Intro to Deep Learning Systems

TensorFlow/PyTorch Tutorial

Parametric model

→ Ideal Feature Extractor → Ideal Decision Function →

window, top-left
clock, top-middle
shelf, left
…
drawing, middle
hat, bottom right

Q: What types of features shall we consider?

Q: What types of decision functions shall we consider?

Representation learning

Features
+ Decision
$f(x\,;\,\theta)$

window, top-left
clock, top-middle
shelf, left
…

drawing, middle
hat, bottom right

Q: Which class of functions shall we consider for *f* ?

Representation learning

Features
+ Decision
$f(x\,;\,\theta)$

window, top-left
clock, top-middle
shelf, left
…

drawing, middle
hat, bottom right

Proposal: Composing a set of nonlinear functions *g*

$$f(x;\theta) = g_1(\,g_2(\,\ldots g_n(x;\theta_n)\,\ldots;\theta_2\,);\theta_1\,)$$

Deep learning (brief intro)

Deep Learning: Composing a set of nonlinear functions *g*

$$f(x;\theta) = g_1(\, g_2(\, \dots g_n(x;\theta_n) \dots ;\theta_2\,);\theta_1\,)$$

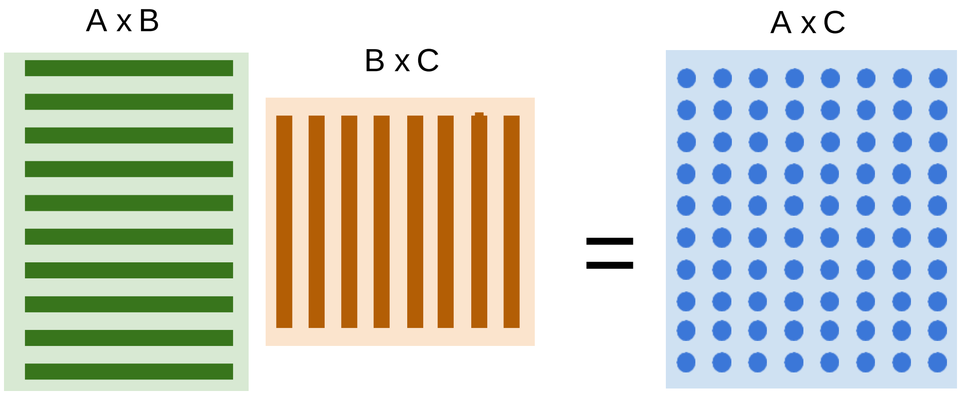
- Make predictions by using a sequence of non-linear processing stages
- The resulting intermediate representations can be interpreted as feature hierarchies
- The whole system is jointly learned from data

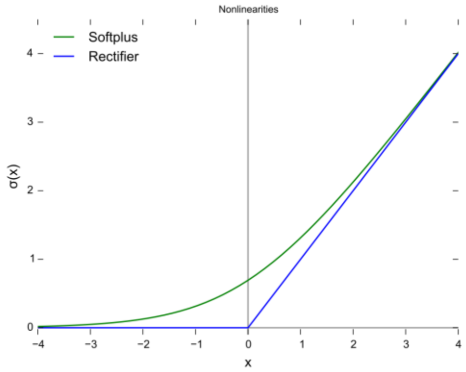
Deep learning (brief intro)

Deep Learning: Composing a set of nonlinear functions g

$$f(x;\theta) = g_1(\, g_2(\, \dots g_n(x;\theta_n) \dots ;\theta_2\,);\theta_1\,)$$

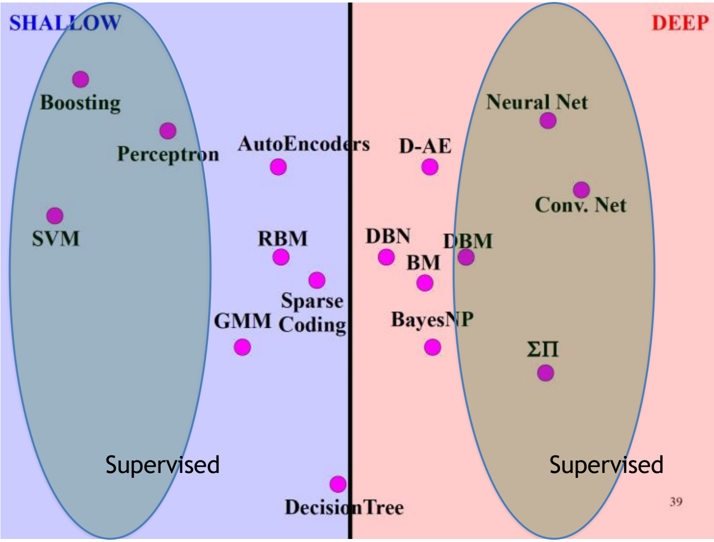
Key Elements: Linear operations + Nonlinear activations





Deep learning (brief intro)

- A large family of methods!
  - supervised / unsupervised
  - probabilistic / deterministic



- Computational demanding
- Require new **hardware and software**

Deep Learning Software

Generations of frameworks!

Caffe (UC Berkeley)	Layers + Blobs
TensorFlow (Google)	Static Computational Graph
PyTorch (Facebook)	Dynamic Computational Graph

What is the difference?

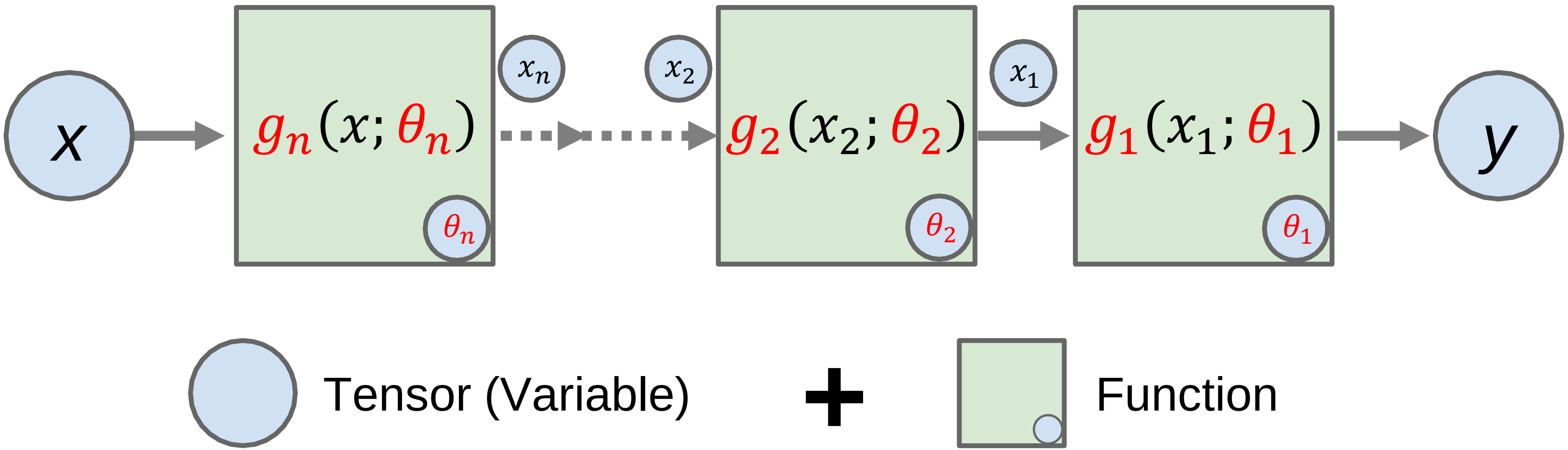
Designing a deep learning system

Key Principle: How to present the composition of nonlinear functions & make the computation efficient?

$$f(x;\theta) = g_1(\, g_2(\, \ldots g_n(x;\theta_n) \ldots ; \theta_2\,); \theta_1\,)$$

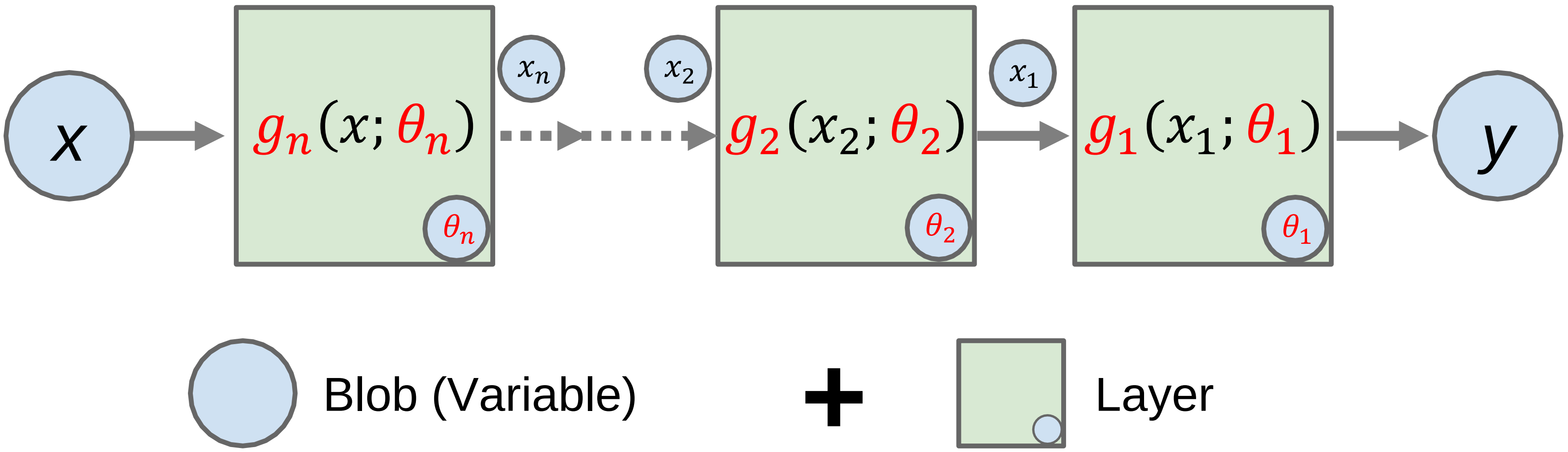
Stack of nonlinear functions

$$f(x;\theta) = g_1(\, g_2(\, \ldots g_n(x;\theta_n) \ldots; \theta_2\,); \theta_1\,)$$



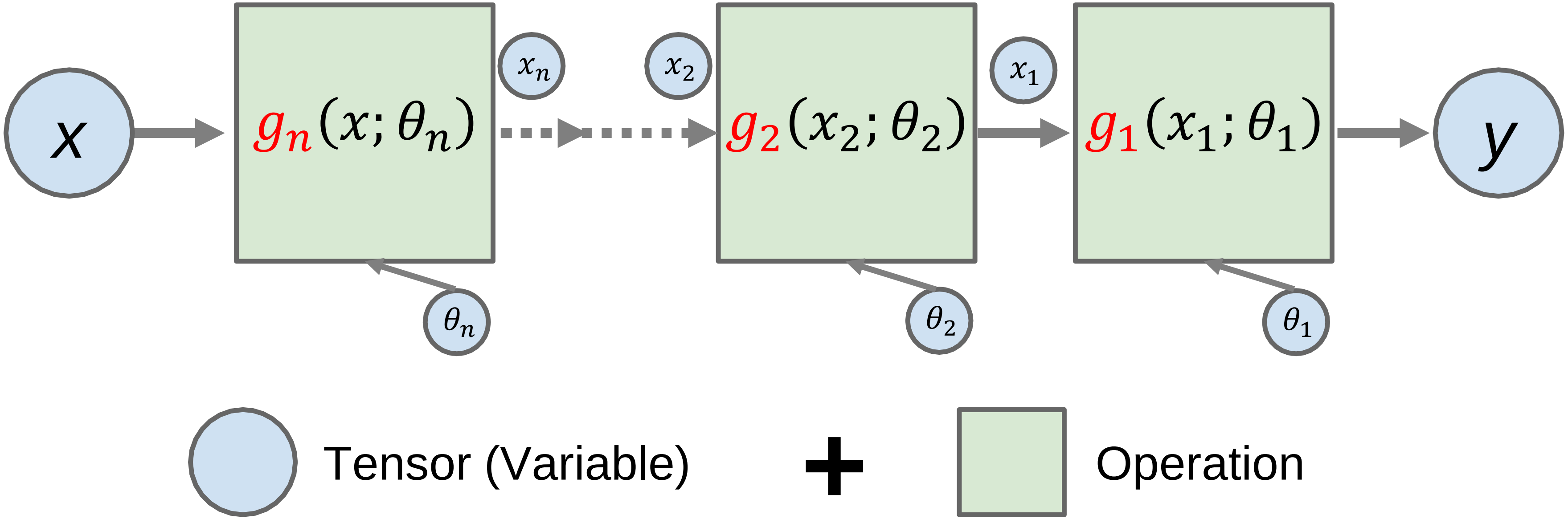
Caffe: Model = Blobs + Layers

$$f(x;\theta) = g_1(\,g_2(\,\dots g_n(x;\theta_n)\,\dots;\theta_2\,);\theta_1\,)$$



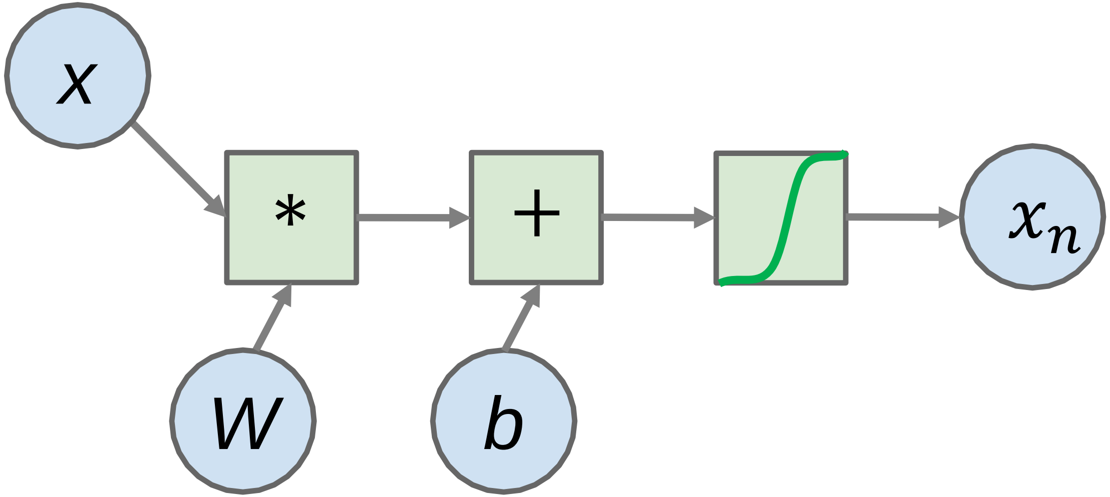
Tensors + Operations

$$f(x;\theta) = g_1(\, g_2(\, \ldots g_n(x;\theta_n) \ldots ;\theta_2\,);\theta_1\,)$$



Tensor + Operations = Computational graph

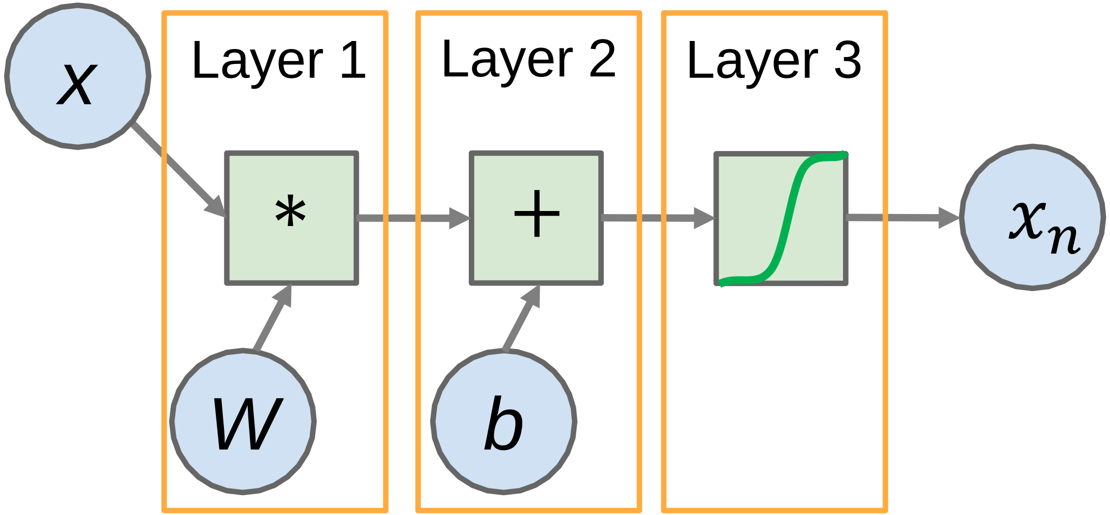
$$g_n(x; W, b) = \sigma(Wx + b)$$



- Decompose functions into atomic operations
- Separate data (tensors) and computing (ops)
- Highly modular

Static computational graph: Caffe

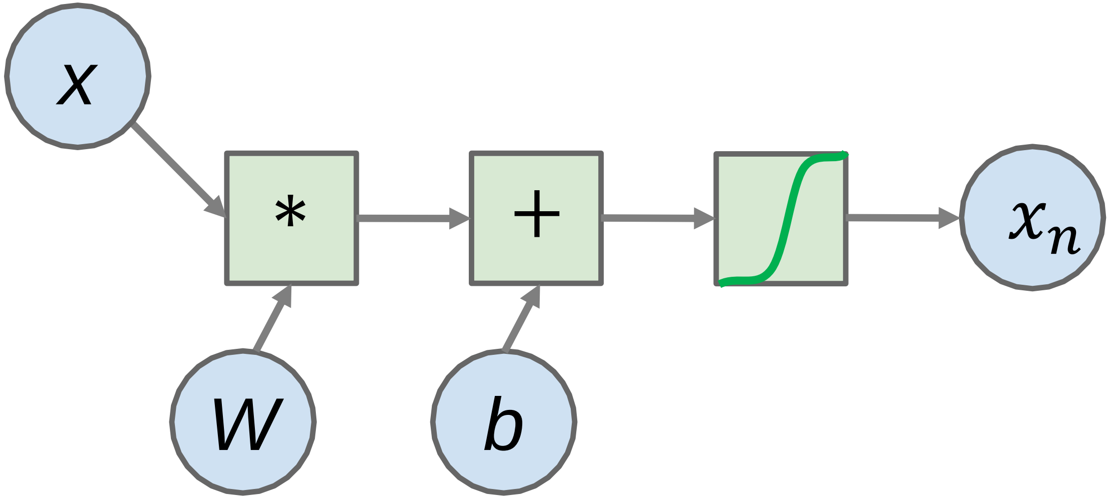
$$g_n(x; W, b) = \sigma(Wx + b)$$



- Intuitive (layer 1, 2, 3)
- Hard to optimize the graph
- Hard to reconfigure the graph (once built)

Static computational graph: TensorFlow

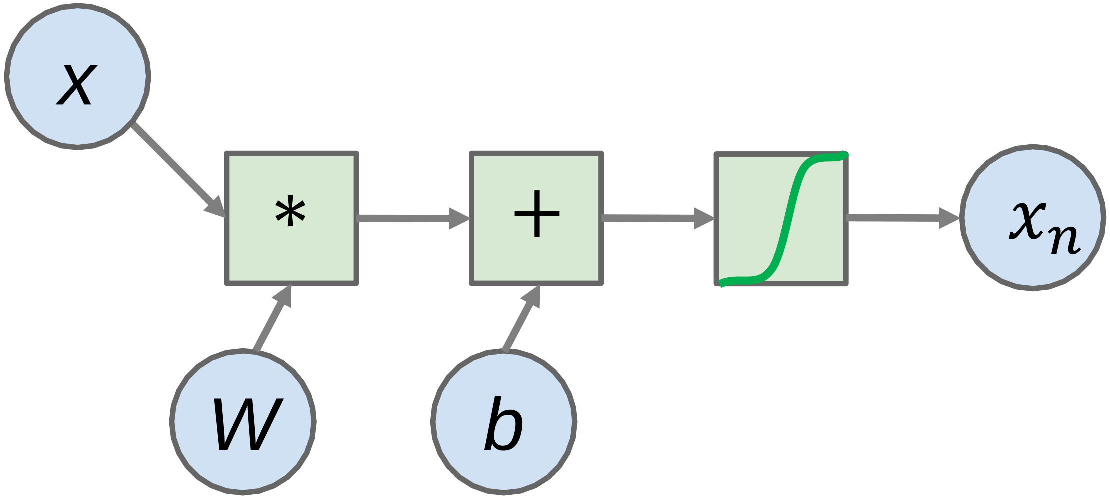
$$g_n(x; W, b) = \sigma(Wx + b)$$



- Build once and run many times
- Easy to optimize the graph
- Hard to reconfigure the graph (once built)

Dynamic computational graph: PyTorch

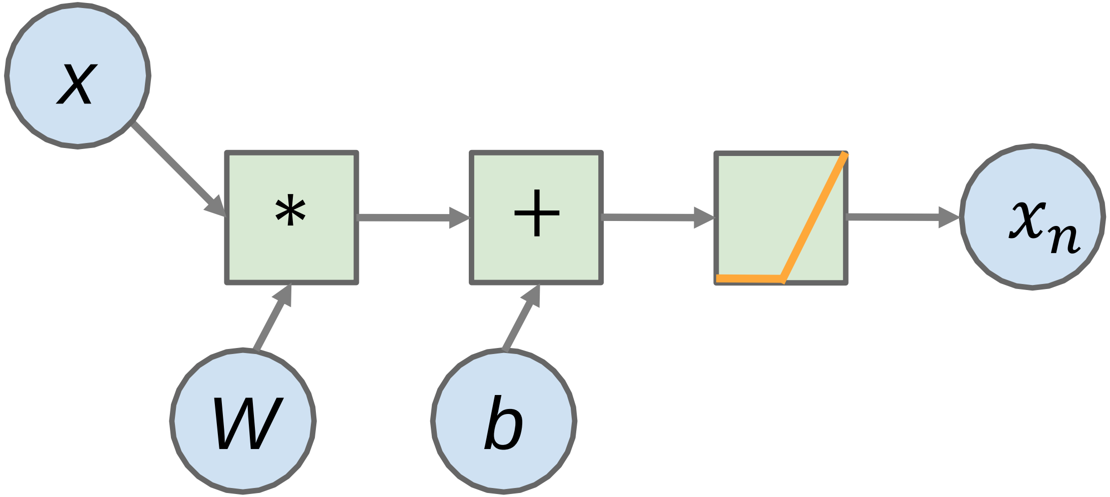
$$g_n(x; W, b) = \sigma(Wx + b)$$



- Rebuild the graph at each run

Dynamic computational graph: PyTorch

$$g_n(x; W, b) = \sigma(Wx + b)$$



- Rebuild the graph at each run
- Easy to reconfigure the graph
- Hard to optimize the graph

The point of deep learning frameworks

(1) Quick to develop and test new ideas
(2) Flexible for building complicated models
(3) Automatically compute gradients
(4) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

A zoo of frameworks!

Caffe
(UC Berkeley)

→

Caffe2
(Facebook)

Torch
(NYU / Facebook)

→

PyTorch
(Facebook)

Theano
(U Montreal)

→

TensorFlow
(Google)

PaddlePaddle
(Baidu)

Chainer

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

CNTK
(Microsoft)

Deeplearning4j

And others...

A zoo of frameworks!

Caffe
(UC Berkeley)

Caffe2
(Facebook)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Theano
(U Montreal)

TensorFlow
(Google)

We'll focus on these

PaddlePaddle
(Baidu)

Chainer

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

CNTK
(Microsoft)

Deeplearning4j

And others...

A zoo of frameworks!

Caffe
(UC Berkeley)

→

Caffe2
(Facebook)

Torch
(NYU / Facebook)

→

PyTorch
(Facebook)

We recommend PyTorch

Theano
(U Montreal)

→

TensorFlow
(Google)

PaddlePaddle
(Baidu)

Chainer

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

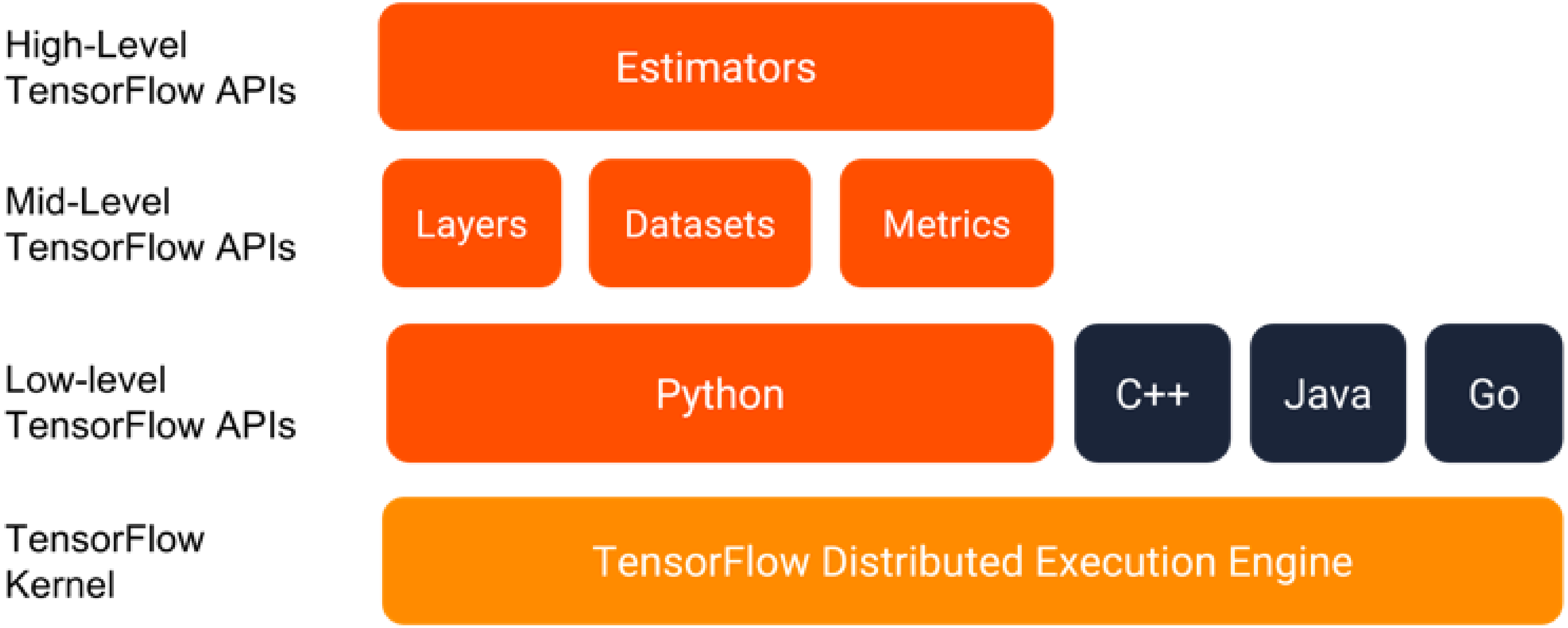
CNTK
(Microsoft)

Deeplearning4j

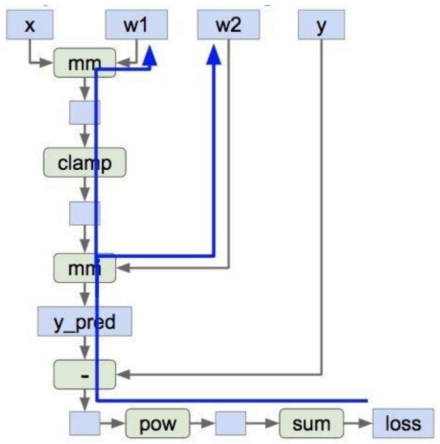
And others...

TensorFlow

TensorFlow Stack



Static Computation Graphs



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

Step 2: Reuse the same graph on every iteration

* Can be a bit tricky to implement!

```
graph = build_graph()

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

TensorFlow: Neural Net

First **define** computational graph

$y = w_2^T \, max(w_1^T x, 0)$

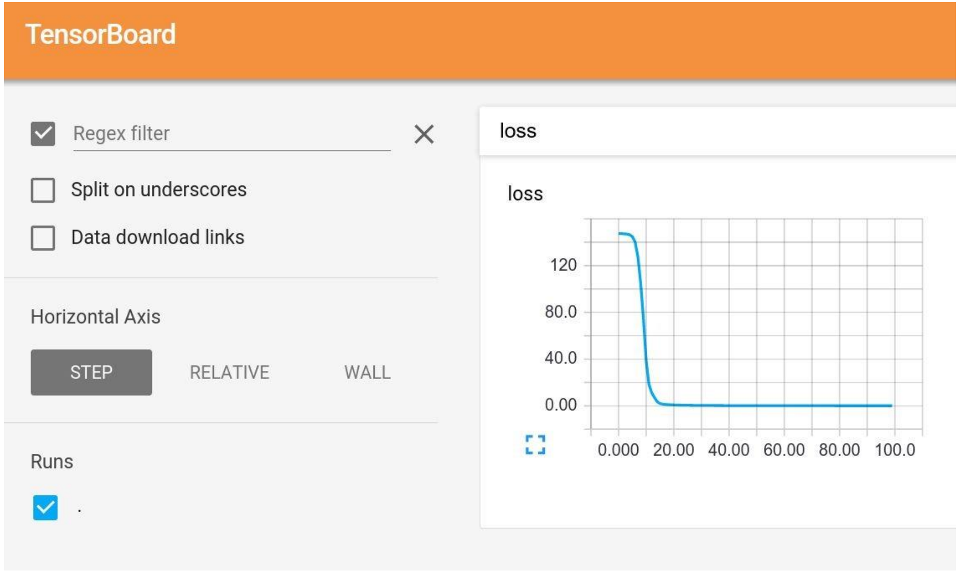
```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

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: Tensorboard

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



PyTorch

PyTorch: Fundamental Concepts

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

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

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

PyTorch: Versions

For this class we are using **PyTorch version 1.0.1**

Be careful if you are looking at older PyTorch code (< 0.4)!

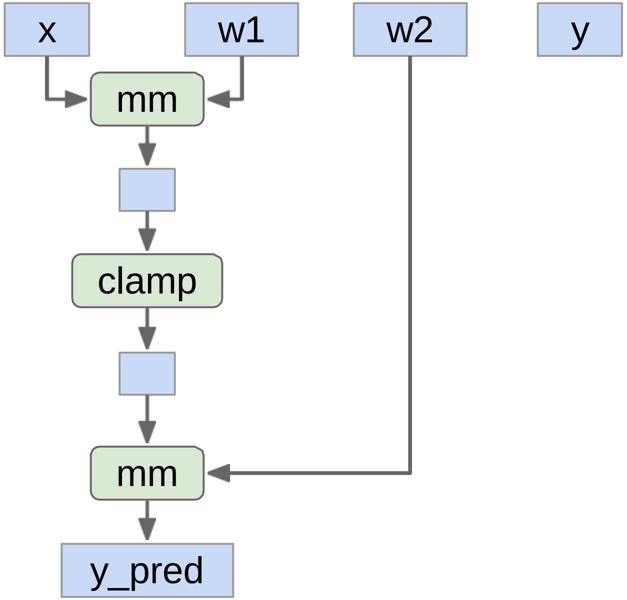
PyTorch: **Dynamic** Computation Graphs

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

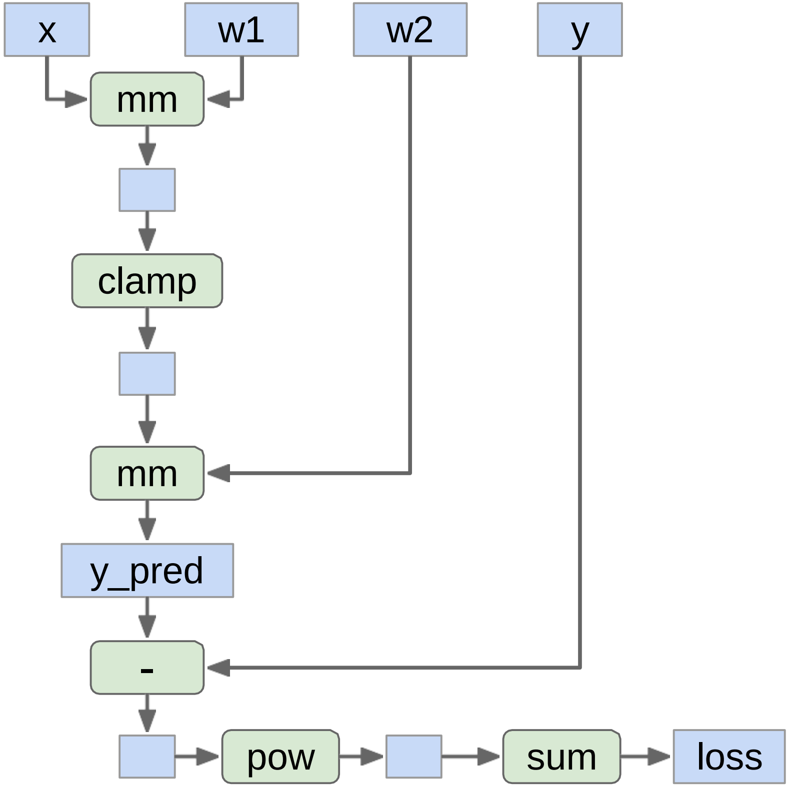



```python
import torch

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()
```

$$y = w_2^T \, max(w_1^T x, 0)$$

Build graph data structure AND perform computation

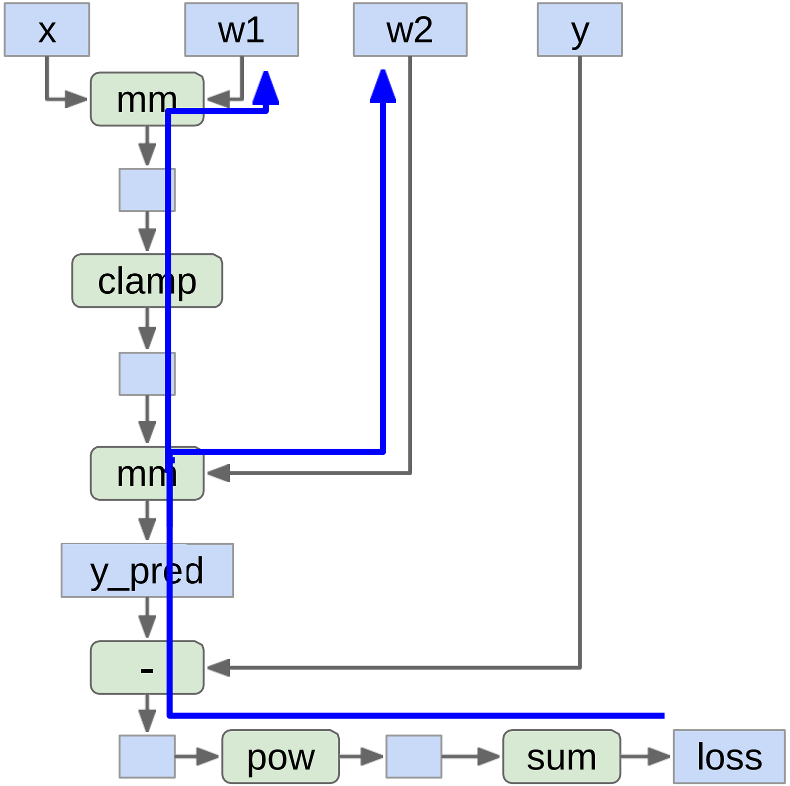
PyTorch: **Dynamic** Computation Graphs




```
import torch

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

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

    loss.backward()
```

Build graph data structure AND perform computation

PyTorch: **Dynamic** Computation Graphs




```
import torch

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

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

    loss.backward()
```

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

PyTorch: **Dynamic** Computation Graphs


```
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

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()
```

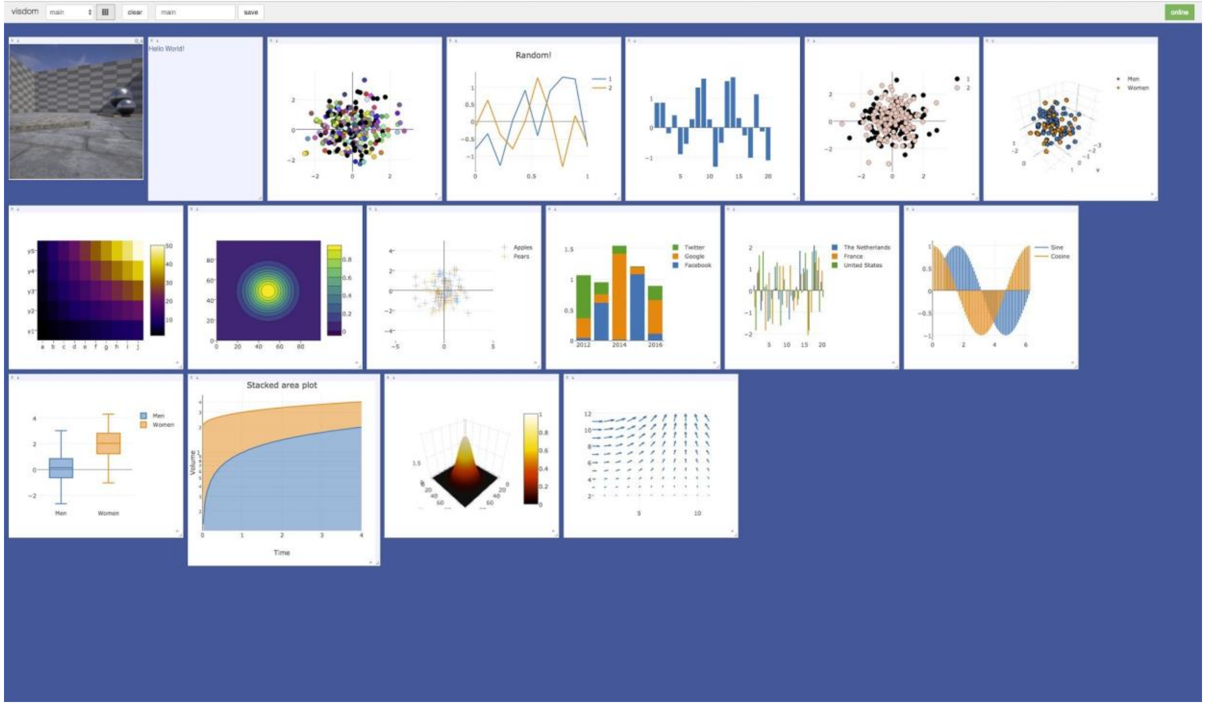
PyTorch: Demos

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py

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
Dynamic Graphs

vs

TensorFlow
Static Graphs

Static vs Dynamic Graphs

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

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((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])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Build graph

Run each iteration

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

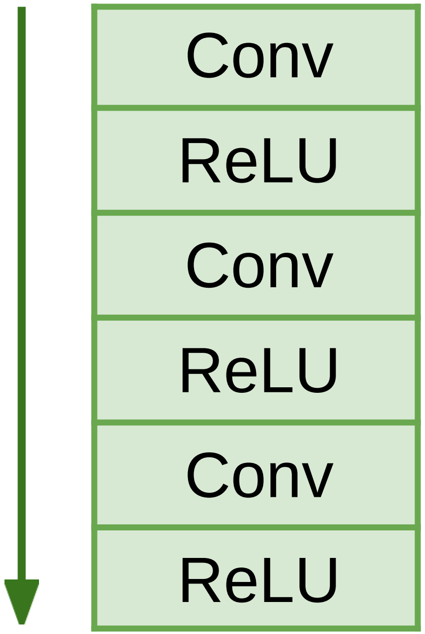
```
import torch

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

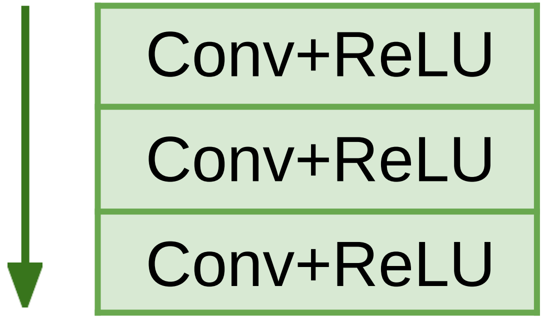
Static vs Dynamic: Optimization

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

Static vs Dynamic: Conditional

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

Static vs Dynamic: Conditional

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

```
N, D, H = 3, 4, 5

x = torch.randn(N, D, requires_grad=True)
w1 = torch.randn(D, H)
w2 = torch.randn(D, H)

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

Static vs Dynamic: Conditional

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

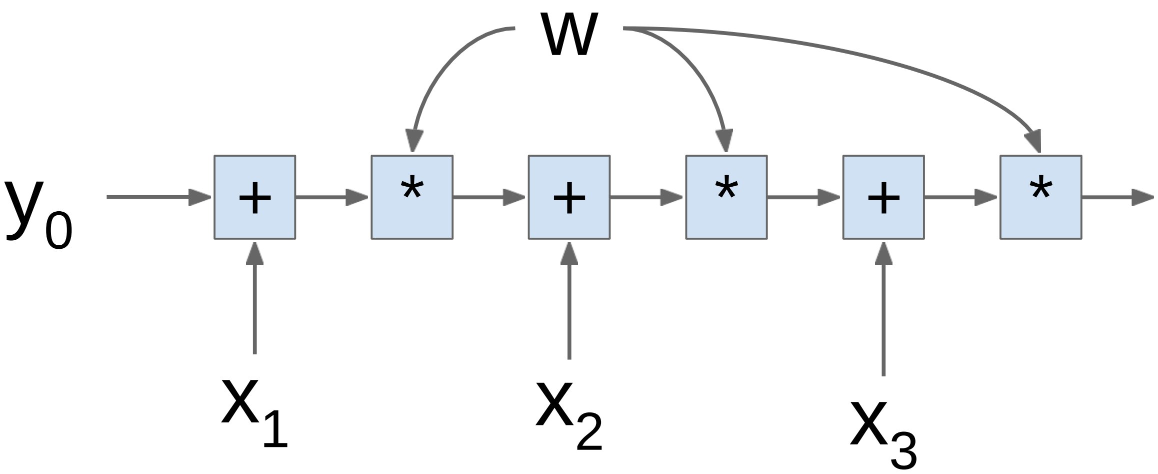
```
N, D, H = 3, 4, 5

x = torch.randn(N, D, requires_grad=True)
w1 = torch.randn(D, H)
w2 = torch.randn(D, H)

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

TensorFlow: Special TF control flow operator!

```
N, D, H = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=None)
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(D, H))

def f1(): return tf.matmul(x, w1)
def f2(): return tf.matmul(x, w2)
y = tf.cond(tf.less(z, 0), f1, f2)

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        z: 10,
        w1: np.random.randn(D, H),
        w2: np.random.randn(D, H),
    }
    y_val = sess.run(y, feed_dict=values)
```

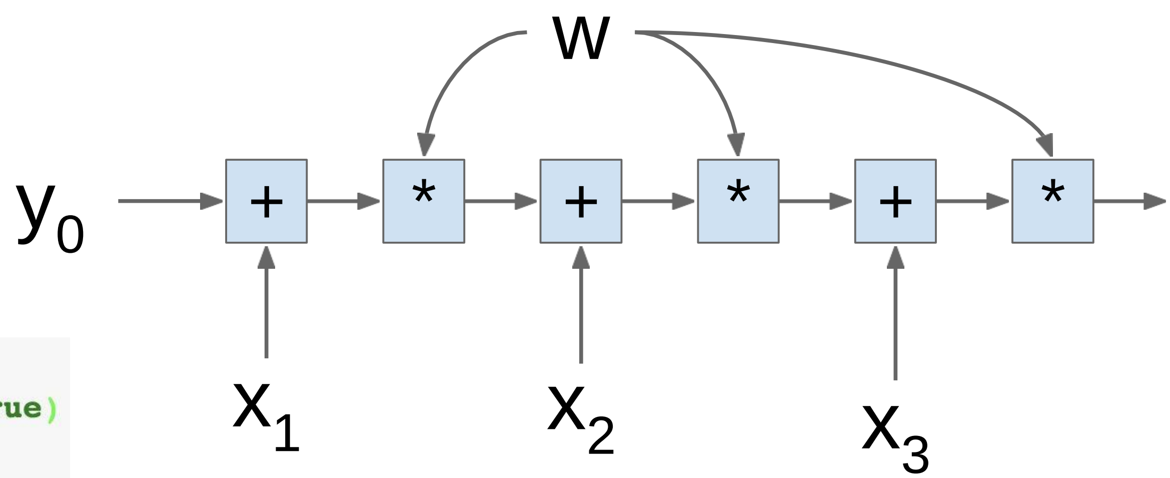
Static vs Dynamic: Loops

$y_t = (y_{t-1} + x_t) * w$



Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$



PyTorch: Normal Python

```
T, D = 3, 4
y0 = torch.randn(D, requires_grad=True)
x = torch.randn(T, D)
w = torch.randn(D)

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```

Static vs Dynamic: Loops

$$y_t = (y_{t-1} + x_t) * w$$

PyTorch: Normal Python

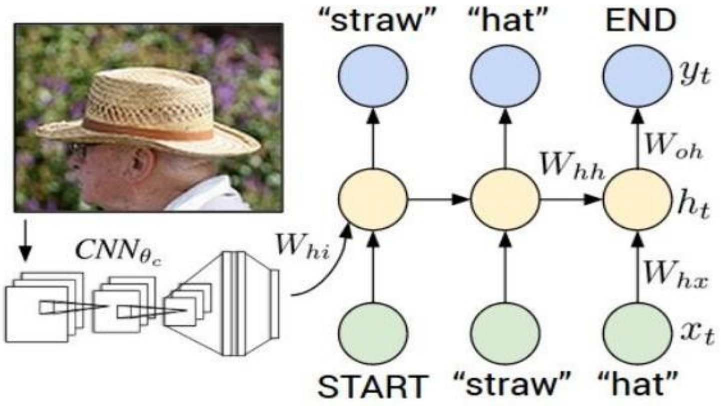
```
T, D = 3, 4
y0 = torch.randn(D, requires_grad=True)
x = torch.randn(T, D)
w = torch.randn(D)

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```

TensorFlow: Special TF control flow

```
T, N, D = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(T, D))
y0 = tf.placeholder(tf.float32, shape=(D,))
w = tf.placeholder(tf.float32, shape=(D,))

def f(prev_y, cur_x):
    return (prev_y + cur_x) * w

y = tf.foldl(f, x, y0)

with tf.Session() as sess:
    values = {
        x: np.random.randn(T, D),
        y0: np.random.randn(D),
        w: np.random.randn(D),
    }
    y_val = sess.run(y, feed_dict=values)
```

Dynamic Graph Applications

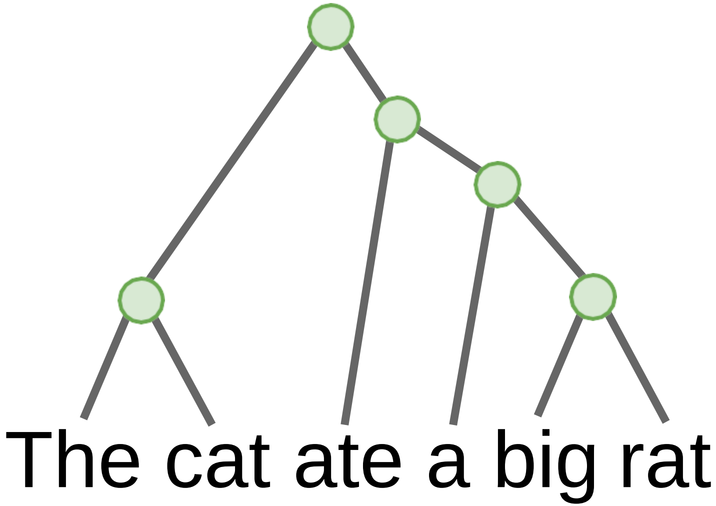
- Recurrent networks



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

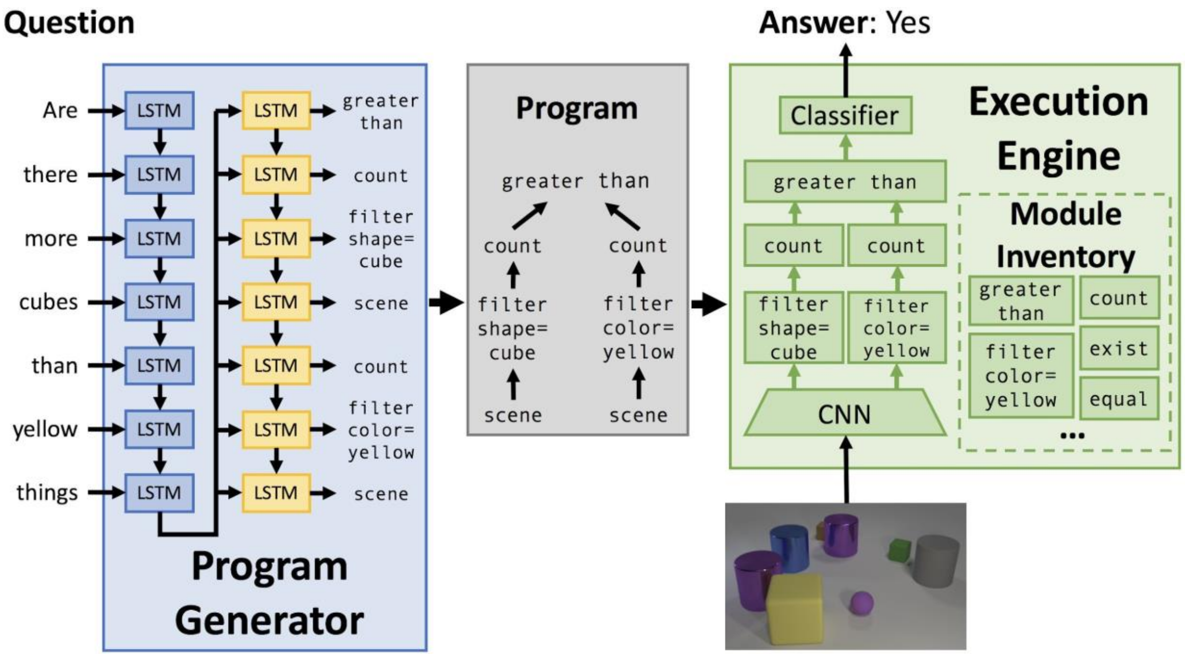
Dynamic Graph Applications

- 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
Static: ONNX

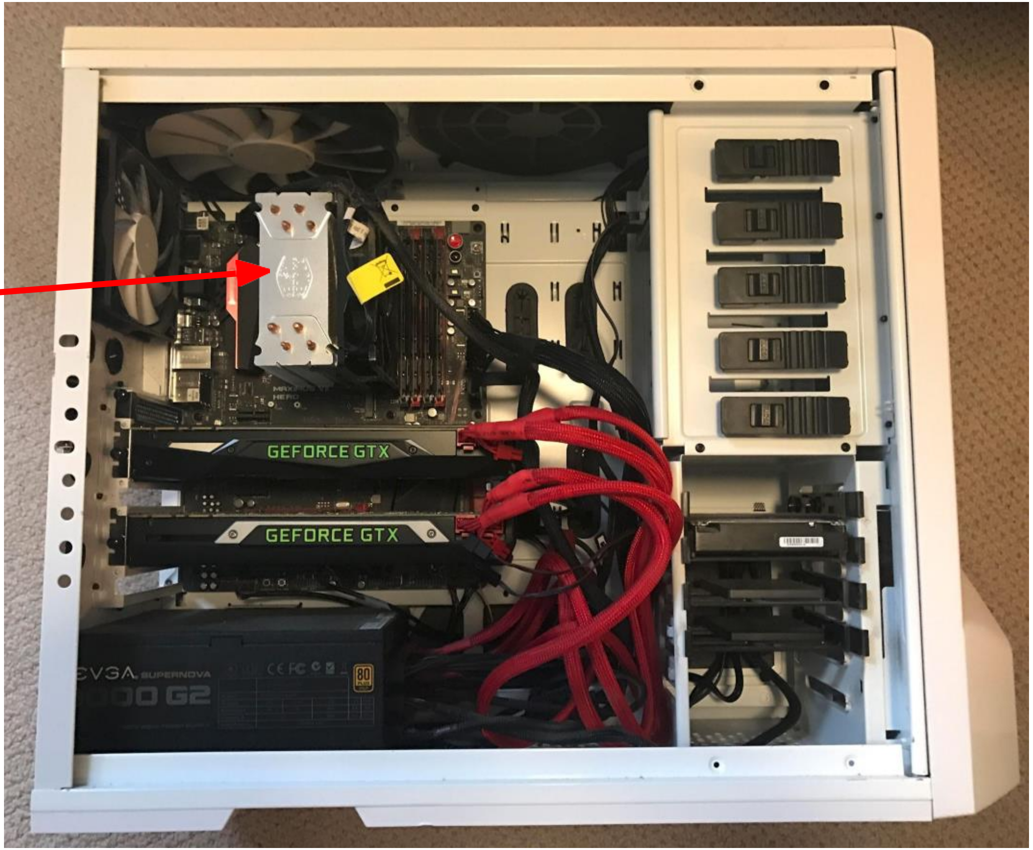
TensorFlow
Static Graphs
Dynamic: Eager

Deep Learning Hardware

My computer

Spot the CPU!

(central processing unit)



Spot the GPUs!
(graphics processing unit)

This image is in the public domain

NVIDIA

vs

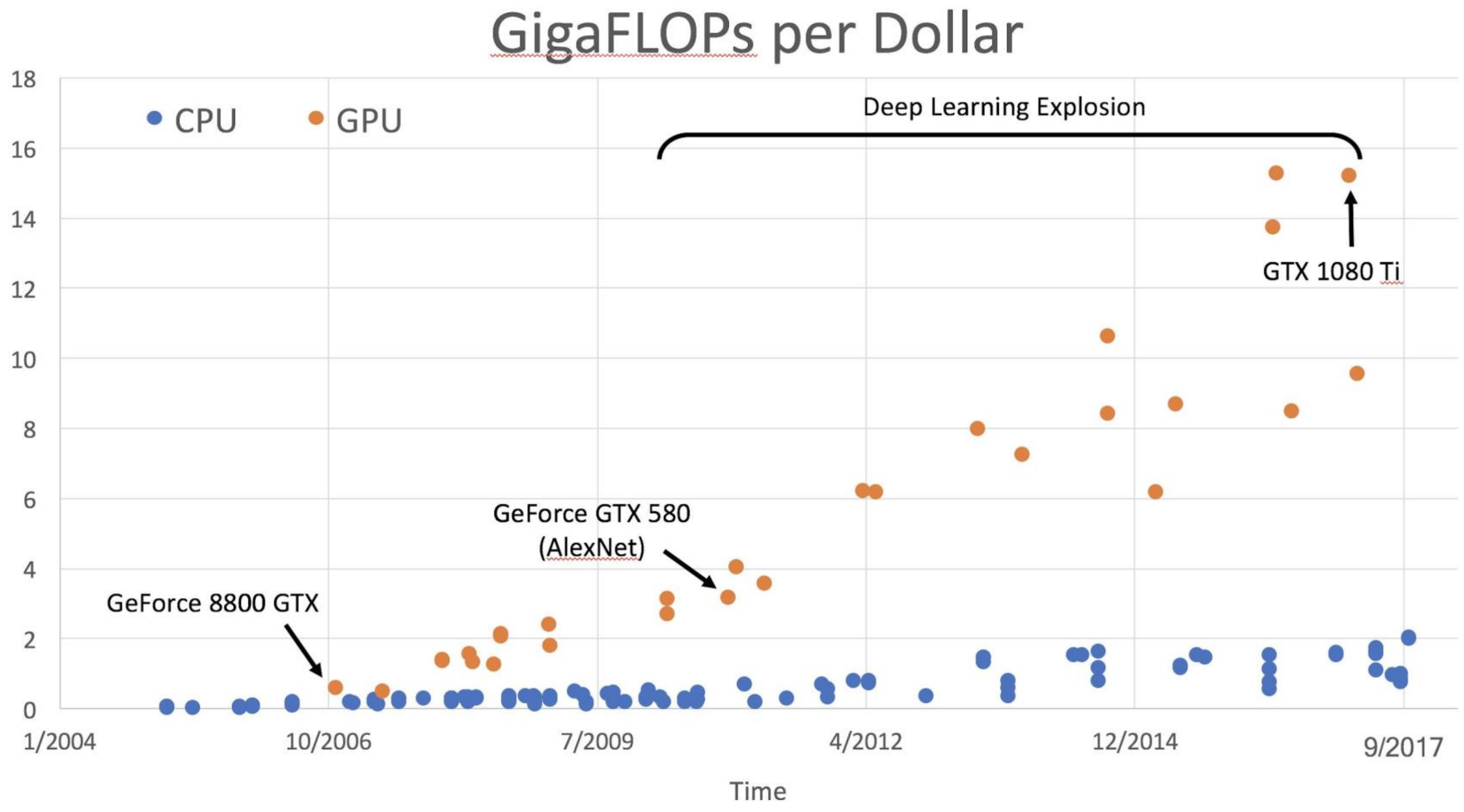
AMD

CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	$699	~11.4 TFLOPs FP32

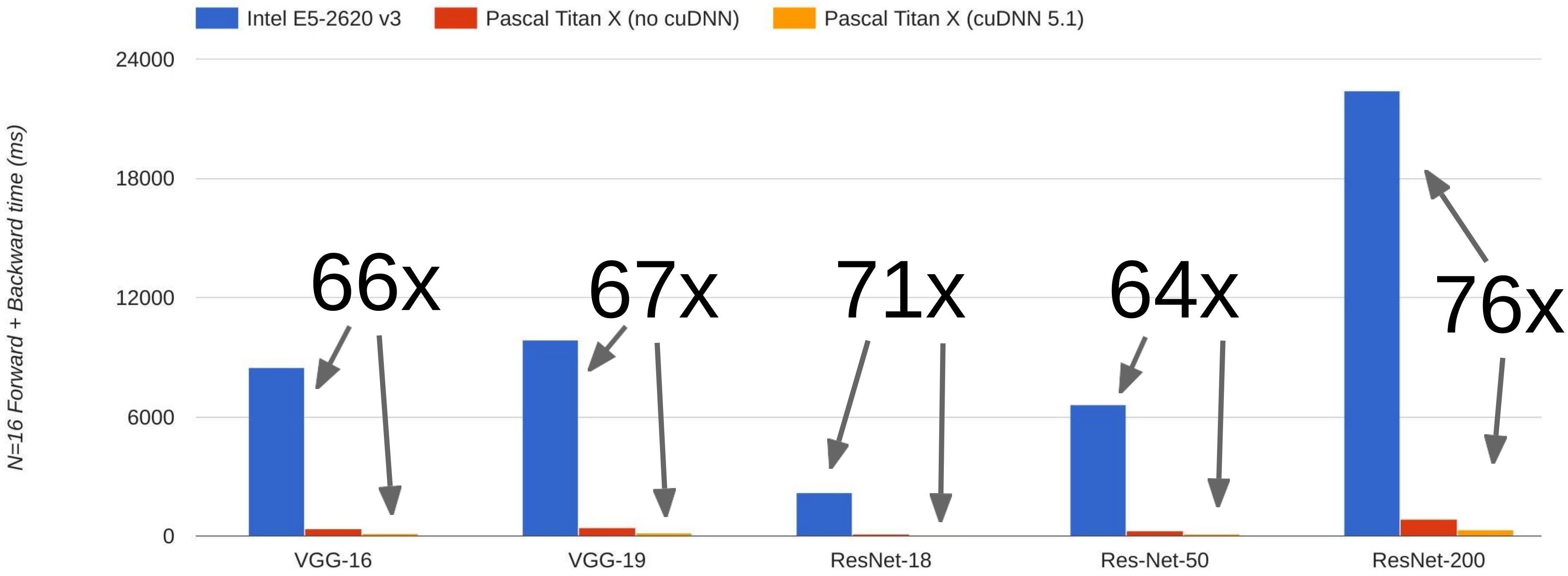
CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

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


GigaFLOPs per Dollar
CPU
GPU
Deep Learning Explosion
GTX 1080 Ti
GeForce GTX 580
(AlexNet)
GeForce 8800 GTX
18
16
14
12
10
8
6
4
2
0
1/2004
10/2006
7/2009
4/2012
12/2014
9/2017
Time

CPU vs GPU in practice

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



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	$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	$699	~11.4 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	$6.50 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

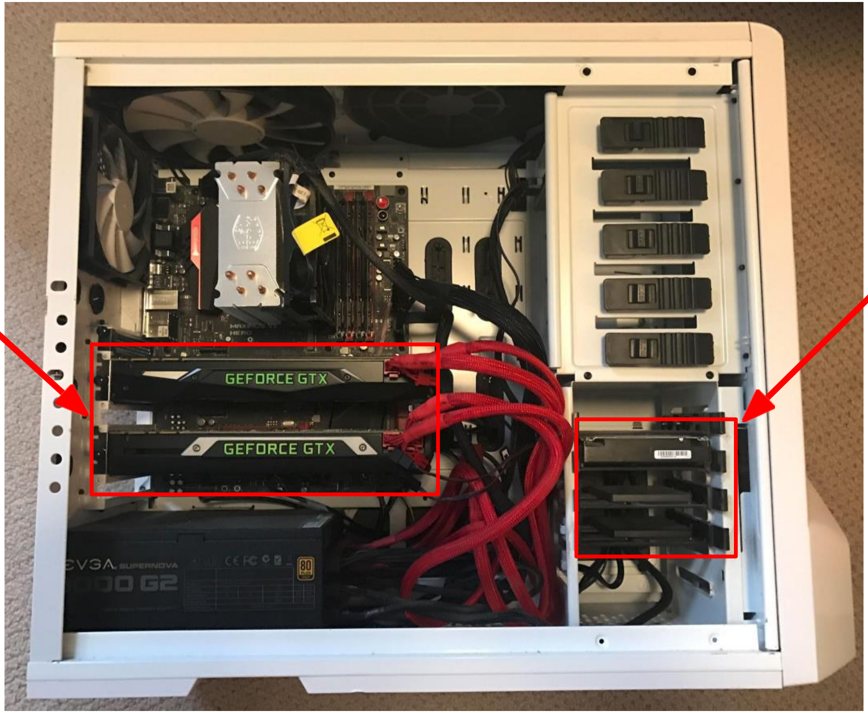
TPU: Specialized hardware for deep learning

Programming GPUs

- CUDA (NVIDIA only)
  - Write C-like code that runs directly on the GPU
  - Optimized APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
  - Similar to CUDA, but runs on anything
  - Usually slower on NVIDIA hardware
- HIP https://github.com/ROCm-Developer-Tools/HIP
  - New project that automatically converts CUDA code to something that can run on AMD GPUs
- Udacity: Intro to Parallel Programming https://www.udacity.com/course/cs344
  - For deep learning just use existing libraries

CPU / GPU Communication

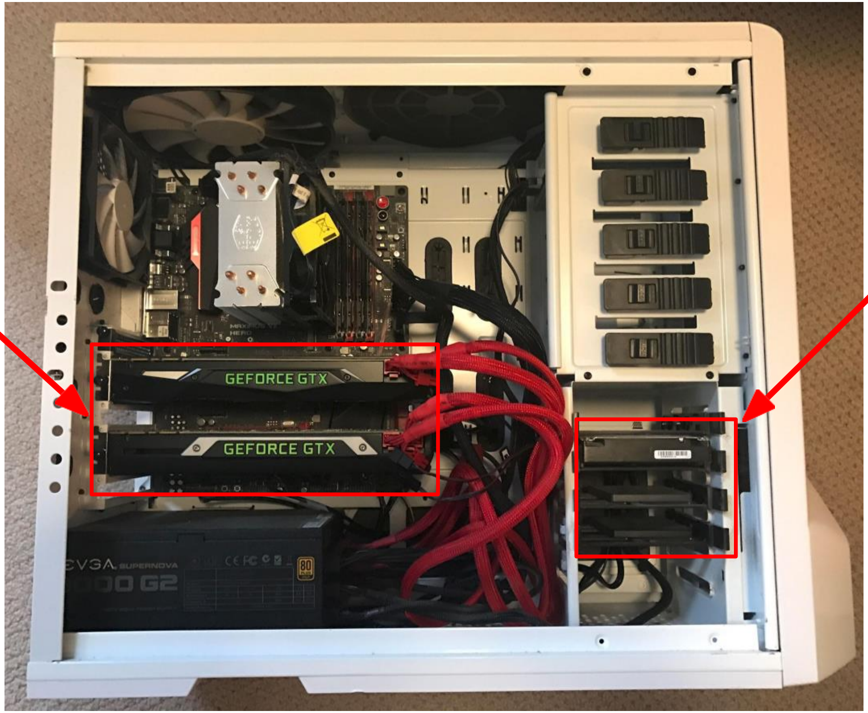
Model
is here



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

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

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