.random.uniform((D, H))) # weights  
w2 = tf.Variable(tf.random.uniform((H, D))) # weights  
  
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))  
gradients = tape.gradient(loss, [w1, w2])
```

TensorFlow: Neural Net



$N, D, H = 64, 1000, 100$

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

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))
gradientes = tape.gradient(loss, [w1, w2])
```

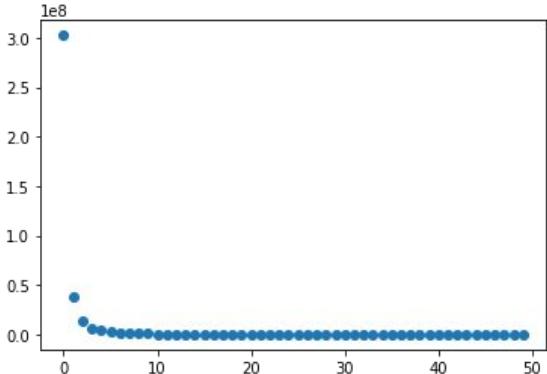
TensorFlow: Neural Net

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

learning_rate = 1e-6
for t in range(50):
    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))
    gradients = tape.gradient(loss, [w1, w2])
    w1.assign(w1 - learning_rate * gradients[0])
    w2.assign(w2 - learning_rate * gradients[1])
```

Train the network: Run the training step over and over, use gradient to update weights

TensorFlow: Neural Net



Train the network: Run the training step over and over, use gradient to update weights

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

learning_rate = 1e-6
for t in range(50):
    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))
    gradients = tape.gradient(loss, [w1, w2])
    w1.assign(w1 - learning_rate * gradients[0])
    w2.assign(w2 - learning_rate * gradients[1])
```

TensorFlow: Optimizer

Can use an **optimizer** to compute gradients and update weights

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
optimizer = tf.optimizers.SGD(1e-6)

learning_rate = 1e-6
for t in range(50):
    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))
    gradients = tape.gradient(loss, [w1, w2])
    optimizer.apply_gradients(zip(gradients, [w1, w2])).
```

TensorFlow: LOSS

Use predefined
common losses

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

optimizer = tf.optimizers.SGD(1e-6)

for t in range(50):
    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.losses.MeanSquaredError()(y_pred, y)
    gradients = tape.gradient(loss, [w1, w2])
    optimizer.apply_gradients(zip(gradients, [w1, w2]))
```

Keras: High-Level Wrapper

Keras is a layer on top of TensorFlow, makes common things easy to do

(Used to be third-party, now merged into TensorFlow)

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

losses = []
for t in range(50):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = tf.losses.MeanSquaredError()(y_pred, y)
    gradients = tape.gradient(
        loss, model.trainable_variables)
    optimizer.apply_gradients(
        zip(gradients, model.trainable_variables))
```

Keras: High-Level Wrapper

Define model as a sequence of layers

Get output by calling the model

Apply gradient to all trainable variables (weights) in the model

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

losses = []
for t in range(50):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = tf.losses.MeanSquaredError()(y_pred, y)
    gradients = tape.gradient(
        loss, model.trainable_variables)
    optimizer.apply_gradients(
        zip(gradients, model.trainable_variables))
```

Keras: High-Level Wrapper

Keras can handle the
training loop for you!

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
model.compile(loss=tf.keras.losses.MeanSquaredError(),
               optimizer=optimizer)
history = model.fit(x, y, epochs=50, batch_size=N.)
```

TensorFlow: High-Level Wrappers

Keras (<https://keras.io/>)

tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)

tf.estimator (https://www.tensorflow.org/api_docs/python/tf/estimator)

Sonnet (<https://github.com/deepmind/sonnet>)

TFLearn (<http://tflearn.org/>)

TensorLayer (<http://tensorlayer.readthedocs.io/en/latest/>)

@tf.function: compile static graph

tf.function decorator
(implicitly) compiles
python functions to
static graph for better
performance

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                               activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

@tf.function
def model_func(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

for t in range(50):
    with tf.GradientTape() as tape:
        y_pred, loss = model_func(x, y)
    gradients = tape.gradient(
        loss, model.trainable_variables)
    optimizer.apply_gradients(
        zip(gradients, model.trainable_variables))
```

@tf.function: compile static graph

Here we compare the forward-pass time of the same model under dynamic graph mode and static graph mode

Ran on Google Colab, April 2020

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph: ", timeit.timeit(lambda: model_dynamic(x, y), number=10))
print("static graph: ", timeit.timeit(lambda: model_static(x, y), number=10))

dynamic graph:  0.02520249200000535
static graph:  0.03932226699998864
```

@tf.function: compile static graph

Ran on Google Colab, April 2020

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss
```

Static graph is *in theory* faster than dynamic graph, but the performance gain depends on the type of model / layer / computation graph.

```
def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph: ", timeit.timeit(lambda: model_dynamic(x, y), number=10))
print("static graph: ", timeit.timeit(lambda: model_static(x, y), number=10))

dynamic graph:  0.02520249200000535
static graph:  0.03932226699998864
```

@tf.function: compile static graph

Ran on Google Colab, April 2020

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss
```

Static graph is *in theory* faster than dynamic graph, but the performance gain depends on the type of model / layer / computation graph.

```
def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

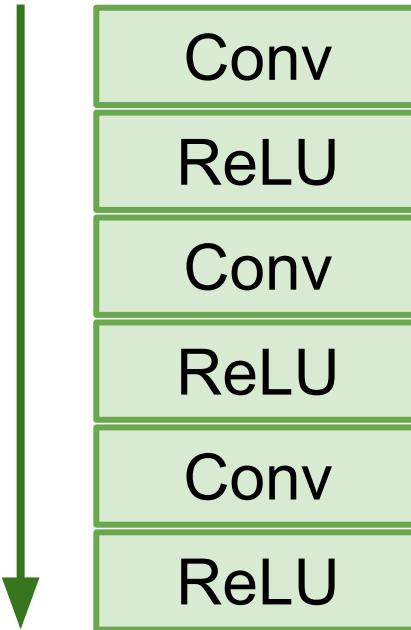
print("dynamic graph:", timeit.timeit(lambda: model_dynamic(x, y), number=1000))
print("static graph:", timeit.timeit(lambda: model_static(x, y), number=1000))

dynamic graph: 2.3648411540000325
static graph: 1.1723986679999143
```

Static vs Dynamic: Optimization

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

The graph you wrote



Equivalent graph with
fused operations



Static PyTorch: ONNX Support

You can export a PyTorch model to ONNX

Run the graph on a dummy input, and save the graph to a file

Will only work if your model doesn't actually make use of dynamic graph - must build same graph on every forward pass, no loops / conditionals

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

dummy_input = torch.randn(N, D_in)
torch.onnx.export(model, dummy_input,
                  'model.proto',
                  verbose=True)
```

Static PyTorch: ONNX Support

```
graph(%0 : Float(64, 1000)
      %1 : Float(100, 1000)
      %2 : Float(100)
      %3 : Float(10, 100)
      %4 : Float(10)) {
    %5 : Float(64, 100) =
    onnx::Gemm[alpha=1, beta=1, broadcast=1,
    transB=1](%0, %1, %2), scope:
    Sequential/Linear[0]
    %6 : Float(64, 100) = onnx::Relu(%5),
    scope: Sequential/ReLU[1]
    %7 : Float(64, 10) = onnx::Gemm[alpha=1,
    beta=1, broadcast=1, transB=1](%6, %3,
    %4), scope: Sequential/Linear[2]
    return (%7);
}
```

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))

dummy_input = torch.randn(N, D_in)
torch.onnx.export(model, dummy_input,
                  'model.proto',
                  verbose=True)
```

After exporting to ONNX, can run the PyTorch model in Caffe2

Static PyTorch: ONNX Support

ONNX is an open-source standard for neural network models

Goal: Make it easy to train a network in one framework, then run it in another framework

Supported by PyTorch, Caffe2, Microsoft CNTK, Apache MXNet

<https://github.com/onnx/onnx>

Static PyTorch: TorchScript

```
graph(%self.1 :  
      __torch__.torch.nn.modules.module.__torch_mangl  
e_4.Module,  
      %input : Float(3, 4),  
      %h : Float(3, 4)):  
  %19 :  
    __torch__.torch.nn.modules.module.__torch_mangl  
e_3.Module =  
    prim::GetAttr[name="linear"](%self.1)  
    %21 : Tensor =  
    prim::CallMethod[name="forward"](%19, %input)  
    %12 : int = prim::Constant[value=1] () #  
<ipython-input-40-26946221023e>:7:0  
    %13 : Float(3, 4) = aten::add(%21, %h, %12) #  
<ipython-input-40-26946221023e>:7:0  
    %14 : Float(3, 4) = aten::tanh(%13) #  
<ipython-input-40-26946221023e>:7:0  
    %15 : (Float(3, 4), Float(3, 4)) =  
    prim::TupleConstruct(%14, %14)  
    return (%15)
```

```
class MyCell(torch.nn.Module):  
    def __init__(self):  
        super(MyCell, self).__init__()  
        self.linear = torch.nn.Linear(4, 4)  
  
    def forward(self, x, h):  
        new_h = torch.tanh(self.linear(x) + h)  
        return new_h, new_h  
  
my_cell = MyCell()  
x, h = torch.rand(3, 4), torch.rand(3, 4)  
traced_cell = torch.jit.trace(my_cell, (x, h))  
print(traced_cell.graph)  
traced_cell(x, h)
```

Build static graph with `torch.jit.trace`

PyTorch vs TensorFlow, Static vs Dynamic

PyTorch

Dynamic Graphs

Static: ONNX,

Caffe2, TorchScript

TensorFlow

Dynamic: Eager

Static: `@tf.function`

Static vs Dynamic: Serialization

Static

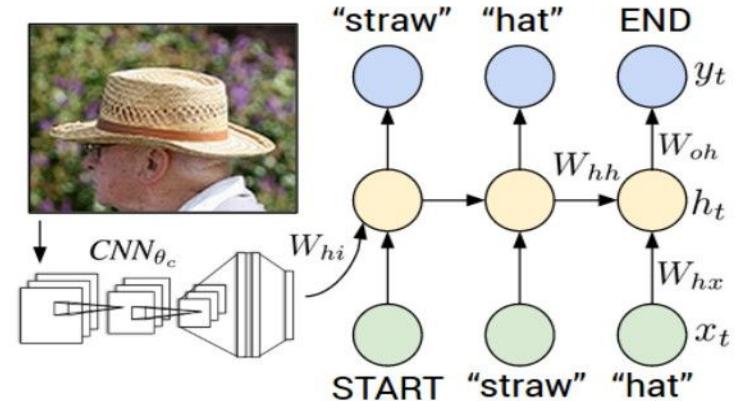
Once graph is built, can **serialize** it and run it without the code that built the graph!

Dynamic

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

Dynamic Graph Applications

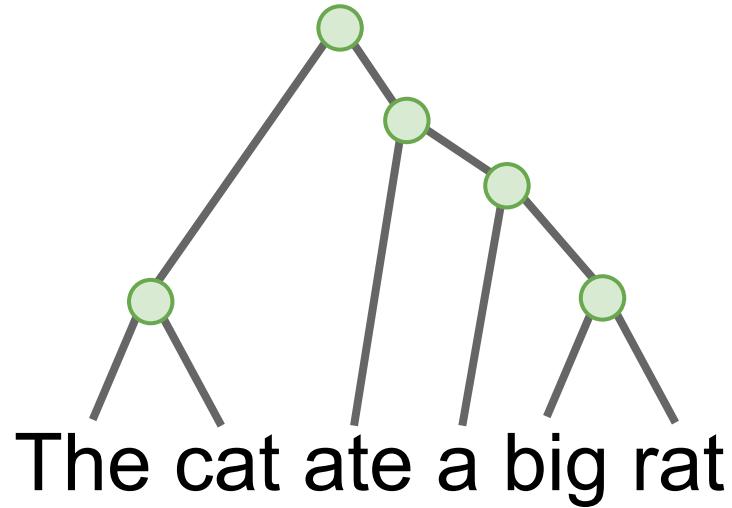
- Recurrent networks



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

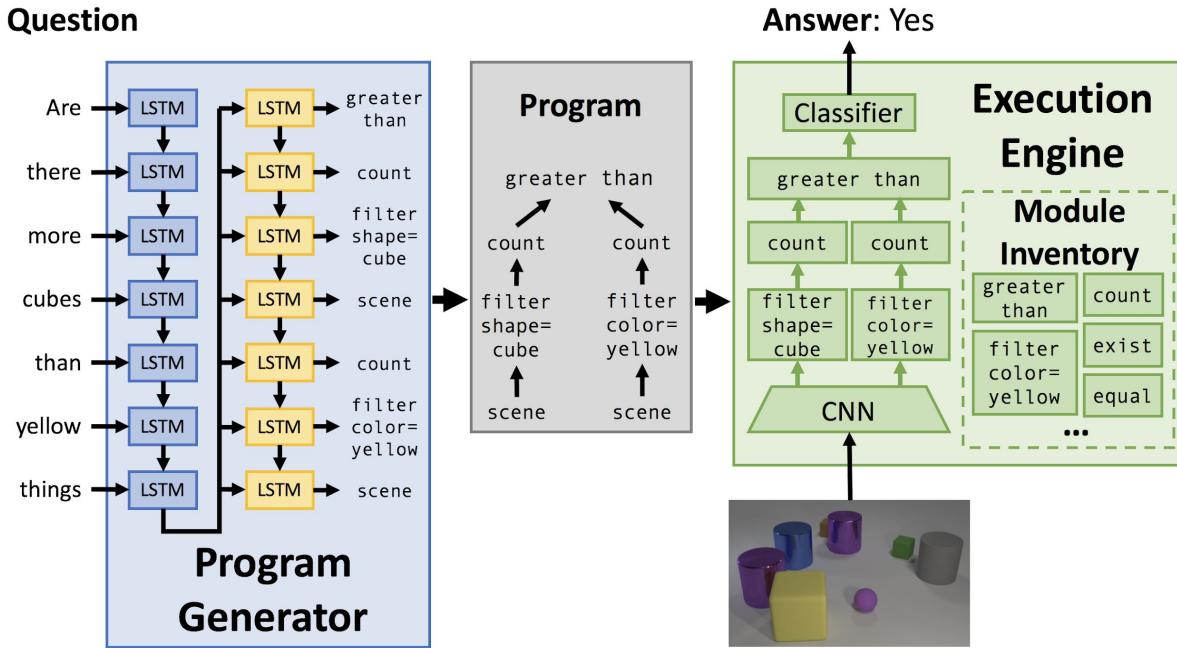
Dynamic Graph Applications

- Recurrent networks
- Recursive networks



Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular networks



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

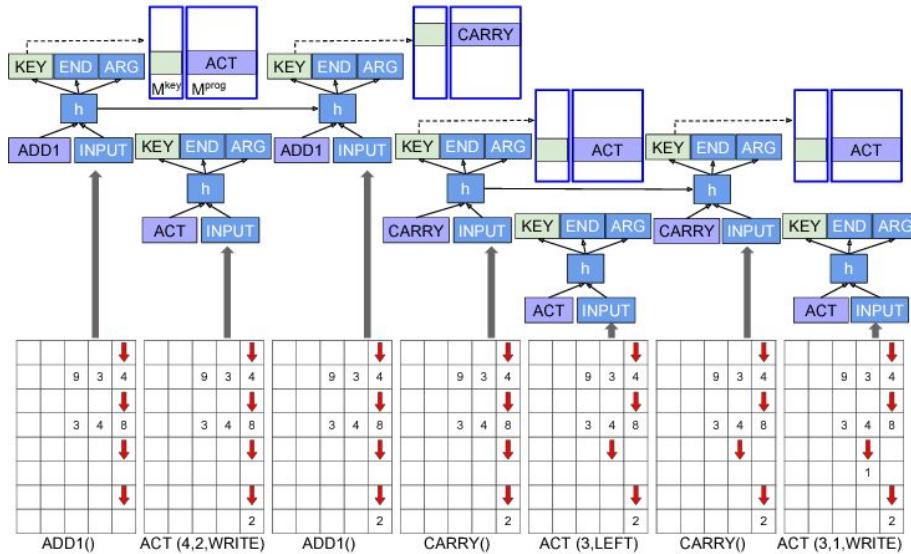
Andreas et al, "Learning to Compose Neural Networks for Question Answering", NAACL 2016

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

Figure copyright Justin Johnson, 2017. Reproduced with permission.

Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular networks
- Neural programs



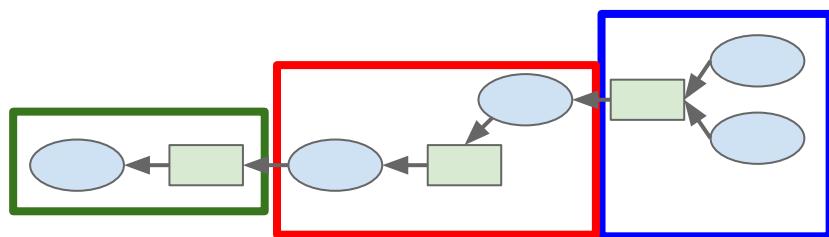
Reed et al., "Neural Programmer-Interpreters", ICLR 2016

Dynamic Graph Applications

- Recurrent networks
- Recursive networks
- Modular Networks
- Neural programs
- (Your creative idea here)

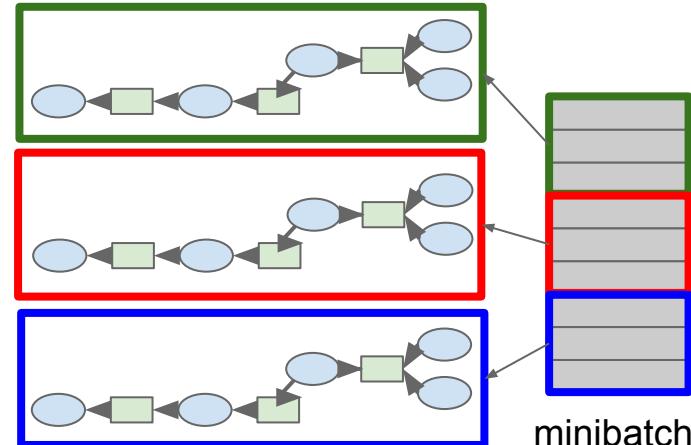
Model Parallel vs. Data Parallel

Model parallel: split computation graph into parts & distribute to GPUs/ nodes



Model Parallel

Data parallel: split minibatch into chunks & distribute to GPUs/ nodes



Data Parallel

PyTorch: Data Parallel

`nn.DataParallel`

Pro: Easy to use (just wrap the model and run training script as normal)

Con: Single process & single node. Can be bottlenecked by CPU with large number of GPUs (8+).

`nn.DistributedDataParallel`

Pro: Multi-nodes & multi-process training

Con: Need to hand-designate device and manually launch training script for each process / nodes.

Horovod (<https://github.com/horovod/horovod>): Supports both PyTorch and TensorFlow

<https://pytorch.org/docs/stable/nn.html#dataparallel-layers-multi-gpu-distributed>

TensorFlow: Data Parallel

`tf.distributed.Strategy`

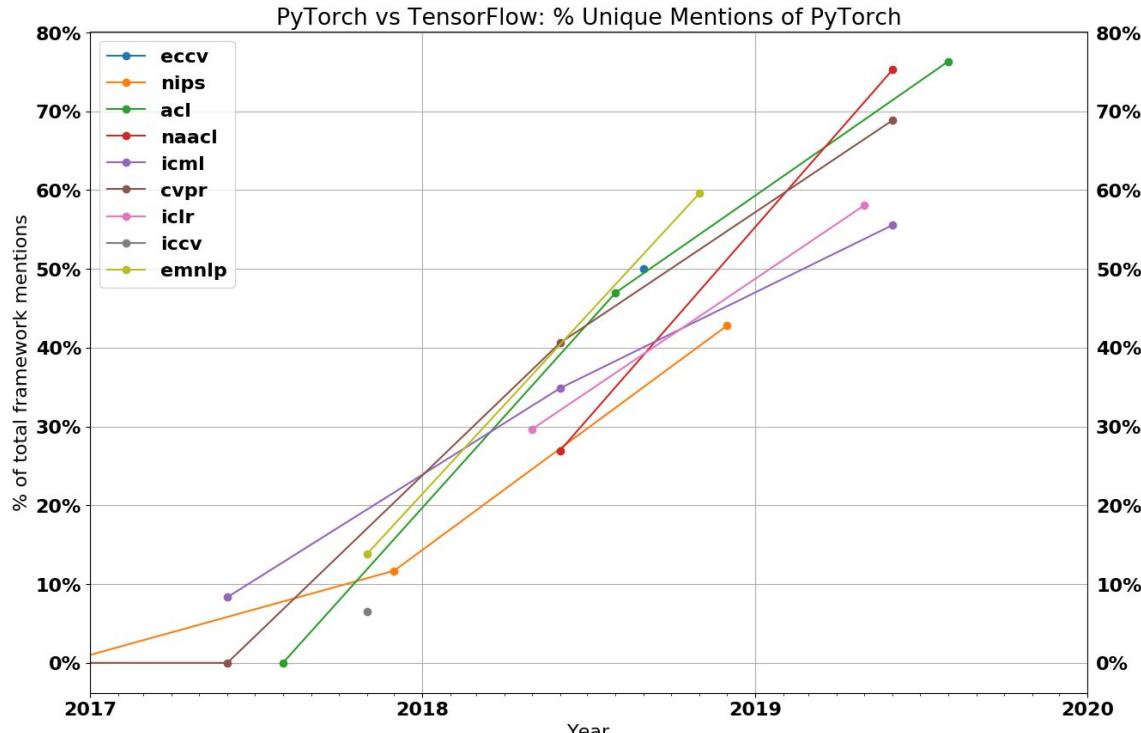
```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10)
    ])

    model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=['accuracy'])
```

<https://www.tensorflow.org/tutorials/distribute/keras>

PyTorch vs. TensorFlow: Academia



<https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/>

PyTorch vs. TensorFlow: Academia

CONFERENCE	PT 2018	PT 2019	PT GROWTH	TF 2018	TF 2019	TF GROWTH
CVPR	82	280	240%	116	125	7.7%
NAACL	12	66	450%	34	21	-38.2%
ACL	26	103	296%	34	33	-2.9%
ICLR	24	70	192%	54	53	-1.9%
ICML	23	69	200%	40	53	32.5%

<https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/>

PyTorch vs. TensorFlow: Industry

- No official survey / study on the comparison.
- A quick search on a job posting website turns up 2389 search results for TensorFlow and 1366 for PyTorch.
- The trend is unclear. Industry is also known to be slower on adopting new frameworks.
- TensorFlow mostly dominates mobile deployment / embedded systems.

My Advice:

PyTorch is my personal favorite. Clean API, native dynamic graphs make it very easy to develop and debug. Can build model using the default API then compile static graph using JIT.

TensorFlow is a safe bet for most projects. Syntax became a lot more intuitive after 2.0. Not perfect but has huge community and wide usage. Can use same framework for research and production. Probably use a high-level framework.

Next Time: Training Neural Networks