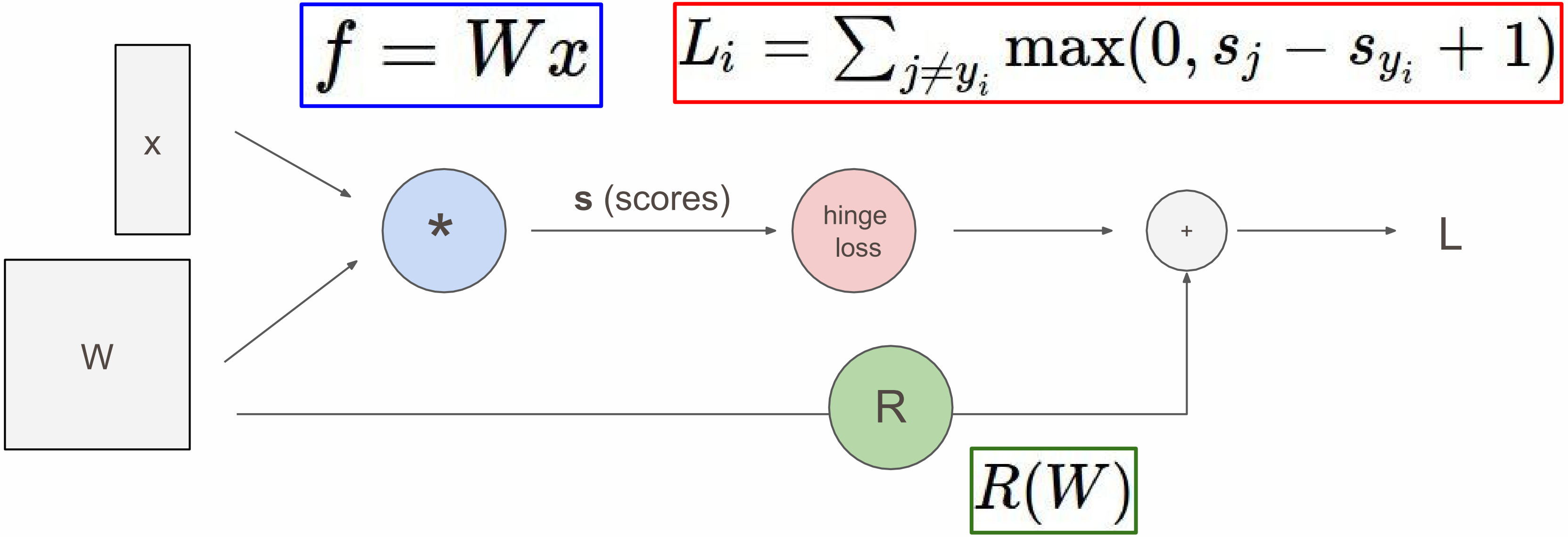
DEEP LEARNING: IMPLEMENTATION

Chih-Chung Hsu (許志仲)
Institute of Data Science
National Cheng Kung University
https://cchsu.info

Computational graphs

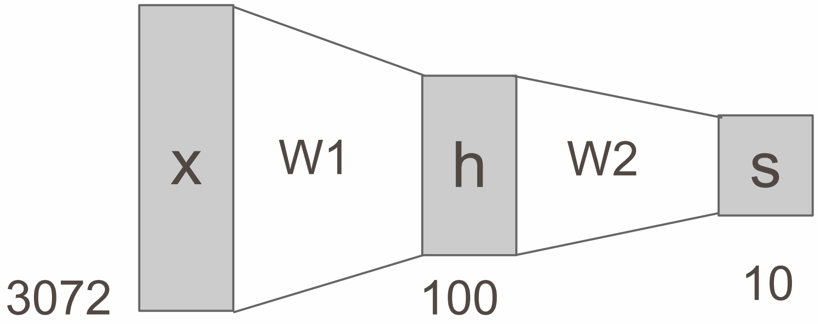


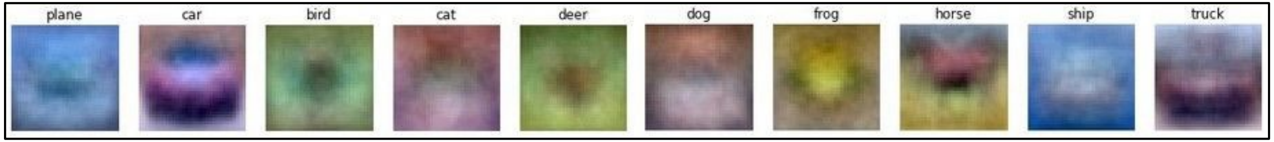
Neural Networks

- Linear score function:
- 2-layer Neural Network

$$f = Wx$$

$$f = W_2 \max(0, W_1 x)$$





Convolutional Neural Networks

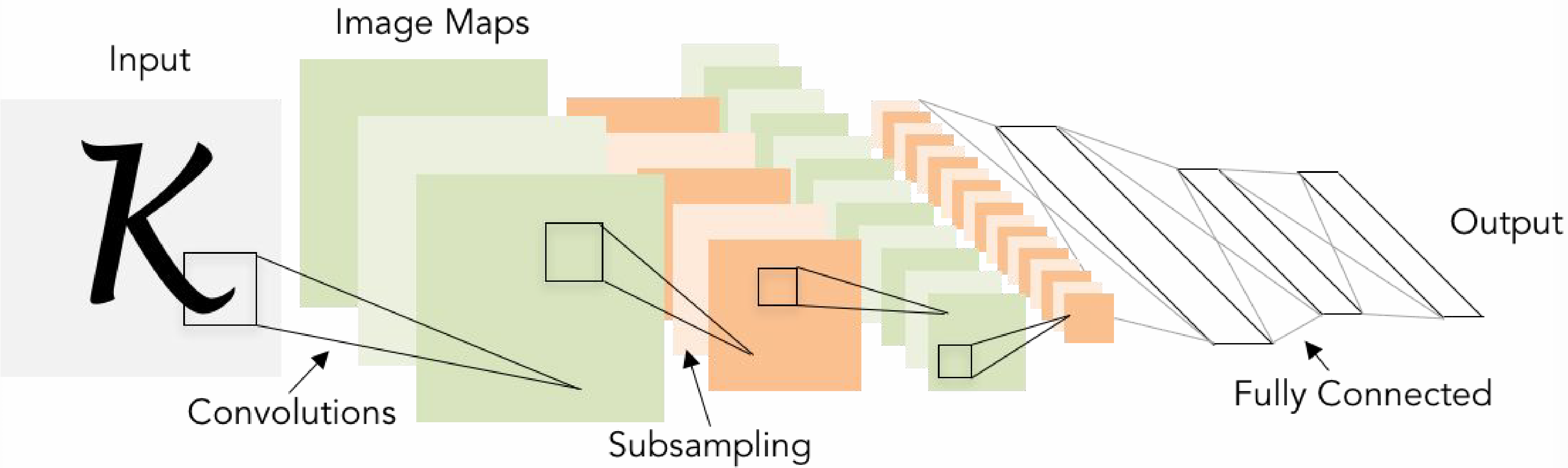
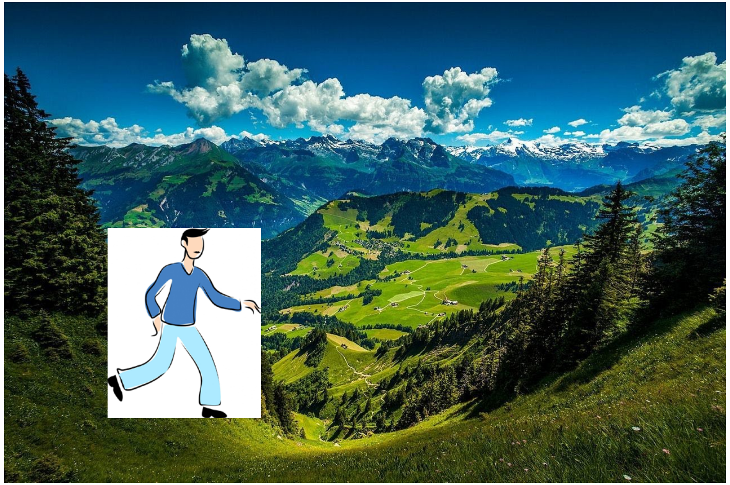
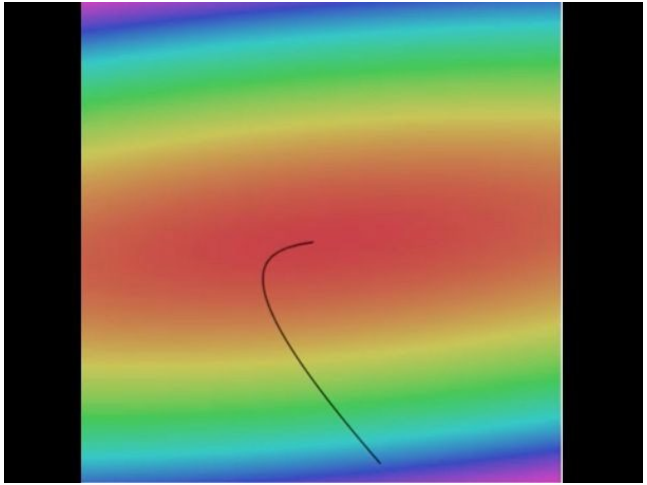


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Learning network parameters through optimization





```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Landscape image is CC0 1.0 public domain
Walking man image is CC0 1.0 public domain

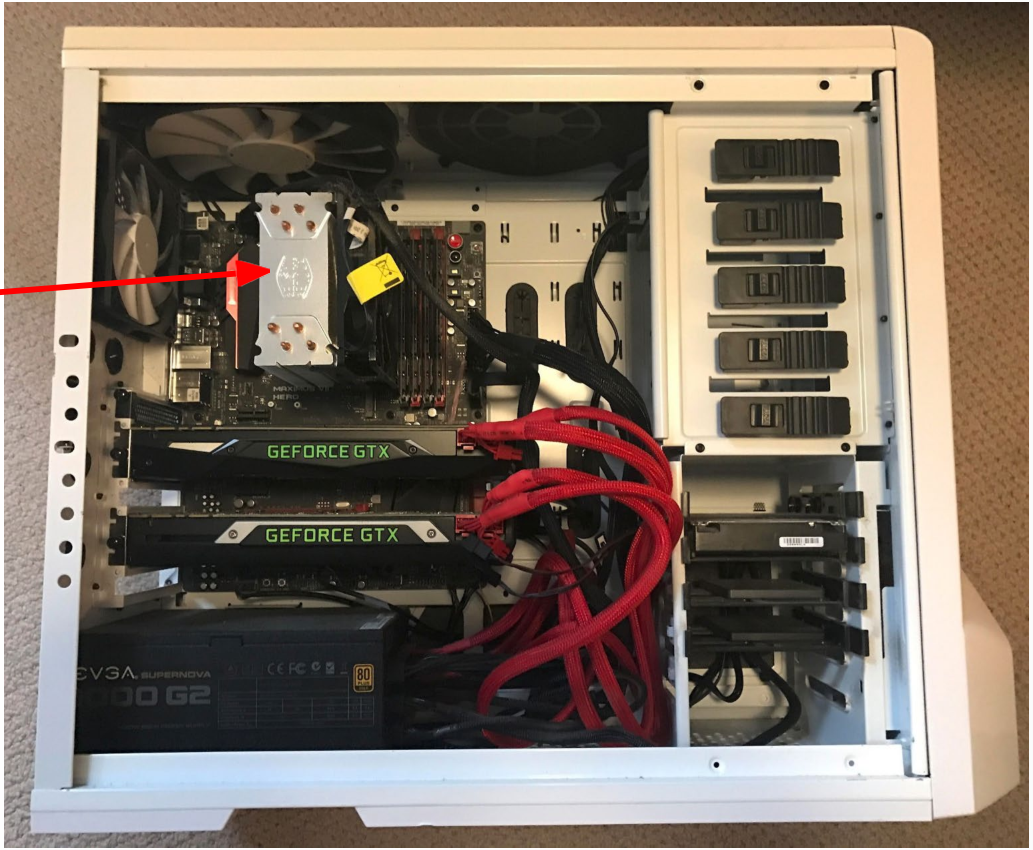
Today

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

DEEP LEARNING HARDWARE

Inside a computer

Spot the CPU!
(central processing unit)



Spot the GPUs!

(graphics processing unit)

This image is in the public domain

NVIDIA vs AMD

Google TPU

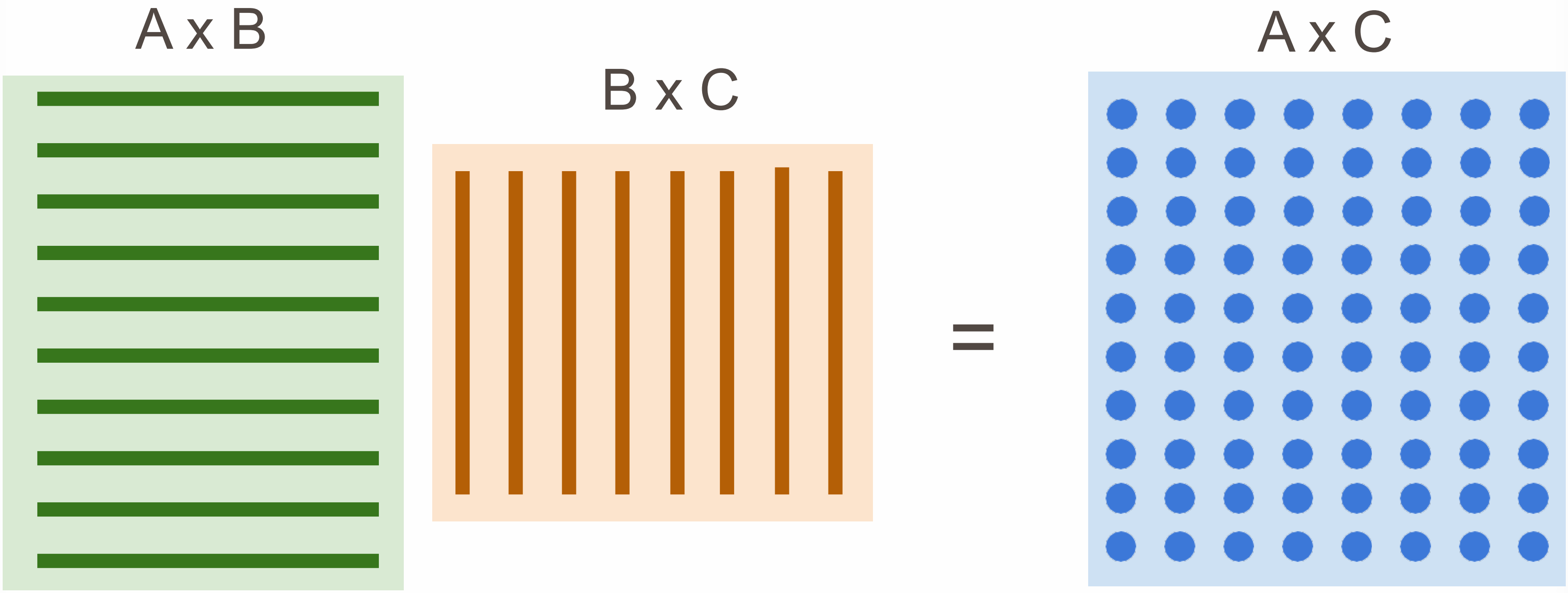
CPU vs GPU

	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 4090)	16,384	2.52 GHz	24 GB GDDR6X	$1999	~83 TFLOPs FP32
GPU (NVIDIA RTX 3090)	10,496	1.7 GHz	24 GB GDDR6	$1499	~35.6 TFLOPs FP32

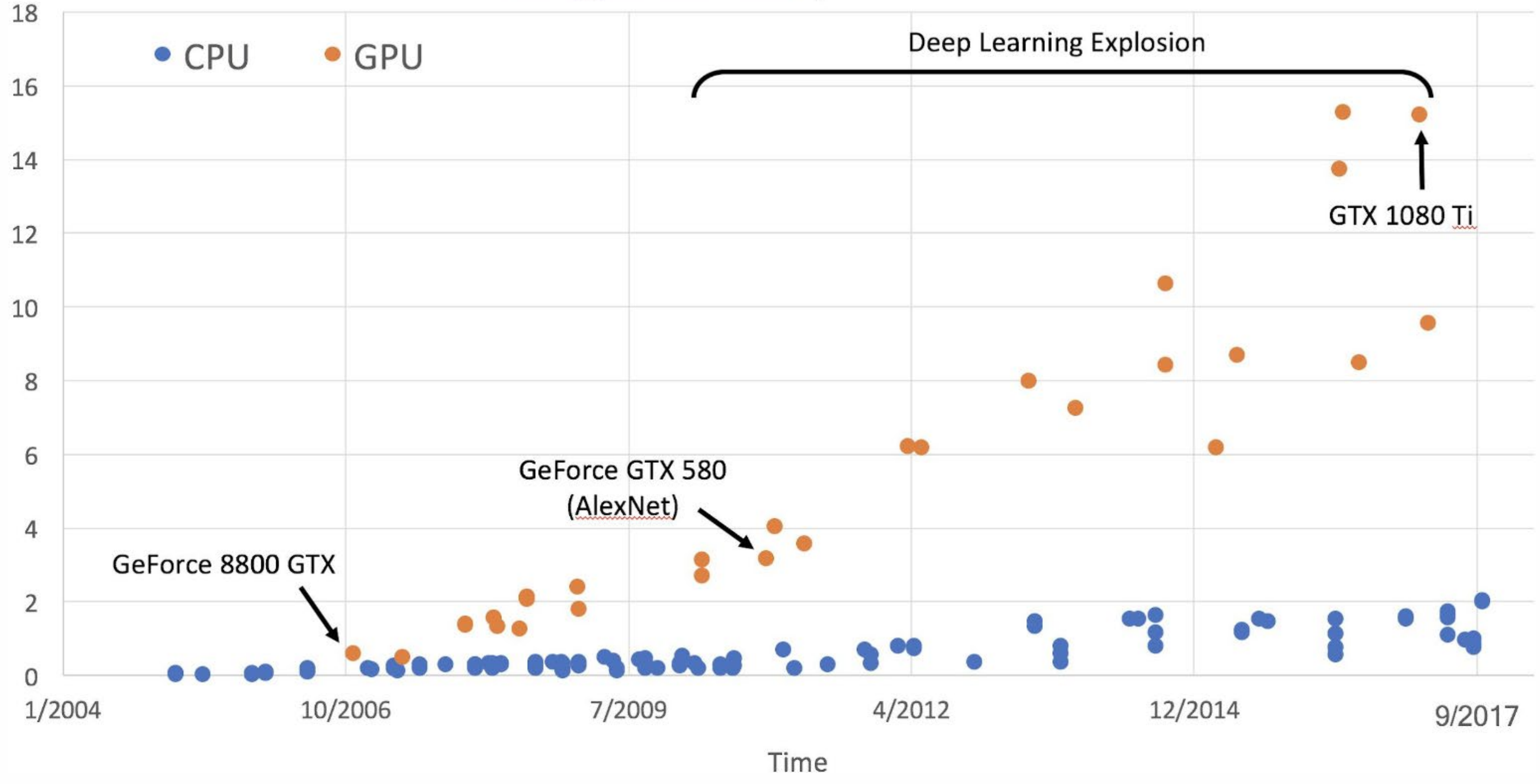
CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

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

Example: Matrix Multiplication

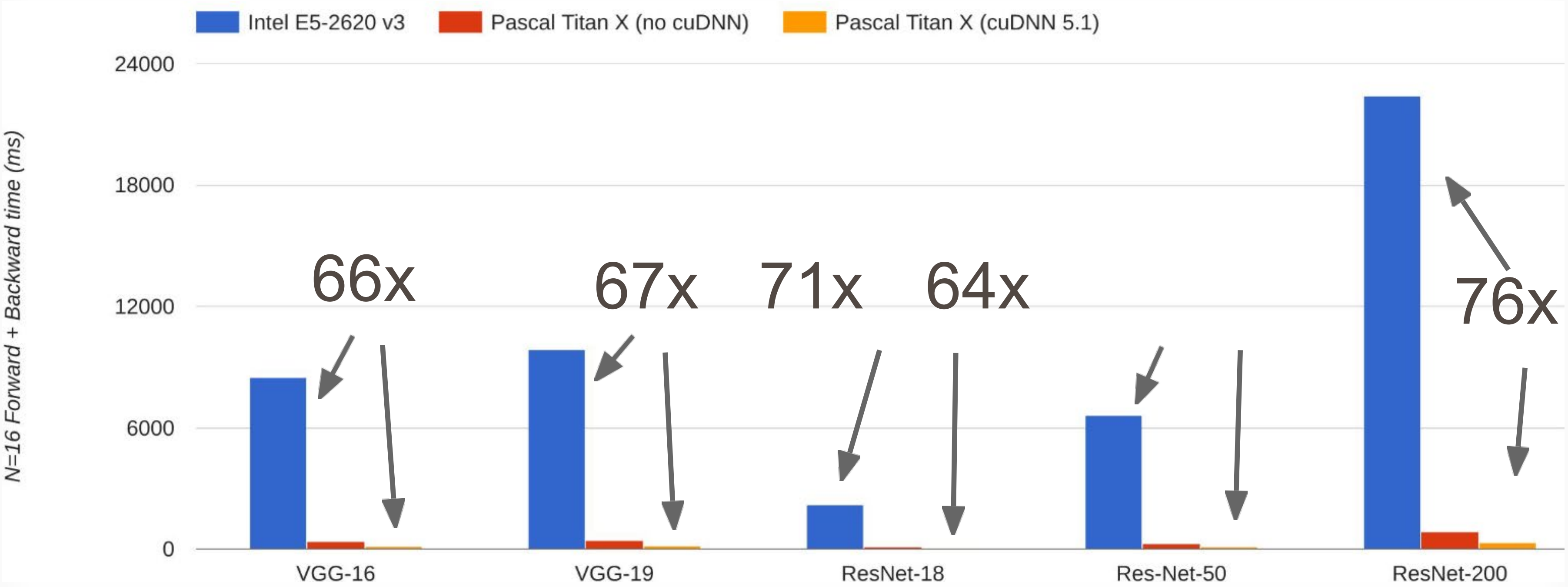


GigaFLOPs per Dollar



CPU vs GPU in practice

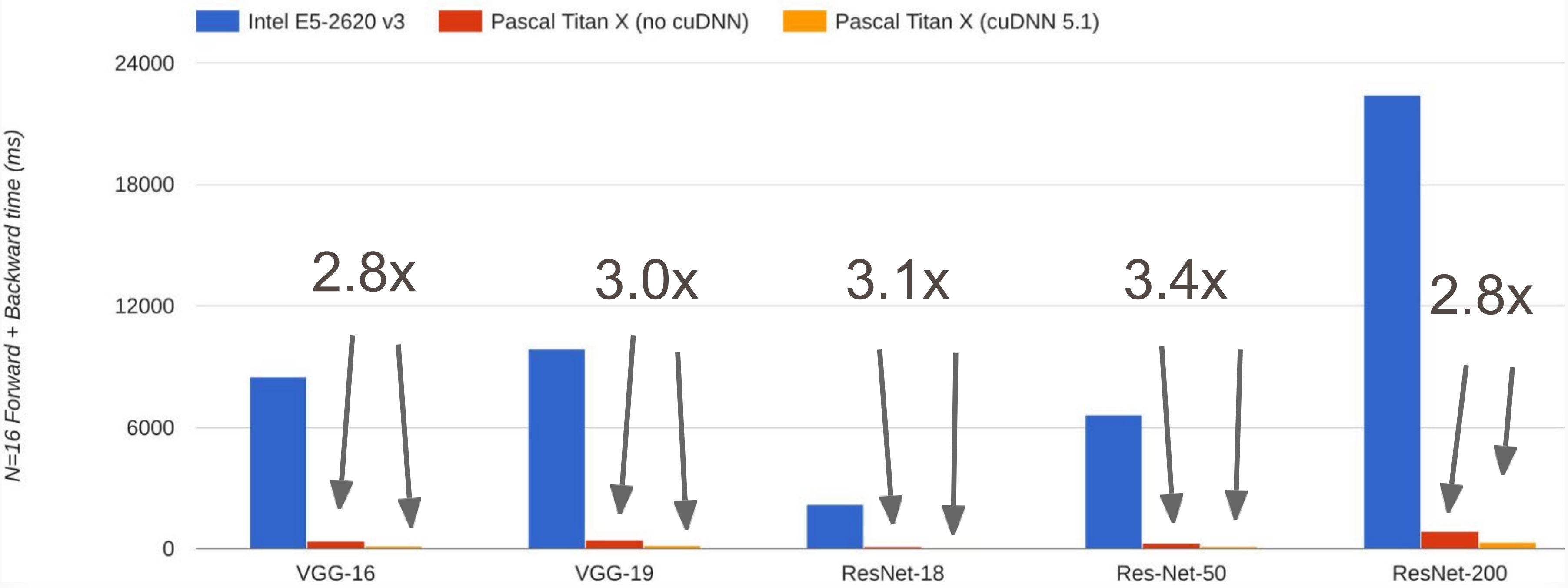
(CPU performance not well-optimized, a little unfair)



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

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
TPU NVIDIA TITAN V	5120 CUDA, 640 Tensor	1.5 GHz	12GB HBM2	$2999	~14 TFLOPs FP32 ~112 TFLOP FP16
TPU Google Cloud TPU	?	?	64 GB HBM	$4.50 per hour	~180 TFLOP

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

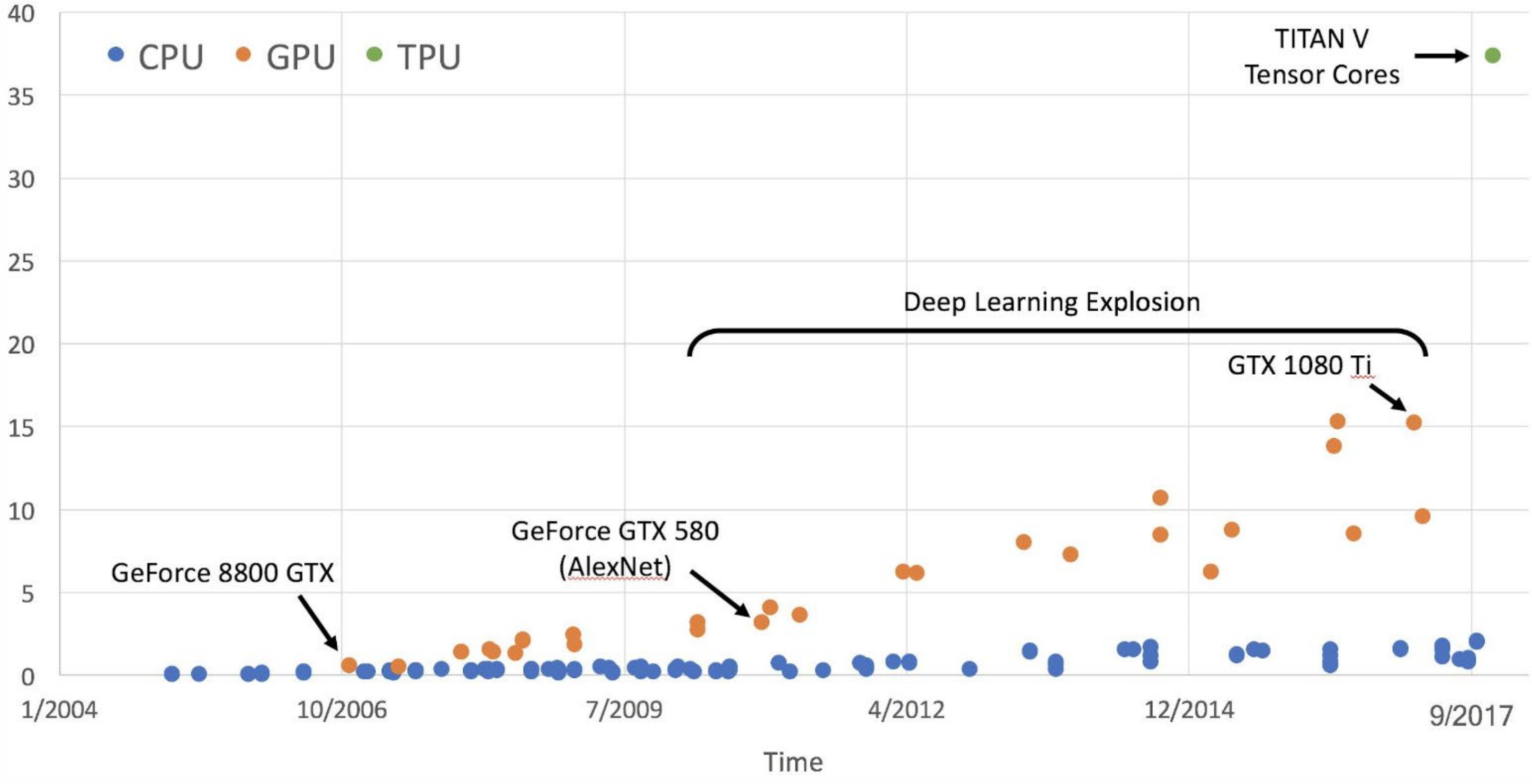
TPU: Specialized hardware for deep learning

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
TPU NVIDIA TITAN V	5120 CUDA, 640 Tensor	1.5 GHz	12GB HBM2	$2999	~14 TFLOPs FP32 ~112 TFLOP FP16
TPU Google Cloud TPU	?	?	64 GB HBM	$4.50 per hour	~180 TFLOP

NOTE: TITAN V isn't technically a "TPU" since that's a Google term, but both have hardware specialized for deep learning

GigaFLOPs per Dollar

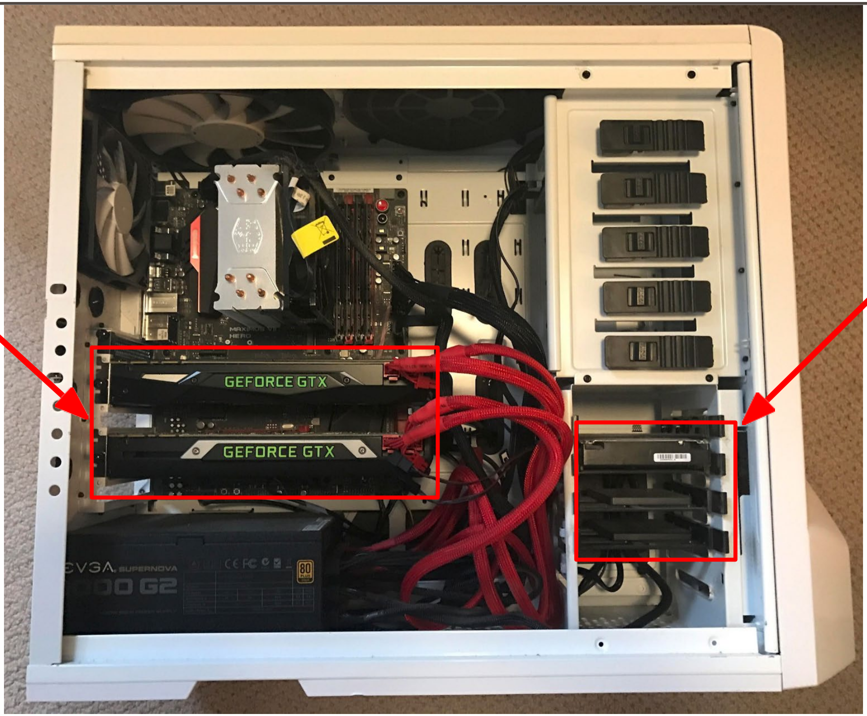


Programming GPUs

- 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
  - CUDA to AMD:
    - New project that automatically converts CUDA code to something that can run on AMD GPUs
- How to parallel programming:
  - https://developer.nvidia.com/udacity-cs344-intro-parallel-programming

CPU / GPU Communication

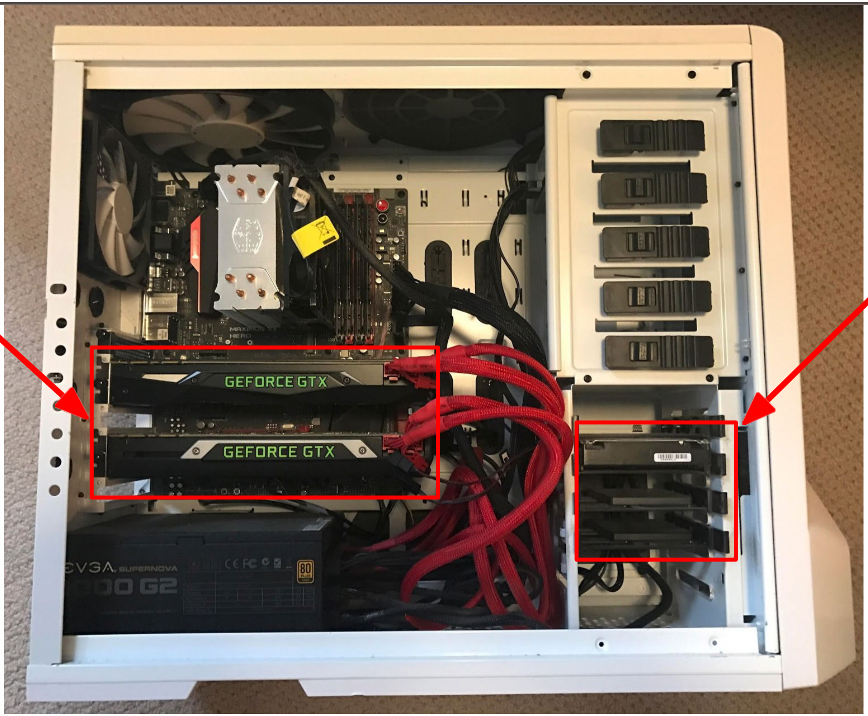
Model is here



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!

PaddlePaddle

(Baidu)

Chainer

Caffe

(UC Berkeley)

Caffe2

(Facebook)

MXNet

(Amazon)

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

CNTK

(Microsoft)

Torch

(NYU / Facebook)

PyTorch

(Facebook)

JAX

(Google)

Theano

(U Montreal)

TensorFlow

(Google)

And others...

A zoo of frameworks!

PaddlePaddle
(Baidu)

Chainer

Caffe
(UC Berkeley)

→ Caffe2
(Facebook)

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

CNTK
(Microsoft)

Torch
(NYU / Facebook)

→ PyTorch
(Facebook)

JAX
(Google)

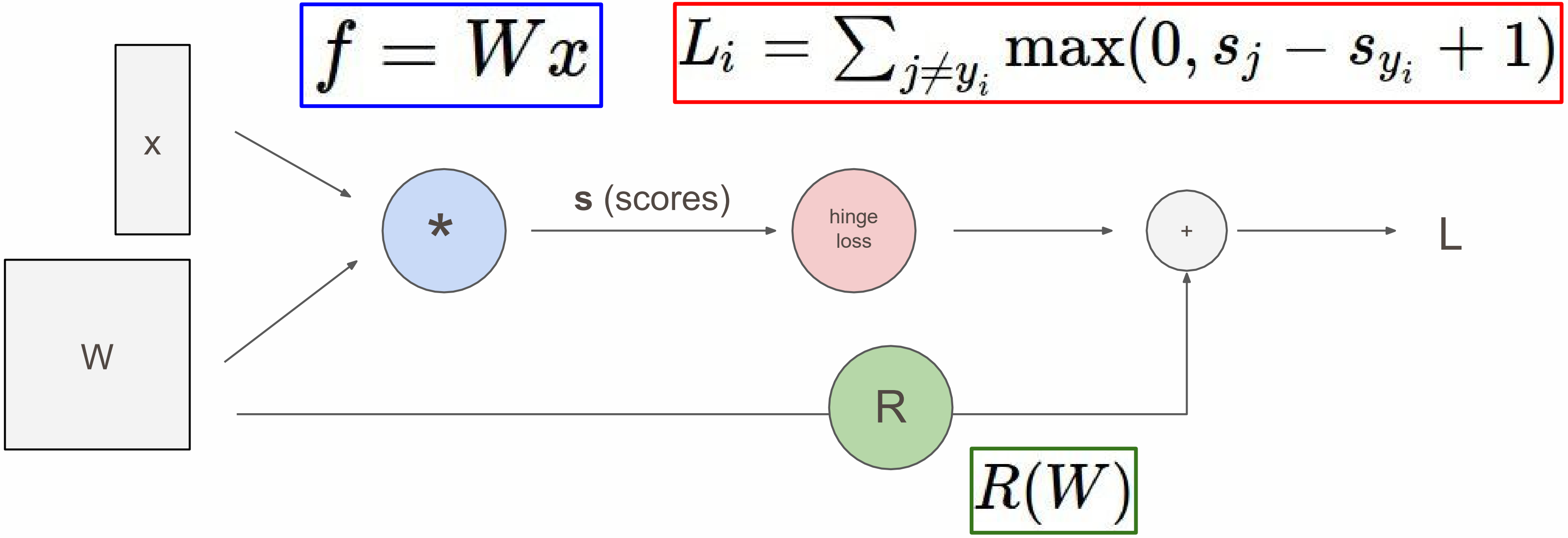
Theano
(U Montreal)

→ TensorFlow
(Google)

We'll focus on these

And others...

Recall: Computational Graphs



Recall: Computational Graphs

input image

weights

loss

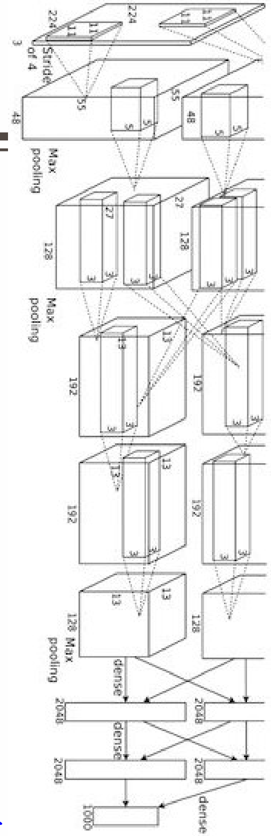


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Recall: Computational Graphs

input image

loss

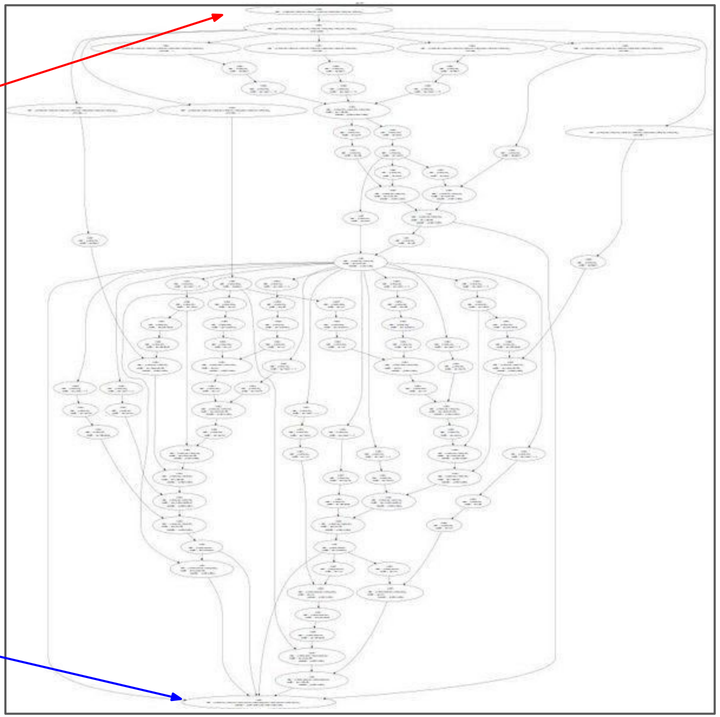


Figure reproduced with permission from a Twitter post by Andrej Karpathy.

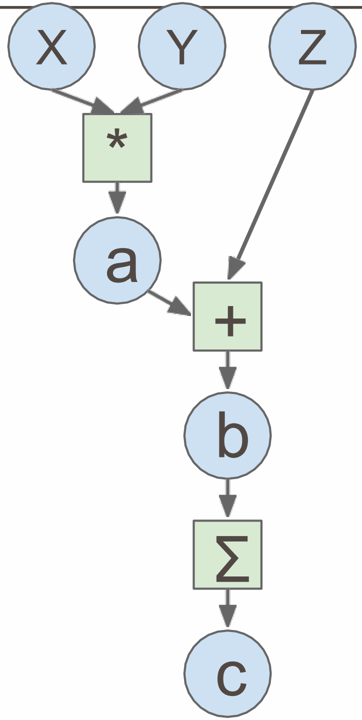
The point of deep learning frameworks

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

Computational Graphs

Numpy

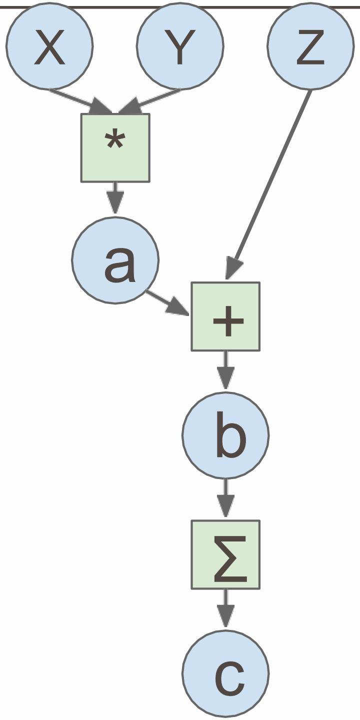
```
import numpy as np
np.random.seed(0)

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)
```



Computational Graphs

Numpy

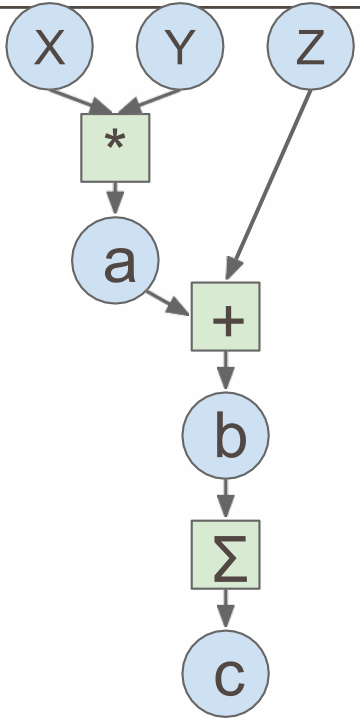
```
import numpy as np
np.random.seed(0)

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
```



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:

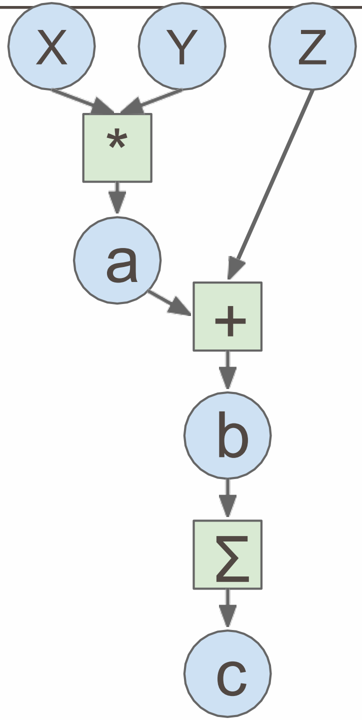
- Have to compute our own gradients
- Can’t run on GPU

Computational Graphs

Numpy

```python
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)
```

```python
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

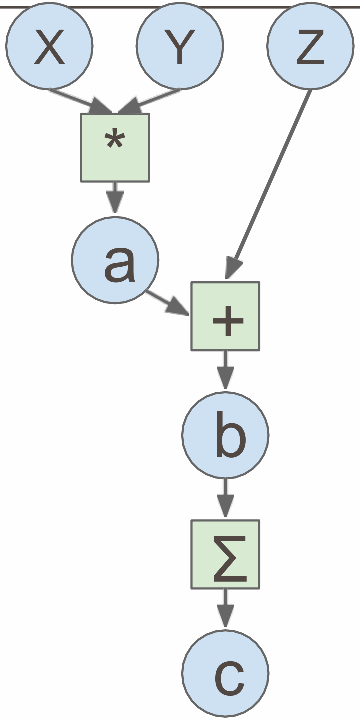
```python
import torch

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

```python
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

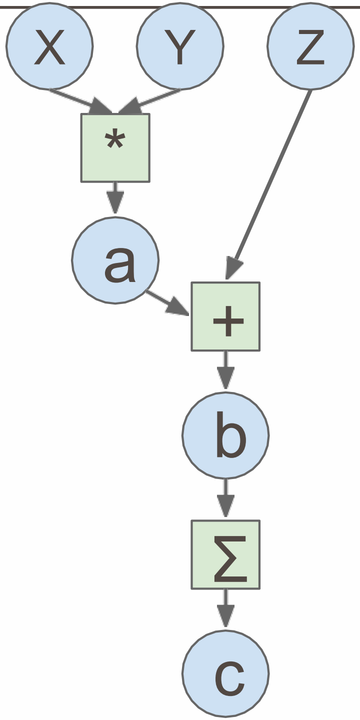
```python
import torch

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 DETAIL)

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 our code was tested on PyTorch version 1.10.1 with CUDA 12.0
  - (Released 2022)

- Be careful if you are looking at older PyTorch code!

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

PyTorch Tensors are just like numpy arrays, but they can run on GPU.

PyTorch Tensor API looks almost exactly like numpy!

Here we fit a two-layer net using PyTorch Tensors:

```python
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

```python
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

```python
import torch

device = torch.device('cpu')

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

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

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

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

Gradient descent step on weights

PyTorch: Tensors

To run on GPU, just use a different device!

```python
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

```python
import torch

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

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

    loss.backward()

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

PyTorch: Autograd

```
import torch

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

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

    loss.backward()

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

Compute gradient of loss with respect to w1 and w2

PyTorch: Autograd

```python
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_()
```

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

PyTorch: Autograd

```
import torch

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

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

    loss.backward()

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

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

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

```python
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)
```

```python
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_()
```

Can use our new autograd function in the forward pass

PyTorch: New Autograd Functions

```python
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

```python
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

```python
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

```python
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

```python
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

```python
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

```python
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

```python
import torch

class ParallelBlock(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(ParallelBlock, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)
        self.linear2 = torch.nn.Linear(D_in, D_out)
    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return (h1 * h2).clamp(min=0)

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

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

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

PyTorch: DataLoaders

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

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

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

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

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

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

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

PyTorch: DataLoaders

Iterate over loader to form minibatches

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

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

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

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

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

PyTorch: Pretrained Models

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

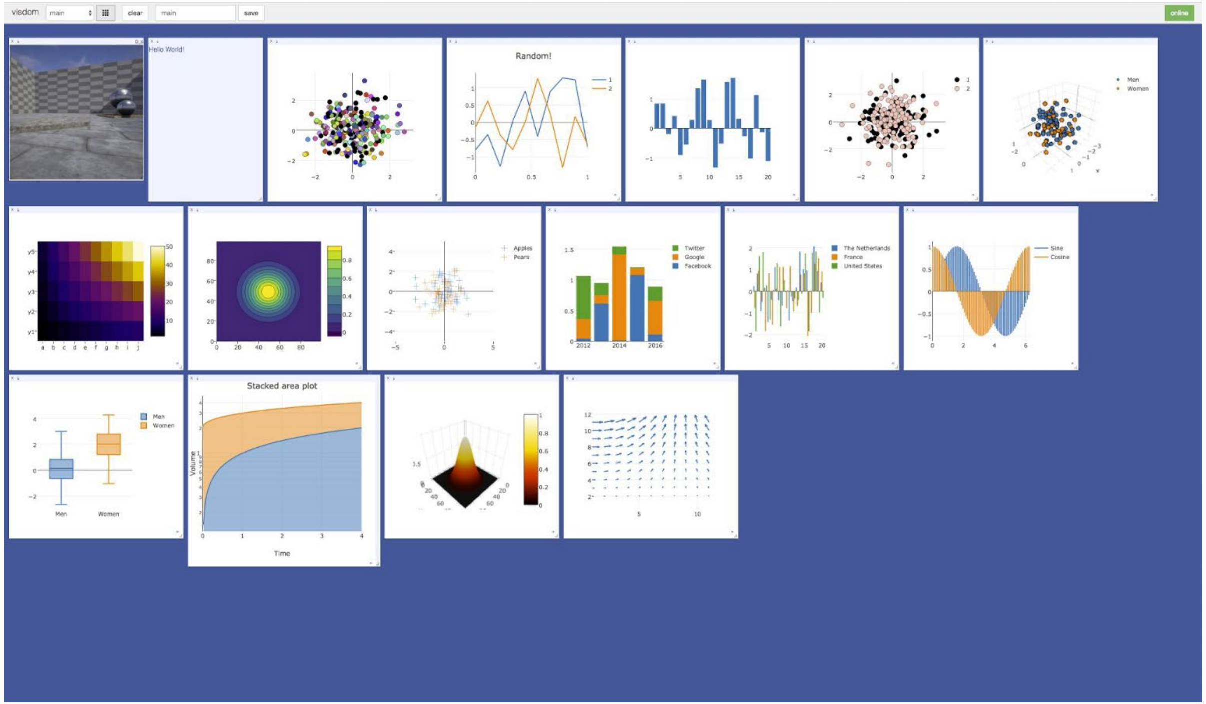
```
import torch
import torchvision

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

PyTorch: Visdom

Visualization tool: add logging to your code, then visualize in a browser

Can't visualize computational graph structure (yet?)

https://github.com/facebookresearch/visdom



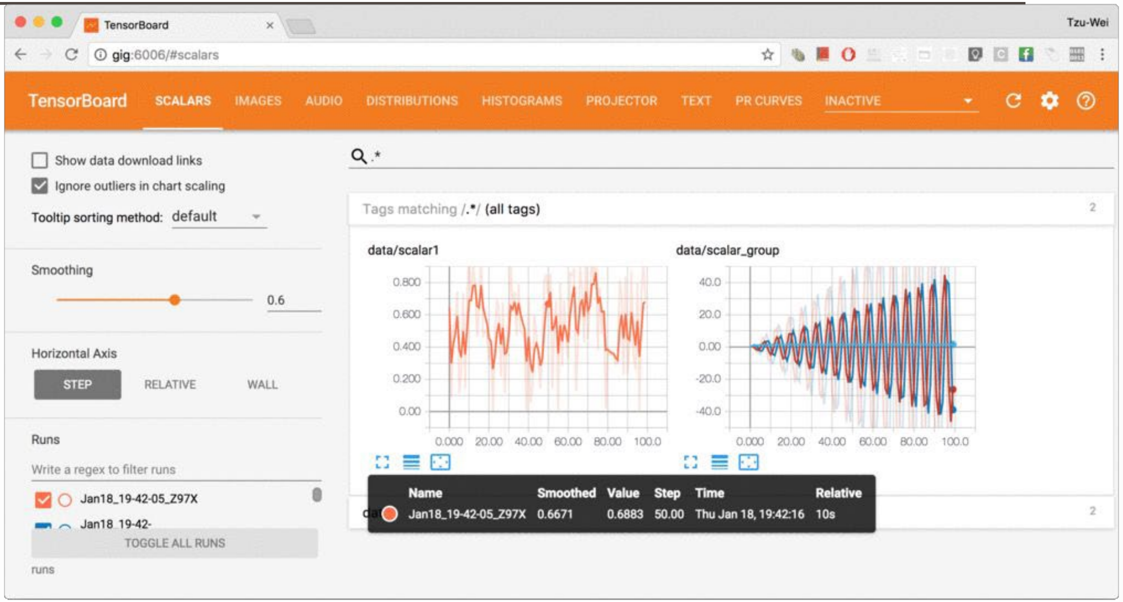
PyTorch: tensorboardX

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

pip install tensorboardx

Or

pip install tensorflow



https://github.com/lanpa/tensorboardX

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

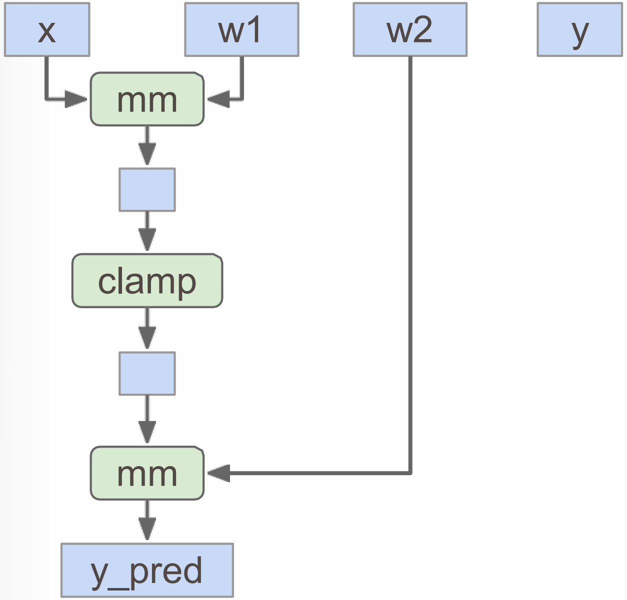
w2

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

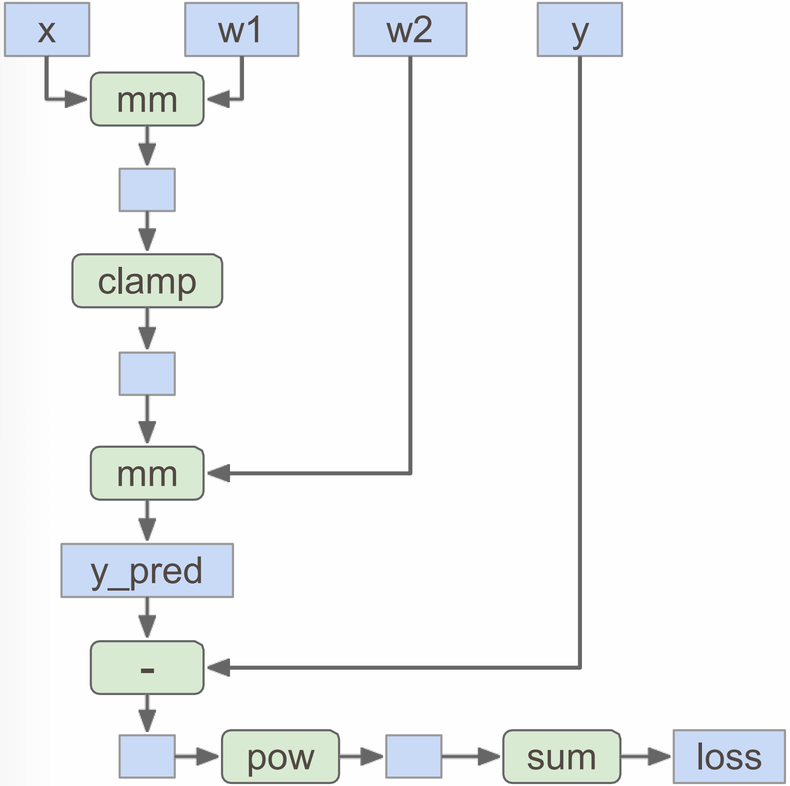
PyTorch: **Dynamic** Computation Graphs




```
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

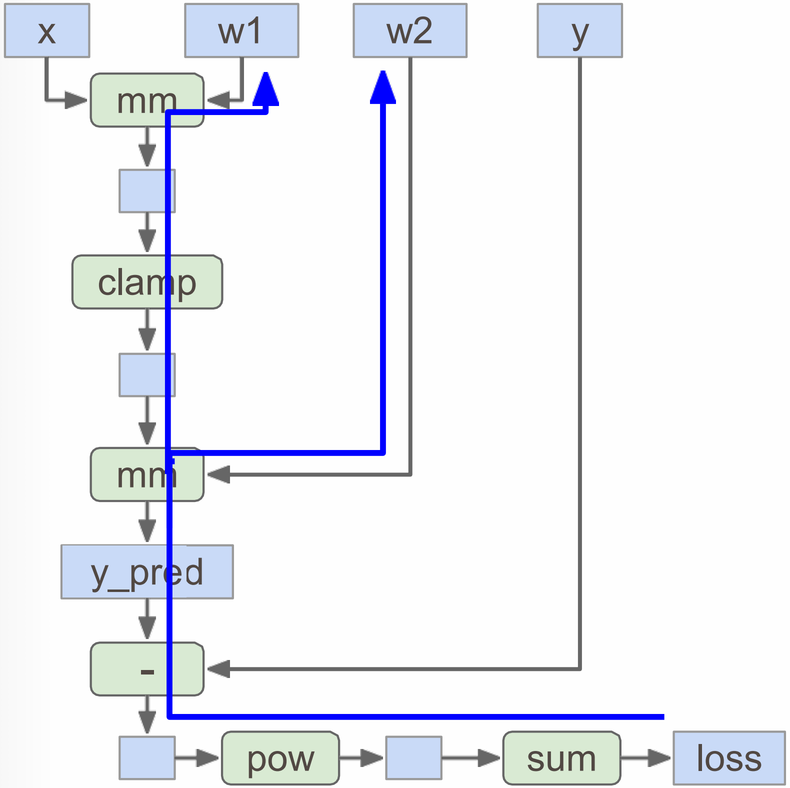
PyTorch: **Dynamic** Computation Graphs




```
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




```python
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()
```

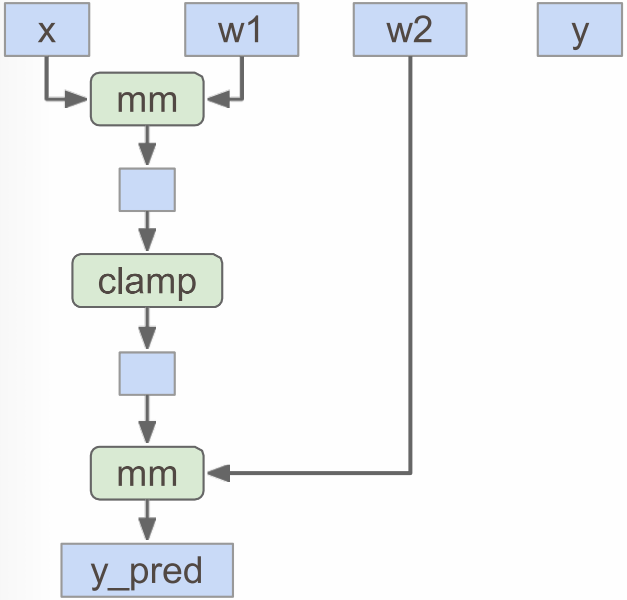
Search for path between loss and w1, w2
(for backprop) AND perform computation

PyTorch: **Dynamic** Computation Graphs


```python
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

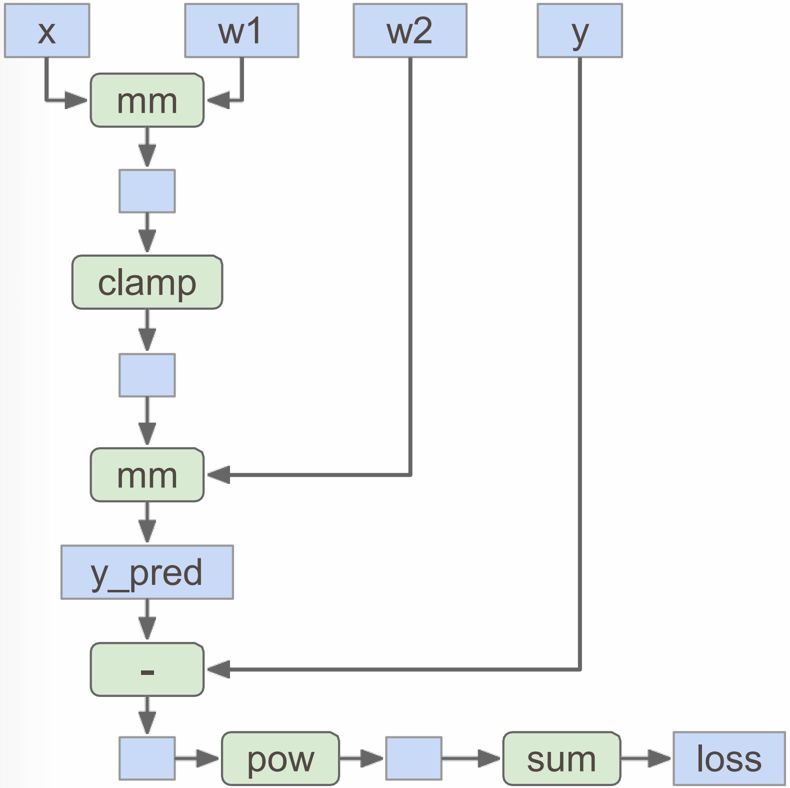
PyTorch: **Dynamic** Computation Graphs




```
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

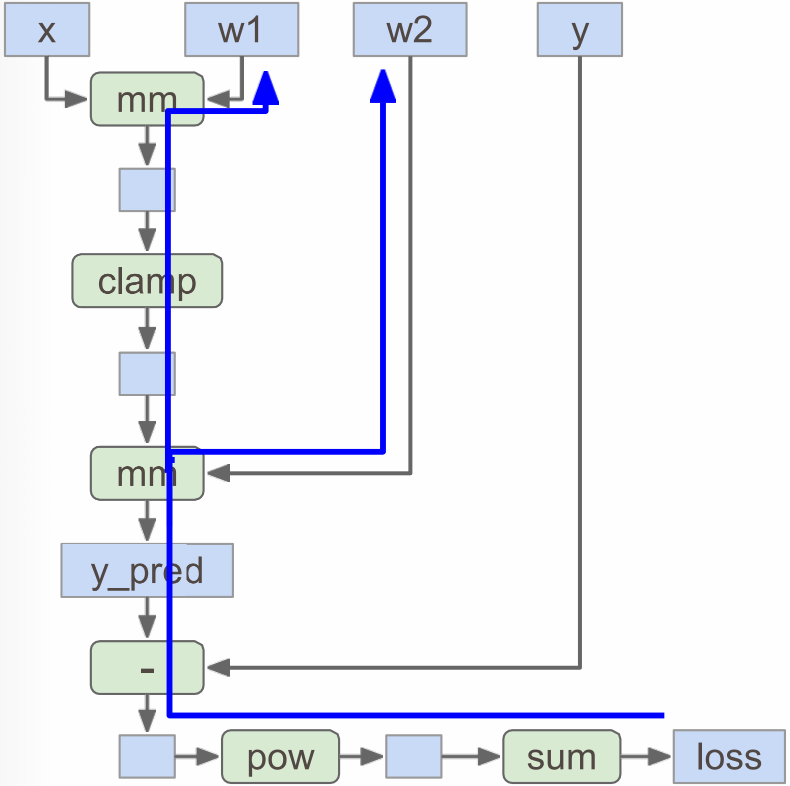
PyTorch: **Dynamic** Computation Graphs




```
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




```python
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.

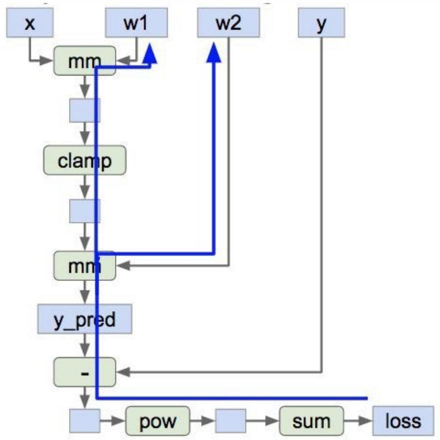
Seems inefficient, especially if we are building the same graph over and over again...

```
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 (1.13 latest)

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

2.12 Alpha (2023 early)

Default dynamic graph, optionally static graph.

We use 2.8.0 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 1.15

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 1.15

TensorFlow: 2.0 vs. pre-2.0

```python
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

```python
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 1.15

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])
```

TensorFlow: Neural Net

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

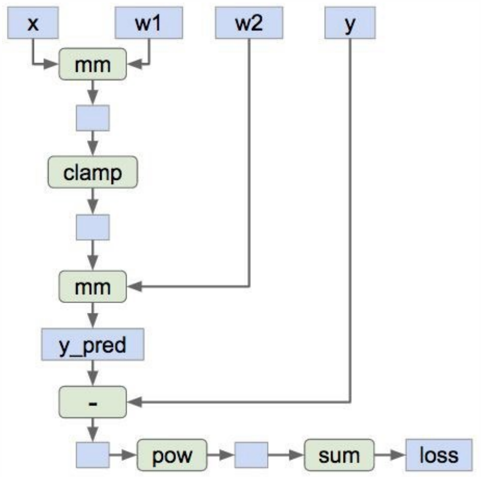
```
N, D, H = 64, 1000, 100

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

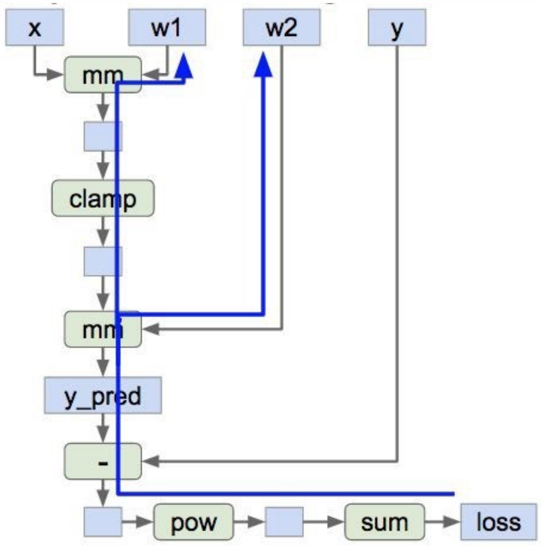
```
N, D, H = 64, 1000, 100

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))
gradients = 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



Backward pass

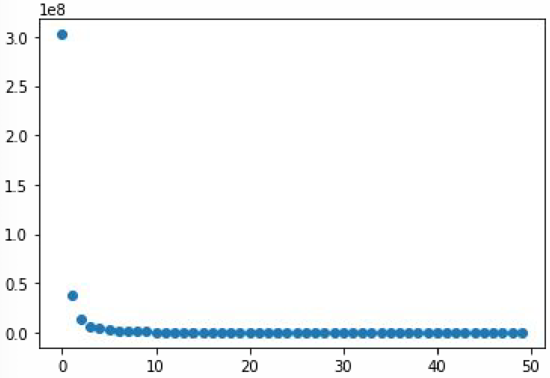
```python
N, D, H = 64, 1000, 100

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

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 graph 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

```
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)
```

Keras can handle the training loop for you!

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

```python
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)
  return y_pred, loss

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

```
static graph: 0.14495624600000667
dynamic graph: 0.02945919699999422
```

@tf.function: compile static 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)
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)
  return y_pred, loss
```

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

```
static graph: 0.14495624600000667
dynamic graph: 0.02945919699999422
```

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

@tf.function: compile static graph

There are some caveats in defining control loops (for, if) with @tf.function.

```
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)
  return y_pred, loss

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

```
static graph: 0.14495624600000667
dynamic graph: 0.02945919699999422
```

TensorFlow: More on Eager Mode

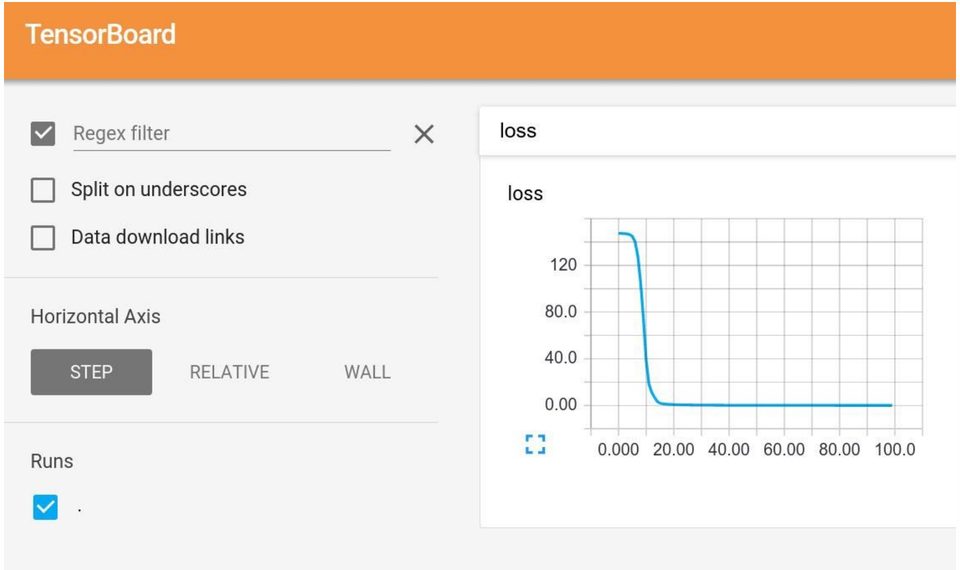
- Eager mode: (https://www.tensorflow.org/guide/eager)
- tf.function: (https://www.tensorflow.org/alpha/tutorials/eager/tf_function)

TensorFlow: Pretrained Models

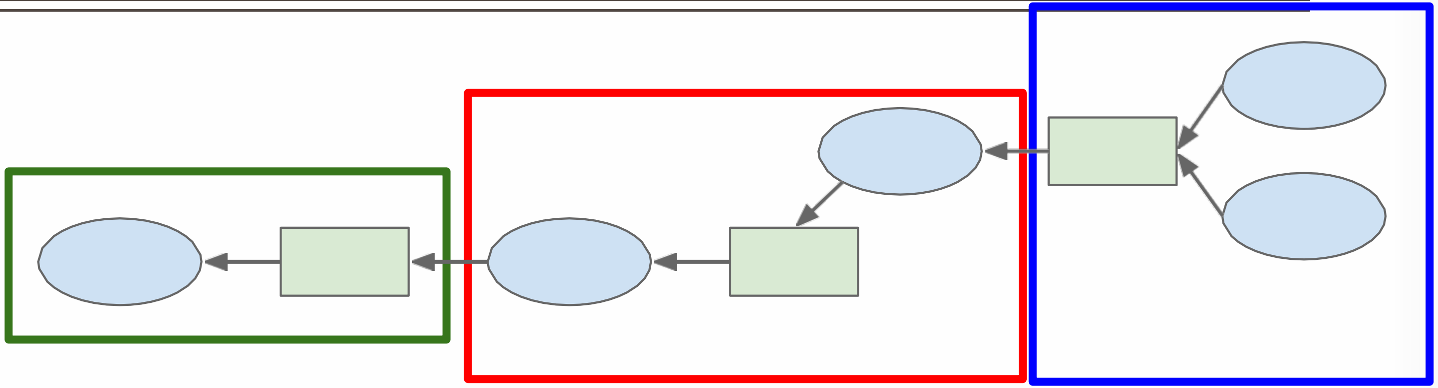
- tf.keras: (https://www.tensorflow.org/api_docs/python/tf/keras/applications)
- TF-Slim: (https://github.com/tensorflow/models/tree/master/research/slim)

TensorFlow: Tensorboard

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



TensorFlow: Distributed Version



Split one graph over multiple machines!

https://www.tensorflow.org/deploy/distributed

TensorFlow: Tensor Processing Units

Google Cloud TPU
= 180 TFLOPs of compute!

TensorFlow: Tensor Processing Units

Google Cloud TPU
= 180 TFLOPs of compute!

NVIDIA Tesla V100
= 125 TFLOPs of compute

TensorFlow: Tensor Processing Units

Google Cloud TPU
= 180 TFLOPs of compute!

NVIDIA Tesla V100
= 125 TFLOPs of compute

NVIDIA Tesla P100 = 11 TFLOPs of compute
GTX 580 = 0.2 TFLOPs

TensorFlow: Tensor Processing Units

Google Cloud TPU
= 180 TFLOPs of compute!

Google Cloud TPU Pod
= 64 Cloud TPUs
= 11.5 PFLOPs of compute!

https://www.tensorflow.org/versions/master/programmers_guide/using_tpu

TensorFlow: Tensor Processing Units

Edge TPU = 64 GFLOPs (16 bit)

https://cloud.google.com/edge-tpu/

Static vs Dynamic Graphs

TensorFlow (tf.function): Build graph once, then run many times (**static)**

```
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))
```

Compile python code into static graph

Run each iteration

PyTorch: Each forward pass defines a new graph (**dynamic)**

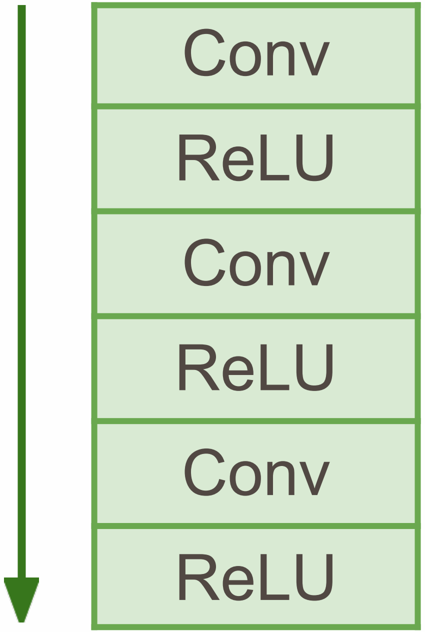
```
import torch

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()
```

New graph each iteration

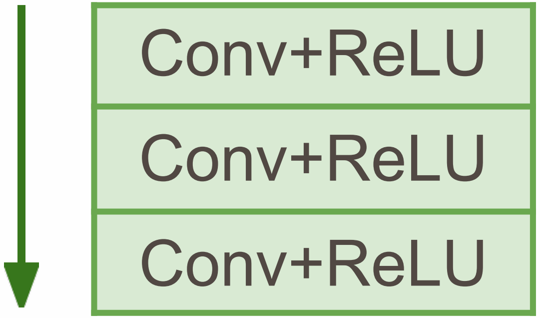
Static vs Dynamic: Optimization

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

The graph you wrote



Equivalent graph with **fused operations**



Static vs Dynamic: Serialization

Static

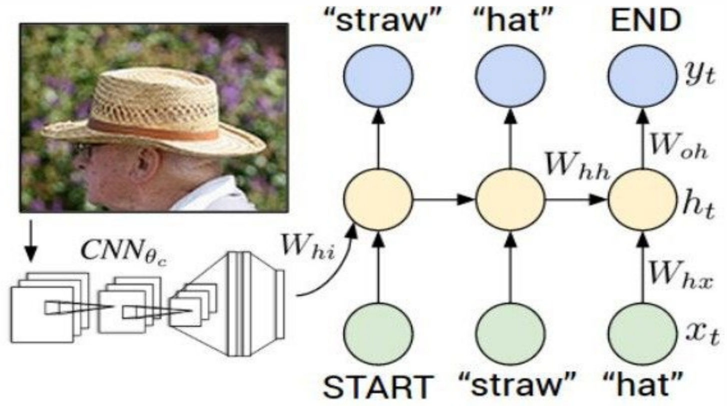
Once graph is built, can **serialize** it and run it without the code that built the graph!

Dynamic

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

Dynamic Graph Applications

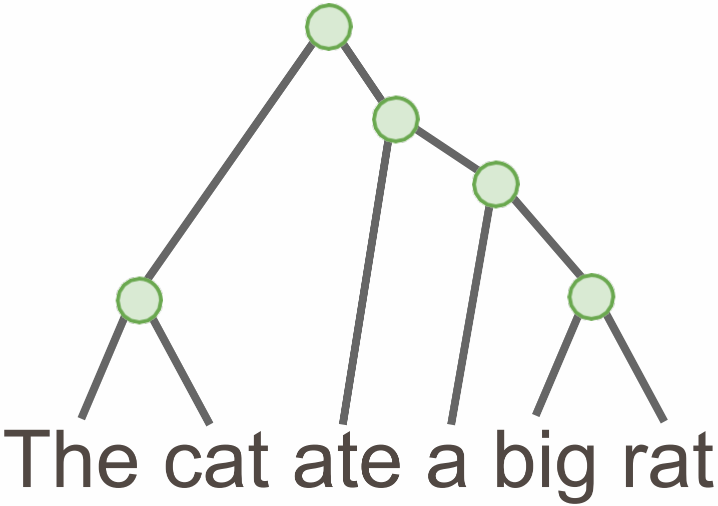
-Recurrent networks



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

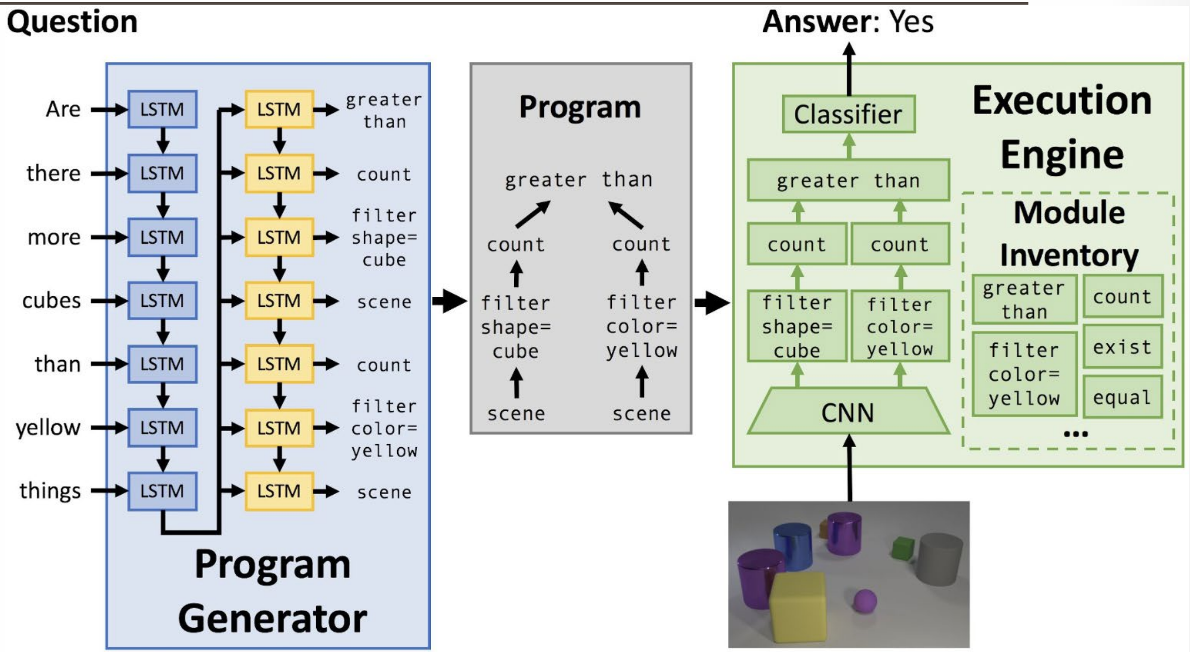
Dynamic Graph Applications

- 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

Dynamic Graph Applications

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

PyTorch vs TensorFlow, Static vs Dynamic

- PyTorch
- Dynamic Graphs

- TensorFlow
- Pre-2.0: Default Static Graph 2.0+: Default Dynamic Graph
- 2.x: Eagar mode

Static PyTorch: Caffe2 https://caffe2.ai/

- Deep learning framework developed by Facebook
- Static graphs, somewhat similar to TensorFlow
- Core written in C++
- Nice Python interface
- Can train model in Python, then serialize and deploy without Python
- Works on iOS / Android, etc

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: 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

```python
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)
```

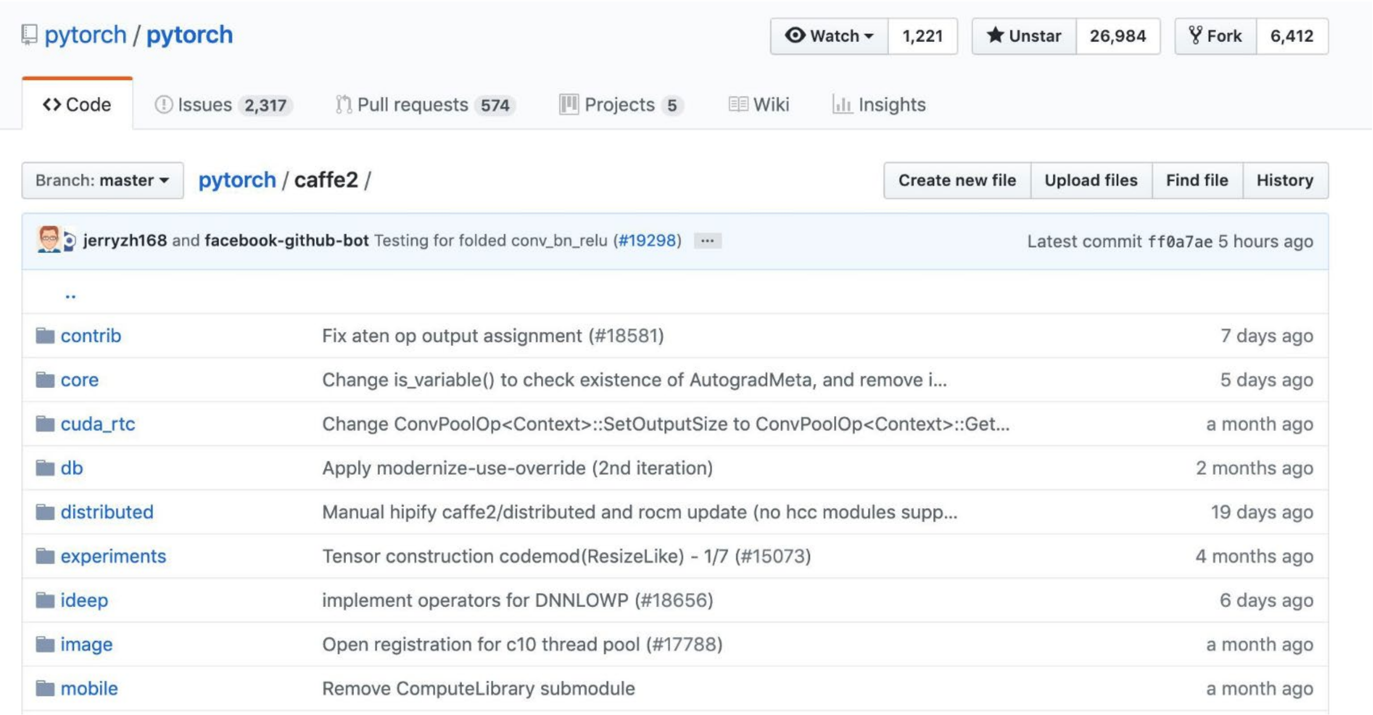
Static PyTorch: ONNX Support

```
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);
}
```

```python
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



PyTorch vs TensorFlow, Static vs Dynamic

PyTorch
Dynamic Graphs
Static: ONNX, Caffe2

TensorFlow
Dynamic: Eager
Static: @tf.function

My Advice:

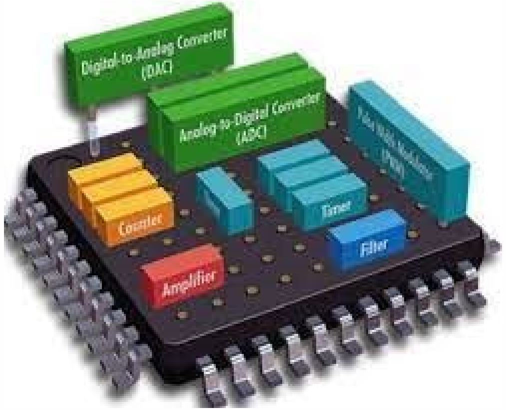
- PyTorch is my personal favorite. Clean API, native dynamic graphs make it very easy to develop and debug. Can build model in PyTorch then export to Caffe2 with ONNX for production / mobile

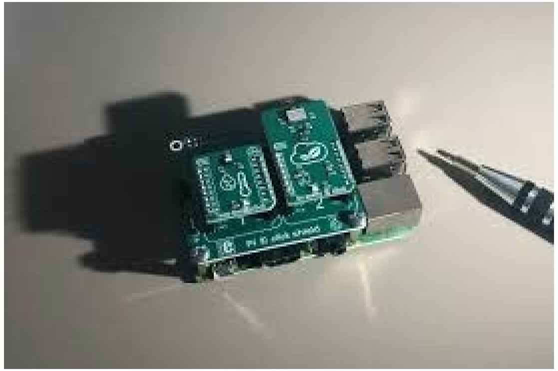
- TensorFlow is a safe bet for most projects. Syntax became a lot more intuitive after 2.x. Not perfect but has huge community and wide usage. Can use same framework for research and production. Probably use a high-level framework. Only choice if you want to run on TPUs.

MODEL COMPRESSION

Low-power required? Mobile Device?

DL looks great. How about the low-power devices?





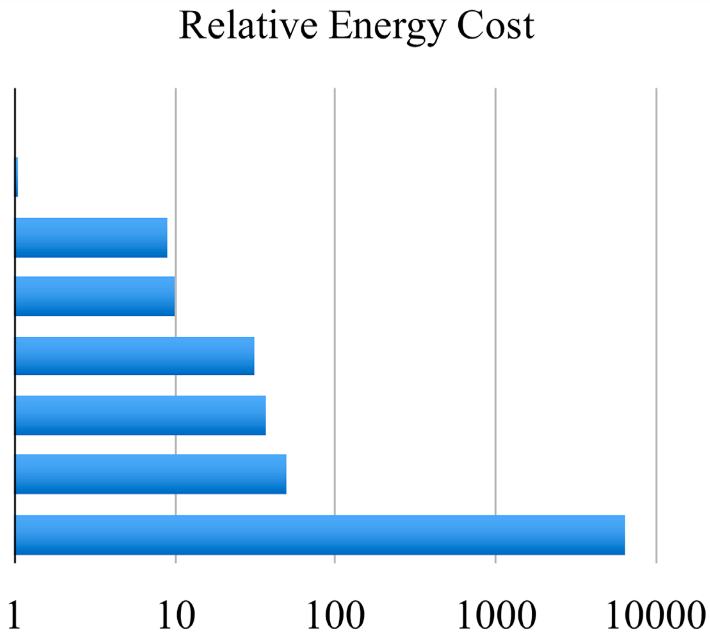
Why model compression?

- Deep Neural Networks are BIG … and getting BIGGER
  - e.g. AlexNet (240 MB), VGG-16 (520 MB), GPT-3.5 (~5215 GB)

- Too big to store on-chip SRAM and DRAM accesses use a lot of energy
- Not suitable for low-power mobile/embedded systems
- Solution: Deep Compression

Why smaller models?

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	640	6400



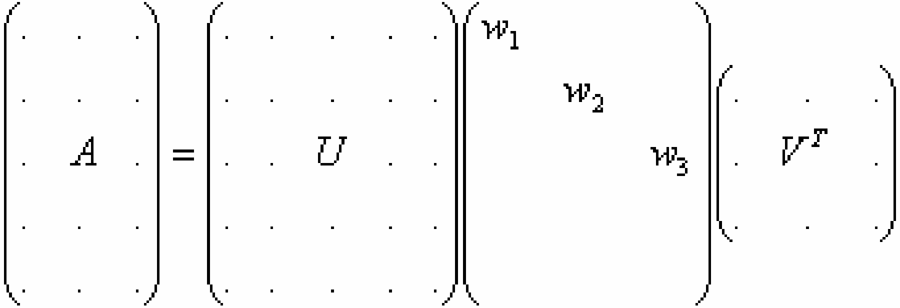
Source:
http://isca2016.eecs.umich.edu/wp-content/uploads/2016/07/4A-1.pdf

Methods to Model Compression

- Technique to reduce size of neural networks without losing accuracy
- **Matrix factorization**
- **Pruning to Reduce Number of Weights**
- **Quantization to Reduce Bits per Weight**

Fully Connected Layers: Singular Value Decomposition

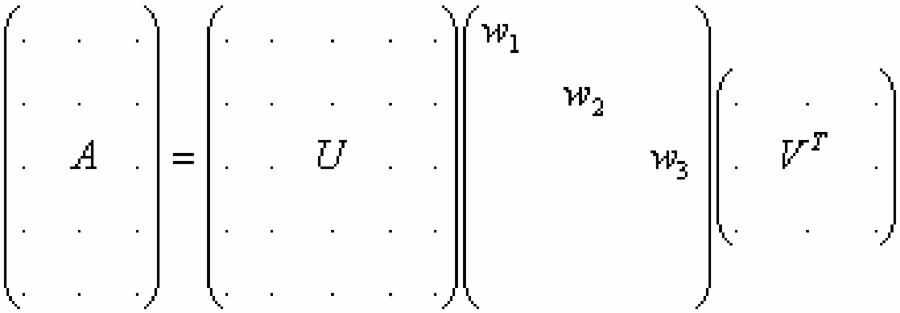
- Most weights are in the fully connected layers (according to Denton et al.)
- $W = USV^{\top}$
  - $W \in \mathbb{R}^{m \times k}, U \in \mathbb{R}^{m \times m}, S \in \mathbb{R}^{m \times k}, V^{\top} \in \mathbb{R}^{k \times k}$
- S is diagonal, decreasing magnitudes along the diagonal



http://www.alglib.net/matrixops/general/i/svd1.gif

Singular Value Decomposition

- By only keeping the t singular values with largest magnitude:
- $\widetilde{W} = \widetilde{U}\tilde{S}\widetilde{V}^{\top}$
  - $\widetilde{W} \in \mathbb{R}^{m\times k}, \tilde{U} \in \mathbb{R}^{m\times t}, \tilde{S} \in \mathbb{R}^{t\times t}, \tilde{V}^{\top} \in \mathbb{R}^{t\times k}$
- $Rank(\widetilde{W}) = t$



http://www.alglib.net/matrixops/general/i/svd1.gif

SVD: Compression

- $W = USV^{\top}, W \in \mathbb{R}^{m\times k}, U \in \mathbb{R}^{m\times m}, S \in \mathbb{R}^{m\times k}, V^{\top} \in \mathbb{R}^{k\times k}$
- $\widetilde{W} = \widetilde{U}\tilde{S}\widetilde{V}^{\top}, \widetilde{W} \in R^{m\times k}, \widetilde{U} \in R^{m\times t}, \tilde{S} \in R^{t\times t}, \widetilde{V}^{\top} \in R^{t\times k}$

- Storage for W: $O(mk)$
- Storage for $\widetilde{W}$: $O(mt + t + tk)$
- Compression Rate: $O\left(\frac{mk}{t(m+k+1)}\right)$
- Theoretical error: $\left\|A\widetilde{W} - AW\right\|_F \leq s_{t+1}\|A\|_F$

Gong, Yunchao, et al. "Compressing deep convolutional networks using vector quantization." *arXiv preprint arXiv:1412.6115* (2014).

SVD: Compression Results

- Trained on ImageNet 2012 database, then compressed
- 5 convolutional layers, 3 fully connected layers, softmax output layer

Approximation method	Number of parameters	Approximation hyperparameters	Reduction in weights	Increase in error
Standard FC	NM			
FC layer 1: Matrix SVD	$NK + KM$	$K = 250$ $K = 950$	$13.4\times$ $3.5\times$	0.8394% 0.09%
FC layer 2: Matrix SVD	$NK + KM$	$K = 350$ $K = 650$	$5.8\times$ $3.14\times$	0.19% 0.06%
FC layer 3: Matrix SVD	$NK + KM$	$K = 250$ $K = 850$	$8.1\times$ $2.4\times$	0.67% 0.02%

K refers to rank of approximation, t in the previous slides.

Denton, Emily L., et al. "Exploiting linear structure within convolutional networks for efficient evaluation." *Advances in Neural Information Processing Systems*. 2014.

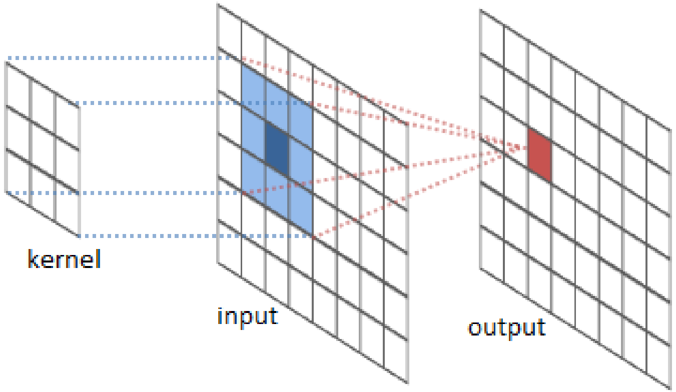
SVD: Side Benefits

- Reduced memory footprint
  - Reduced in the dense layers by 5-13x
- Speedup: $A\widetilde{W}, A \in \mathbb{R}^{n\times m}$, computed in $O(nmt + nt^2 + ntk)$ instead of $O(nmk)$
  - Speedup factor is $O\left(\frac{mk}{t(m+t+k)}\right)$
- Regularization
  - "Low-rank projections effectively decrease number of learnable parameters, suggesting that they might improve generalization ability."
  - Paper applies SVD after training

Denton, Emily L., et al. "Exploiting linear structure within convolutional networks for efficient evaluation." *Advances in Neural Information Processing Systems*. 2014.

Convolutions: Matrix Multiplication

Most time is spent in the convolutional layers

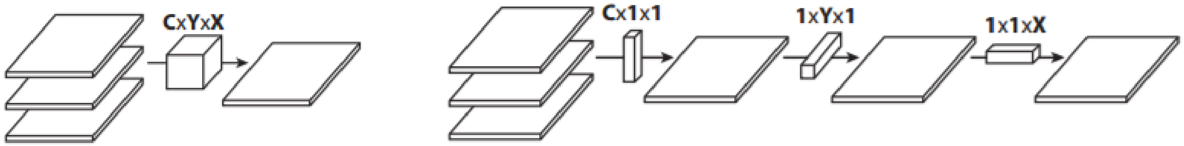


$$F(x, y) = I * W$$

http://stackoverflow.com/questions/15356153/how-do-convolution-matrices-work

Flattened Convolutions

- Replace $c \times y \times x$ convolutions with $c \times 1 \times 1$, $1 \times y \times 1$, and $1 \times 1 \times x$ convolutions



(a) 3D convolution   (b) 1D convolutions over different directions

Jin, Jonghoon, Aysegul Dundar, and Eugenio Culurciello. "Flattened convolutional neural networks for feedforward acceleration." *arXiv preprint arXiv:1412.5474* (2014).

Flattened Convolutions

$$\hat{F}(x,y) = I * \hat{W} = \sum_{x'=1}^{X}\left(\sum_{y'=1}^{Y}\left(\sum_{c=1}^{C} I(c, x-x', y-y')\alpha(c)\right)\beta(y')\right)\gamma(x')$$

$\alpha \in \mathbb{R}^C, \beta \in \mathbb{R}^Y, \gamma \in \mathbb{R}^X$

- Compression and Speedup:
  - Parameter reduction: $O(XYC)$ to $O(X+Y+C)$
  - Operation reduction: $O(mnCXY)$ to $O\big(mn(C+X+Y)\big)$ (where $W_f \in \mathbb{R}^{m\times n}$)

Jin, Jonghoon, Aysegul Dundar, and Eugenio Culurciello. "Flattened convolutional neural networks for feedforward acceleration." *arXiv preprint arXiv:1412.5474* (2014).

Flattening = MF

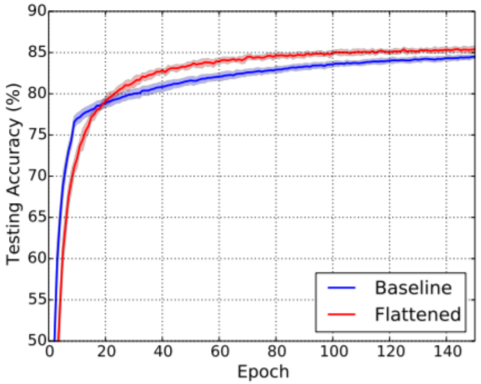
$$\hat{F}(x,y) = \sum_{x=1}^{X}\sum_{y'=1}^{Y}\sum_{c=1}^{C} I(c, x-x', y-y')\alpha(c)\,\beta(y')\,\gamma(x')$$
$$= \sum_{x=1}^{X}\sum_{y'=1}^{Y}\sum_{c=1}^{C} I(c, x-x', y-y')\,\widehat{W}(c, x', y')$$

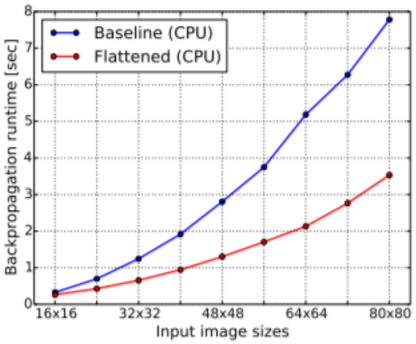
- $\widehat{W} = \alpha \otimes \beta \otimes \gamma, Rank\left(\widehat{W}\right) = 1$
- $\widehat{W}_S = \sum_{k=1}^{K} \alpha_k \otimes \beta_k \otimes \gamma_k$, Rank K
- SVD: Can reconstruct the original matrix as $A = \sum_{k=1}^{K} w_k u_k \otimes v_k$

Denton, Emily L., et al. "Exploiting linear structure within convolutional networks for efficient evaluation." *Advances in Neural Information Processing Systems*. 2014.

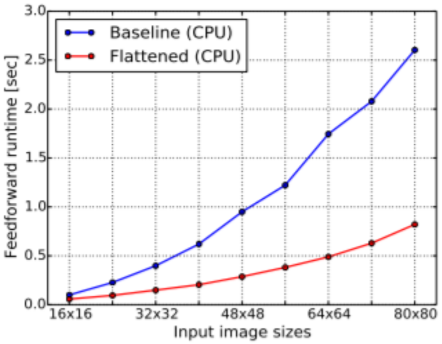
Flattening: Speedup Results

- 3 convolutional layers (5x5 filters) with 96, 128, and 256 channels
- Used stacks of 2 rank-1 convolutions





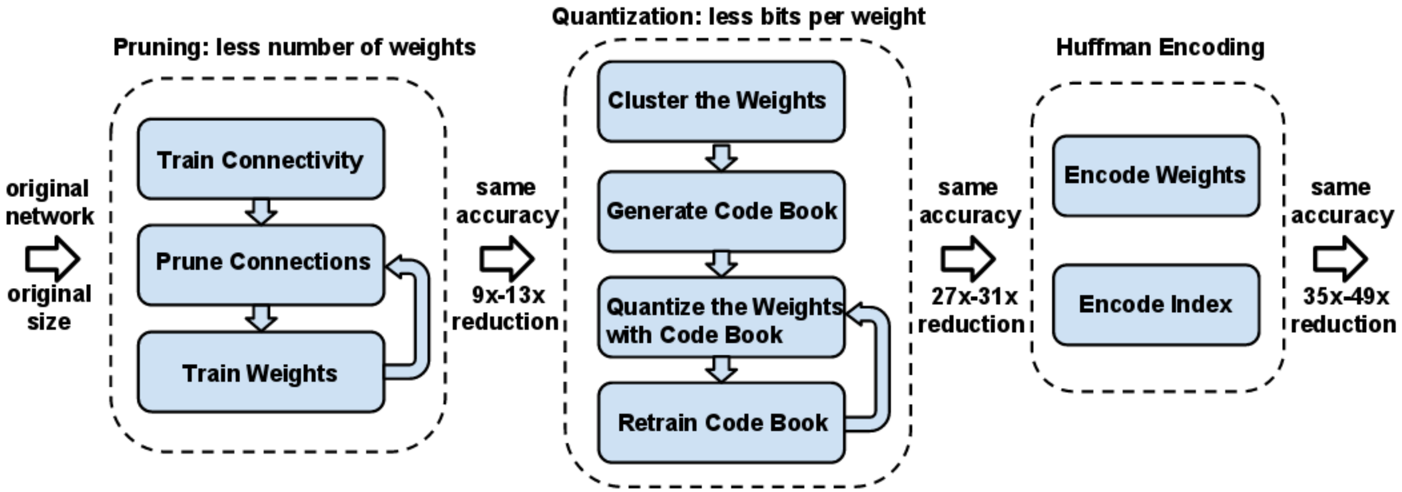
(c) Backpropagation on CPU



(a) Feedforward on CPU

Jin, Jonghoon, Aysegul Dundar, and Eugenio Culurciello. "Flattened convolutional neural networks for feedforward acceleration." *arXiv preprint arXiv:1412.5474* (2014).

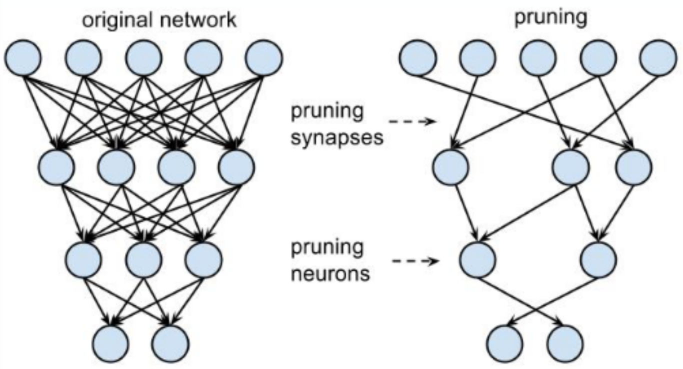
Other Deep Compressions?



Pruning, Trained Quantization and Huffman Coding", Song Han et al., ICLR 2016

Pruning

- Remove weights/synapses "close to zero"
- Retrain to maintain accuracy
- Repeat



Sparse Network

Pruning Results

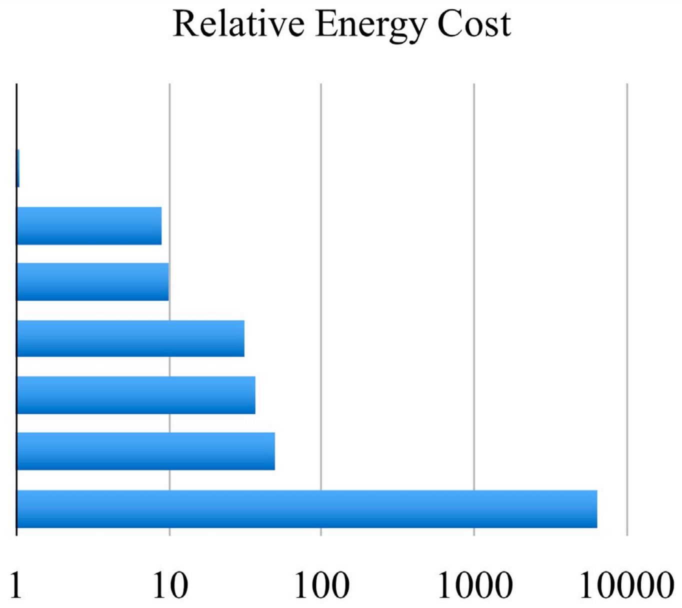
Network	Top-1 Error	Top-5 Error	Parameters	Compression Rate
LeNet-300-100 Ref	1.64%	-	267K	
LeNet-300-100 Pruned	1.59%	-	**22K**	**12**×
LeNet-5 Ref	0.80%	-	431K	
LeNet-5 Pruned	0.77%	-	**36K**	**12**×
AlexNet Ref	42.78%	19.73%	61M	
AlexNet Pruned	42.77%	19.67%	**6.7M**	**9**×
VGG16 Ref	31.50%	11.32%	138M	
VGG16 Pruned	31.34%	10.88%	**10.3M**	**13**×

Table 1: Network pruning can save 9× to 13× parameters with no drop in predictive performance

Magnitude-based method: Iterative Pruning + Retraining

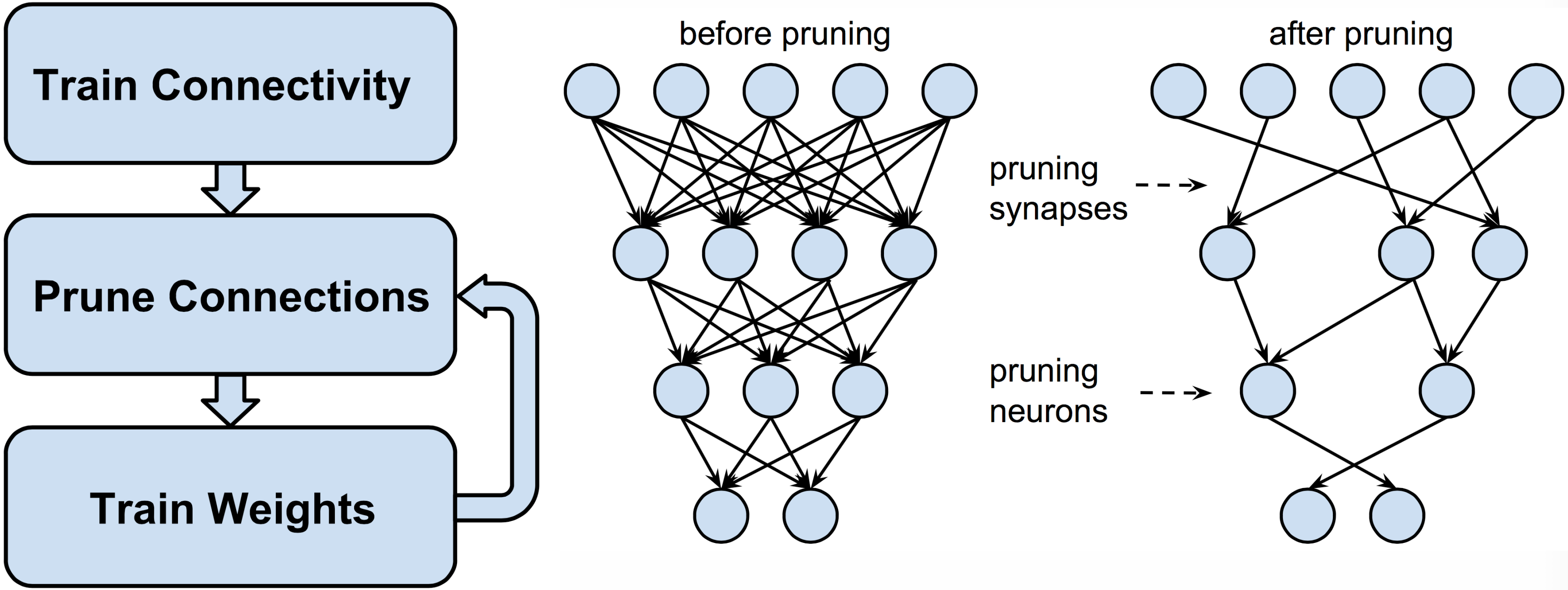
- Pruning connection with small magnitude.
- Iterative pruning an re-training.

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	640	6400



Han, Song, et al. "Learning both weights and connections for efficient neural network." NIPS. 2015.

Magnitude-based method: Iterative Pruning + Retraining



Han, Song, et al. "Learning both weights and connections for efficient neural network." NIPS. 2015.

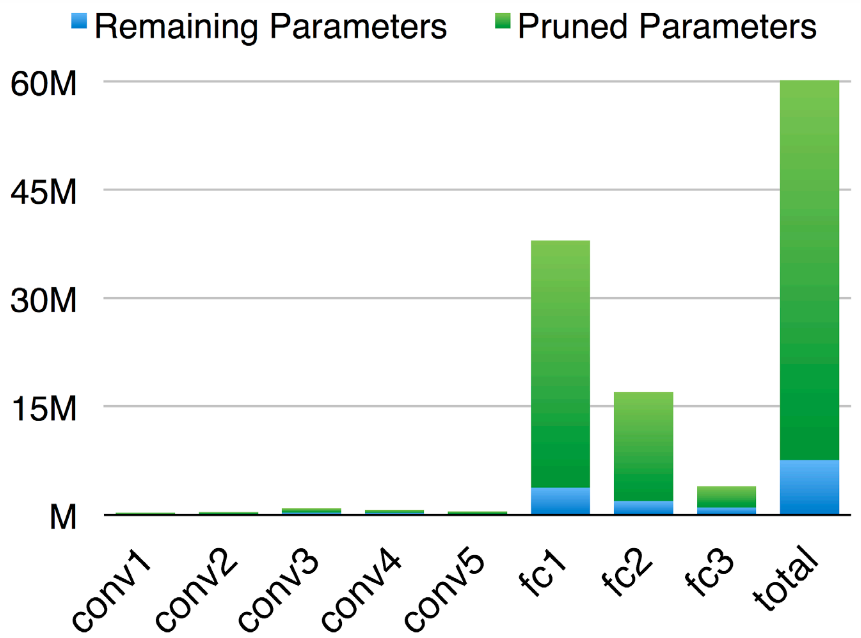
Magnitude-based method: Iterative Pruning + Retraining (Algorithm)

- 1. Choose a neural network architecture.
- 2. Train the network until a reasonable solution is obtained.
- 3. Prune the weights of which magnitudes are less than a threshold τ.
- 4. Train the network until a reasonable solution is obtained.
- 5. Iterate to step 3.

Han, Song, et al. "Learning both weights and connections for efficient neural network." NIPS. 2015.

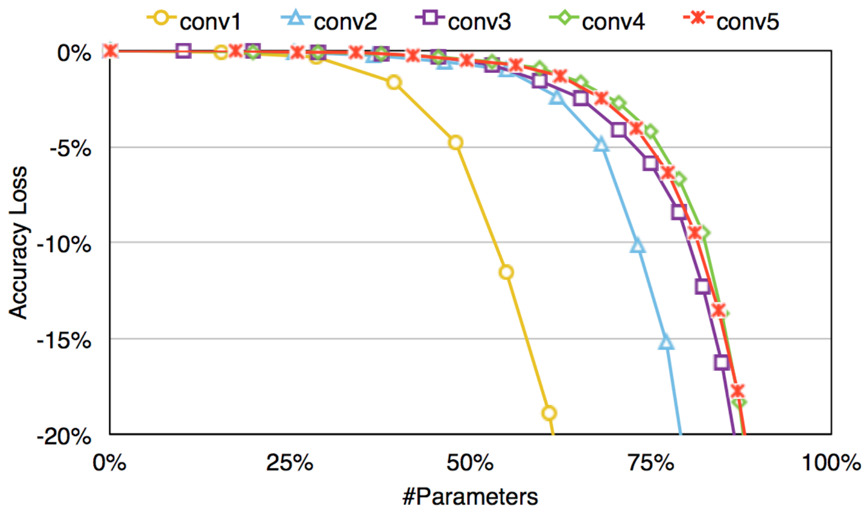
Magnitude-based method: Iterative Pruning + Retraining (Experiment: AlexNet)

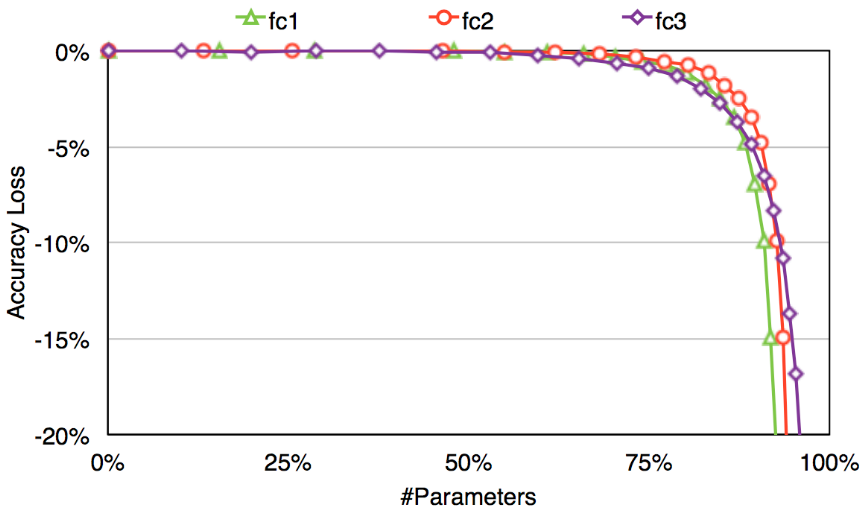
Layer	Weights	FLOP	Act%	Weights %	FLOP%
conv1	35K	211M	88%	84%	84%
conv2	307K	448M	52%	38%	33%
conv3	885K	299M	37%	35%	18%
conv4	663K	224M	40%	37%	14%
conv5	442K	150M	34%	37%	14%
fc1	38M	75M	36%	9%	3%
fc2	17M	34M	40%	9%	3%
fc3	4M	8M	100%	25%	10
Total	61M	1.5B	54%	11%	30%



Han, Song, et al. "Learning both weights and connections for efficient neural network." NIPS. 2015.

Magnitude-based method: Iterative Pruning + Retraining (Experiment: Tradeoff)





Han, Song, et al. "Learning both weights and connections for efficient neural network." NIPS. 2015.

Pruning with rehabilitation: Dynamic Network Surgery (Motivation)

- Pruned connections have no chance to come back.
- Incorrect pruning may cause severe accuracy loss.
- Avoid the risk of irretrievable network damage .
- Improve the learning efficiency.

Guo, Yiwen, et al. "Dynamic Network Surgery for Efficient DNNs." NIPS. 2016.

Pruning with rehabilitation: Dynamic Network Surgery (Formulation)

- W_k denotes the weights, and T_k denotes the corresponding 0/1 masks.
- $\min_{W_k, T_k} L(W_k \odot T_k) \quad s.t.\ \ {T_k}^{(i,j)} = h_k\left({W_k}^{(i,j)}\right), \forall (i,j) \in \mathfrak{I}$
  - $\odot$ is the element-wise product. $L(\cdot)$ is the loss function.

- Dynamic network surgery updates only W_k. T_k is updated based on $h_k(\cdot)$.
- $h_k\left({W_k}^{(i,j)}\right) = \begin{cases} 0 & a_k \geq \left|{W_k}^{(i,j)}\right| \\ {T_k}^{(i,j)} & a_k \leq \left|{W_k}^{(i,j)}\right| \leq b_k \\ 1 & b_k \leq \left|{W_k}^{(i,j)}\right| \end{cases}$
  - a_k is the pruning threshold. $b_k = a_k + t$, where t is a pre-defined small margin.

Guo, Yiwen, et al. "Dynamic Network Surgery for Efficient DNNs." NIPS. 2016.

Pruning with rehabilitation: Dynamic Network Surgery (Algorithm)

- 1. Choose a neural network architecture.
- 2. Train the network until a reasonable solution is obtained.
- 3. Update T_k based on $h_k(\cdot)$.
- 4. Update W_k based on back-propagation.
- 5. Iterate to step 3.

Guo, Yiwen, et al. "Dynamic Network Surgery for Efficient DNNs." NIPS. 2016.

Pruning with rehabilitation: Dynamic Network Surgery (Experiment on AlexNet)

Layer	Parameters	Parameters (Han et al. 2015)	Parameters (DNS)
conv1	35K	84%	53.8%
conv2	307K	38%	40.6%
conv3	885K	35%	29.0%
conv4	664K	37%	32.3%
conv5	443K	37%	32.5%
fc1	38M	9%	3.7%
fc2	17M	9%	6.6%
fc3	4M	25%	4.6%
Total	61M	11%	5.7%

Guo, Yiwen, et al. "Dynamic Network Surgery for Efficient DNNs." NIPS. 2016.

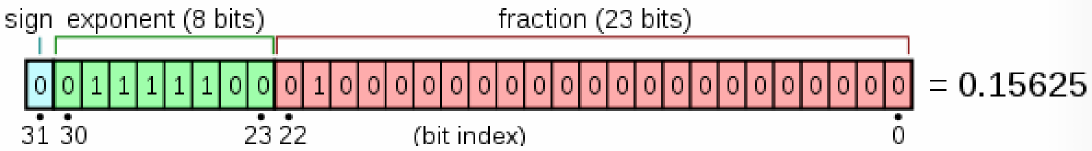
Why Reduced Precision Computing?

- Decrease the inference speed
- Decrease the model size (memory)
- Keeping same level of accuracy
- → Approach: Reduced Precision Computing

Default Arithmetic in Deep Learning

- The default arithmetic in deep learning frameworks (TensorFlow, PyTorch) is 32-bit floating point (float32) or single precision

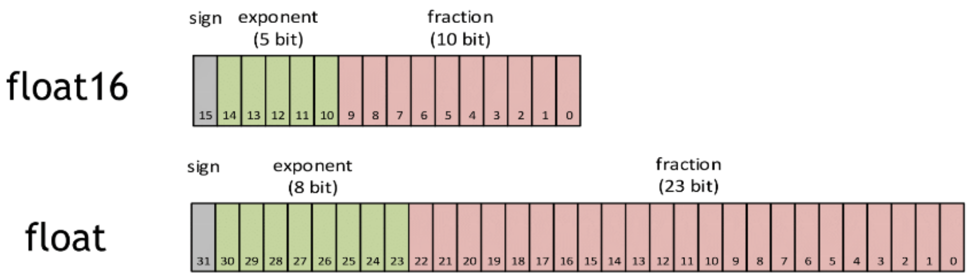
- Float32:
  - 1 bit for the sign
  - 8 bits for the exponent
  - 24 bits for the fraction



- Max value: $3.4 * 10^{38}$

Reduced Precision

- Float16 or half precision:
- 16 bits
  - 1 sign
  - 5 exponent
  - 10 fraction
- Max value : 65504

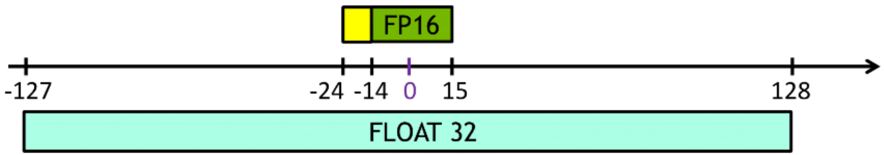


- Integer 8 (int8):
- 8 bits (no fraction!)
- Max value : 256 values

- And even smaller formats

FLOAT16 has wide range (2^{40}) … but not as wide as FP32!

Normal range: [6×10^{-5} , 65504]
Sub-normal range: [6×10^{-8} , 6×10^{-5}]



What is Quantization?

- Converting numbers from one format to another
- More specific: From a higher precision to a lower precision (float32 → int8)

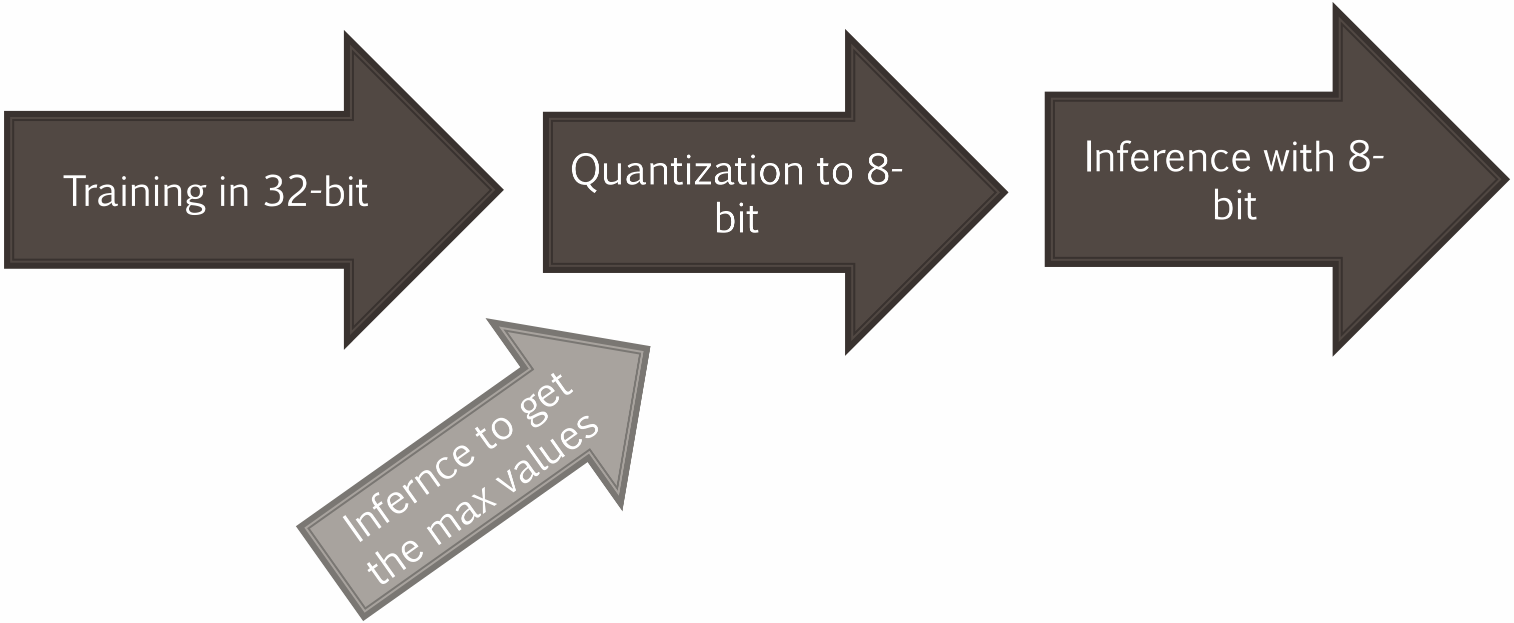
- In theory quantization is simple:
- Example float32 to int8
  - Find scaling factor:
    - Search the maximum abslute number B
    - Scaling factor: $S = \frac{255}{B}$
  - Quantizing:
    - $Q = round(S * N_{32})$

$$N = [15, 50, 200] \rightarrow B = 200$$

$$S = \frac{255}{200} = 1{,}275$$

$$Q_1 = round(1{,}275 * 15) = 19$$
$$Q_2 = round(1{,}275 * 50) = 64$$
$$Q_3 = round(1{,}275 * 200) = 255$$

Post-Training Quantization

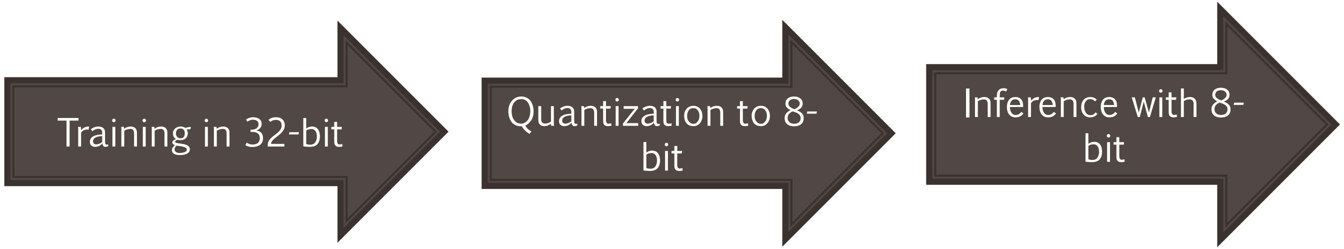


Quantization with Neural Networks

- Operations to quantize:
  - Weights, Bias
  - Activations

- Depending on the quantization tool, it is necessary to quantize the input into the neural network

- Depending on the tool you can get the output in quantized format



Possible Benefits of Quantization

- Reduction in model size
- Reduction in memory bandwidth
- Faster inference

- Example: float32 → int8
- 4x
- 2-4x *
- 2-4x *

- *depending on hardware and the modle

Disadvantage of Quantization

- Only one disadvantage: Slightly less accuracy
  - Roughly 2-3 %
  - Depends greatly on the model

- Constraint: Not many hardware devices support low precision computing
  - Even if low precisions are not supported yet, you can run inference with the quantized models, but you don 't see any speed up

Why is 8-bit Quantization so Famous?

- Because it is the smallest format which is supported by mainstream hardware devices
- Lower bit numbers have a strong drop in accuracy
- Typically when it is talked about quantization, they mean int8

Example Quantization Tools

- In general HWs
  - TensorFlow Lite supports quantization to float16 and int8
    - Generally work well: like MTK dimensity series, Quancomm Adreno, etc.
  - PyTorch -> Pytorch mobile
  - Intel Low-Precision Quantization tool

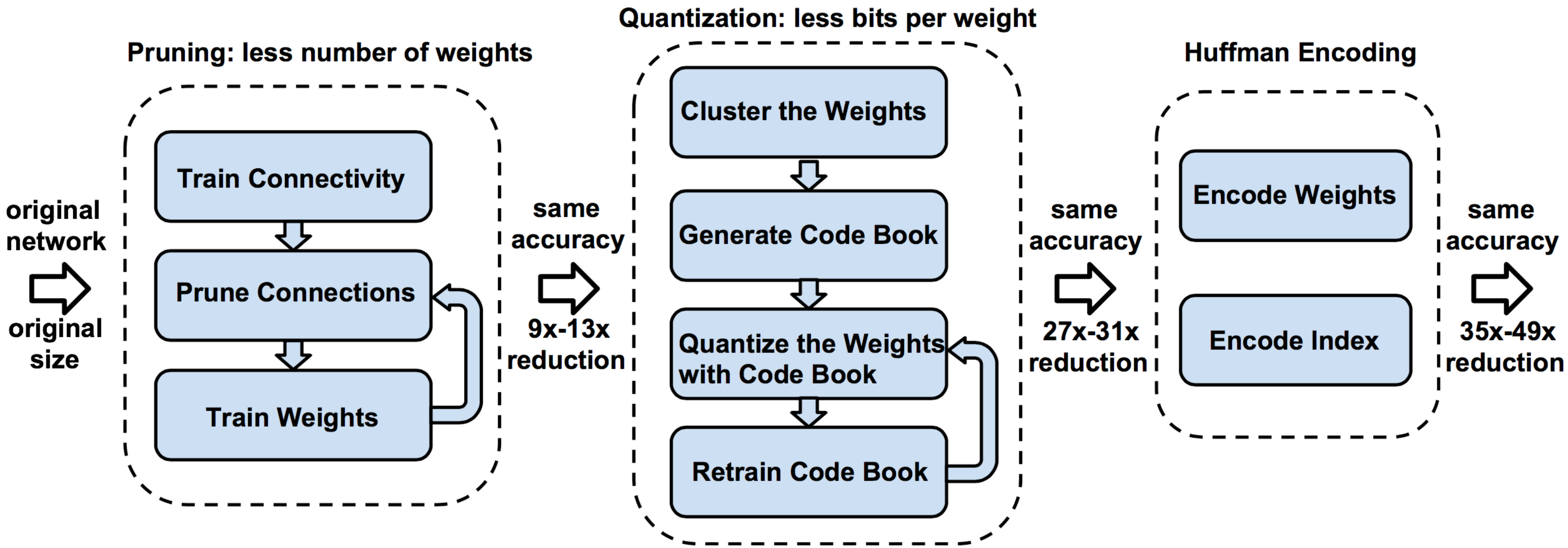
- In Nvidia-related platform, like nx, tx2, px2
  - Tensorrt
- In other platforms like Kneron
  - Onnx (cross-platform)
- Intel-based
  - Openvino

Many tricks on Embedding System

- TF Lite version
  - Could be affected by TF version, HW supports, and even naïve ops.
  - Quantization supporting: <TF1.3, only QUint8 and FP32. >TF1.3.2, FP16 supported
- Pytorch mobile
  - New one, but not verified comprehensively yet.
- Intel openvino
  - OK but the supported ops are incomplete
- Other platforms
  - Many issues on "versions" from Pytorch or TF

COMBINE THEM TOGETHER

Pruning + Quantization + Encoding: Deep Compression



Courbariaux, Matthieu, et al. "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1." arXiv preprint arXiv:1602.02830 (2016).

Pruning + Quantization + Encoding: Deep Compression

- 1. Choose a neural network architecture.
- 2. Train the network until a reasonable solution is obtained.
- 3. Prune the network with magnitude-based method until a reasonable solution is obtained.
- 4. Quantize the network with k-means based method until a reasonable solution is obtained.
- 5. Further compress the network with Huffman coding.

Courbariaux, Matthieu, et al. "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1." arXiv preprint arXiv:1602.02830 (2016).

Pruning + Quantization + Encoding: Deep Compression

Table 1: The compression pipeline can save $35\times$ to $49\times$ parameter storage with no loss of accuracy.

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100 Ref	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	**40**$\times$
LeNet-5 Ref	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	**39**$\times$
AlexNet Ref	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	**35**$\times$
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	**49**$\times$

Table 2: Compression statistics for LeNet-300-100. P: pruning, Q:quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
ip1	235K	8%	6	4.4	5	3.7	3.1%	2.32%
ip2	30K	9%	6	4.4	5	4.3	3.8%	3.04%
ip3	1K	26%	6	4.3	5	3.2	15.7%	12.70%
Total	266K	8%(12$\times$)	6	5.1	5	3.7	3.1% (**32**$\times$)	2.49% (**40**$\times$)

Table 3: Compression statistics for LeNet-5. P: pruning, Q:quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1	0.5K	66%	8	7.2	5	1.5	78.5%	67.45%
conv2	25K	12%	8	7.2	5	3.9	6.0%	5.28%
ip1	400K	8%	5	4.5	5	4.5	2.7%	2.45%
ip2	5K	19%	5	5.2	5	3.7	6.9%	6.13%
Total	431K	8%(12$\times$)	5.3	4.1	5	4.4	3.05% (**33**$\times$)	2.55% (**39**$\times$)

Pruning + Quantization + Encoding: Deep Compression

Table 4: Compression statistics for AlexNet. P: pruning, Q: quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1	35K	84%	8	6.3	4	1.2	32.6%	20.53%
conv2	307K	38%	8	5.5	4	2.3	14.5%	9.43%
conv3	885K	35%	8	5.1	4	2.6	13.1%	8.44%
conv4	663K	37%	8	5.2	4	2.5	14.1%	9.11%
conv5	442K	37%	8	5.6	4	2.5	14.0%	9.43%
fc6	38M	9%	5	3.9	4	3.2	3.0%	2.39%
fc7	17M	9%	5	3.6	4	3.7	3.0%	2.46%
fc8	4M	25%	5	4	4	3.2	7.3%	5.85%
Total	61M	11%(9×)	5.4	4	4	3.2	3.7% (**27**×)	2.88% (**35**×)

Table 5: Compression statistics for VGG-16. P: pruning, Q:quantization, H:Huffman coding.

Layer	#Weights	Weights% (P)	Weigh bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1_1	2K	58%	8	6.8	5	1.7	40.0%	29.97%
conv1_2	37K	22%	8	6.5	5	2.6	9.8%	6.99%
conv2_1	74K	34%	8	5.6	5	2.4	14.3%	8.91%
conv2_2	148K	36%	8	5.9	5	2.3	14.7%	9.31%
conv3_1	295K	53%	8	4.8	5	1.8	21.7%	11.15%
conv3_2	590K	24%	8	4.6	5	2.9	9.7%	5.67%
conv3_3	590K	42%	8	4.6	5	2.2	17.0%	8.96%
conv4_1	1M	32%	8	4.6	5	2.6	13.1%	7.29%
conv4_2	2M	27%	8	4.2	5	2.9	10.9%	5.93%
conv4_3	2M	34%	8	4.4	5	2.5	14.0%	7.47%
conv5_1	2M	35%	8	4.7	5	2.5	14.3%	8.00%
conv5_2	2M	29%	8	4.6	5	2.7	11.7%	6.52%
conv5_3	2M	36%	8	4.6	5	2.3	14.8%	7.79%
fc6	103M	4%	5	3.6	5	3.5	1.6%	1.10%
fc7	17M	4%	5	4	5	4.3	1.5%	1.25%
fc8	4M	23%	5	4	5	3.4	7.1%	5.24%
Total	138M	7.5%(13×)	6.4	4.1	5	3.1	3.2% (**31**×)	2.05% (**49**×)

NEXT TIME: LATEST CNNS