10-605/10-805: Machine Learning with Large Datasets

Fall 2022

(Deep) Neural Networks, DL Frameworks, Hardware

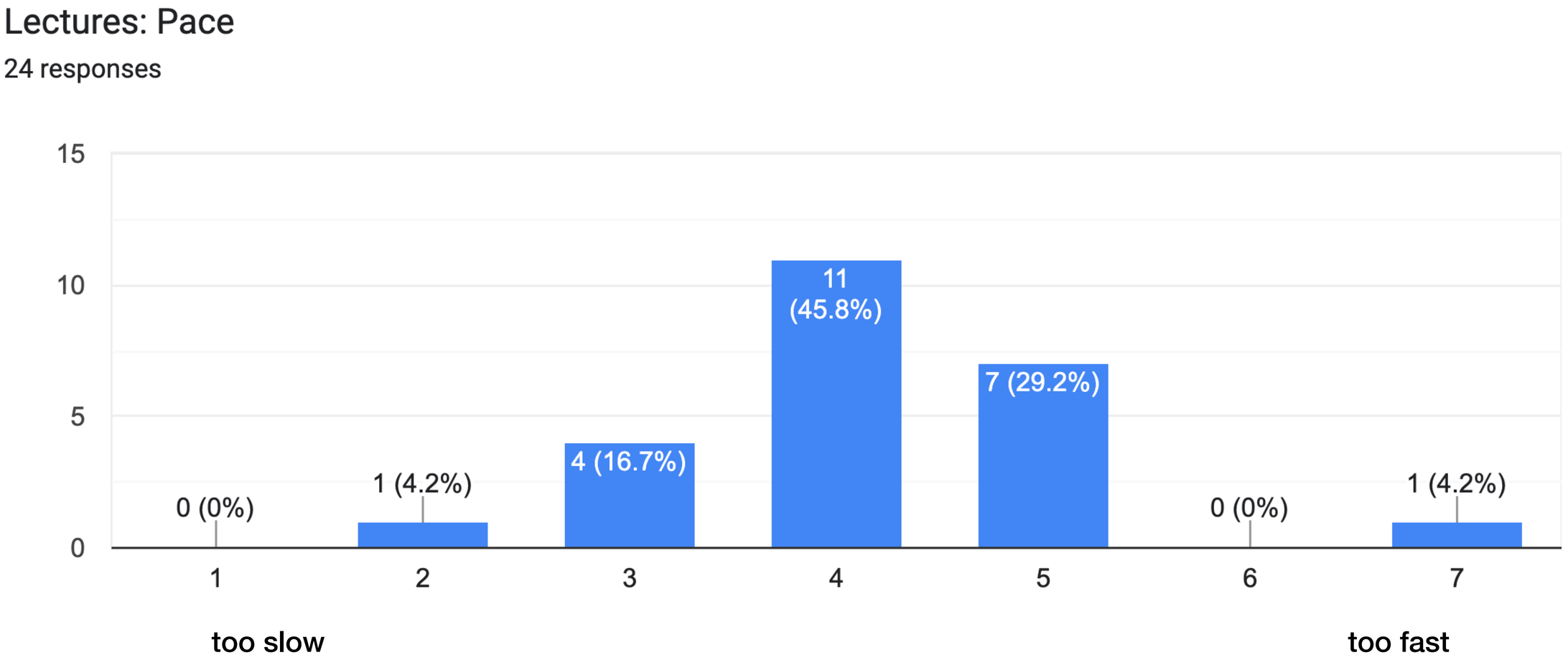
Announcements

- Slight change in schedule to cover material required for HW5 in sufficient time while still providing ample time for miniprojects
  - Combining content for Deep learning, DL frameworks, and Hardware into a single lecture (today's lecture)
  - Moving subsequent lectures up one
  - Moving the Guest Lecture hosted by Krishna Rangasayee to November 17th
  - The lecture on December 1st is TBD
- See website for details

Announcements

- HW4a due on Thursday (10/27)
  - please start ASAP if you haven’t already!
  - revisit HW4A tutorial (recitation on 10/14)
- Friday’s recitation is a TensorFlow tutorial (recorded — Community Day)
- Mid-semester grades have been entered
  - Included HW1 - HW3 and Exam 1

Results from course survey

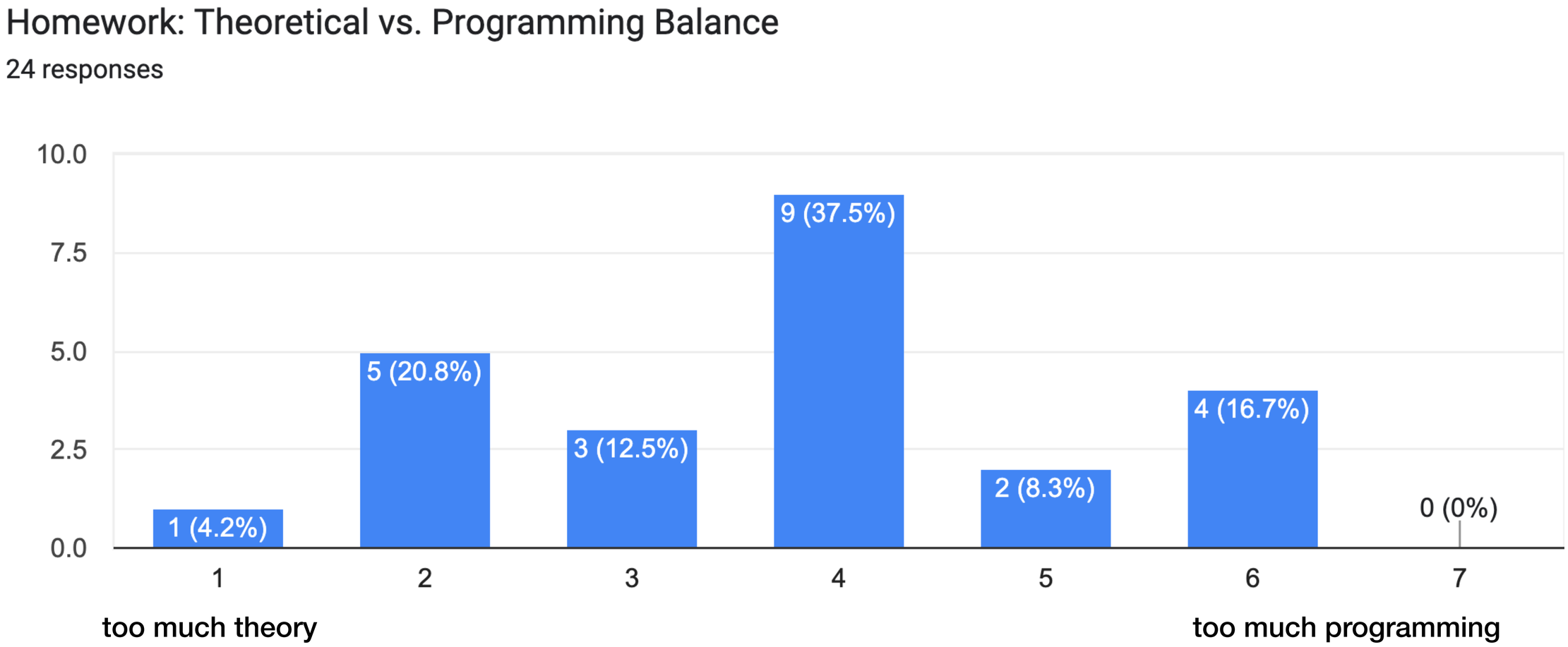


Results from course survey

Would you like to see more or less of a mathematical focus in the course?

24 responses

Results from course survey



Results from course survey

- *“The office hours for this course are wonderful. The TAs are very skilled instructors and know the materials very well.*

  >> Great to hear!

- *“Please keep half OH online and half in-person.”*

  >> This is the plan!

- *“Please make the programming assignments a bit less hand-held.”*

  >> remaining HWs will naturally be more open-ended

Key course topics

Data preparation
- Data cleaning
- Data summarization
- Visualization
- Dimensionality reduction

Training
- Distributed ML
- Large-scale optimization
- Scalable deep learning
- Efficient data structures
- Hyperparameter tuning

Inference
- Hardware for ML
- Techniques for low-latency inference (compression, pruning, distillation)

Infrastructure / Frameworks
- Apache Spark
- AWS / GCP / Azure
- TensorFlow

Advanced topics
- Federated learning
- Neural architecture search
- Productionizing ML

Key course topics

Data preparation
- Data cleaning
- Data summarization
- Visualization
- Dimensionality reduction

Training
- Distributed ML
- Large-scale optimization
- **Scalable deep learning**
- Efficient data structures
- Hyperparameter tuning

Inference
- Hardware for ML
- Techniques for low-latency inference (compression, pruning, distillation)

Infrastructure / Frameworks
- Apache Spark
- AWS / GCP / Azure
- **TensorFlow**

Advanced topics
- Federated learning
- Neural architecture search
- Productionizing ML

Outline

1. Neural networks
2. Backprop
3. ML Frameworks
4. DL Hardware

RECALL

Empirical risk minimization

$$\hat{f}_n = \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

- A popular approach in supervised ML
- Given a loss ℓ and data (x_1, y_1), … (x_n, y_n), we estimate a predictor *f* by minimizing the *empirical risk*
- We typically restrict this predictor to lie in some class, *F*
  - Could reflect our prior knowledge about the task
  - Or may be for computational convenience

Question: how should we select our function class, *F*, and our loss function, ℓ ??

RECALL

Previous assumption: linear models

$$\hat{f}_n = \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

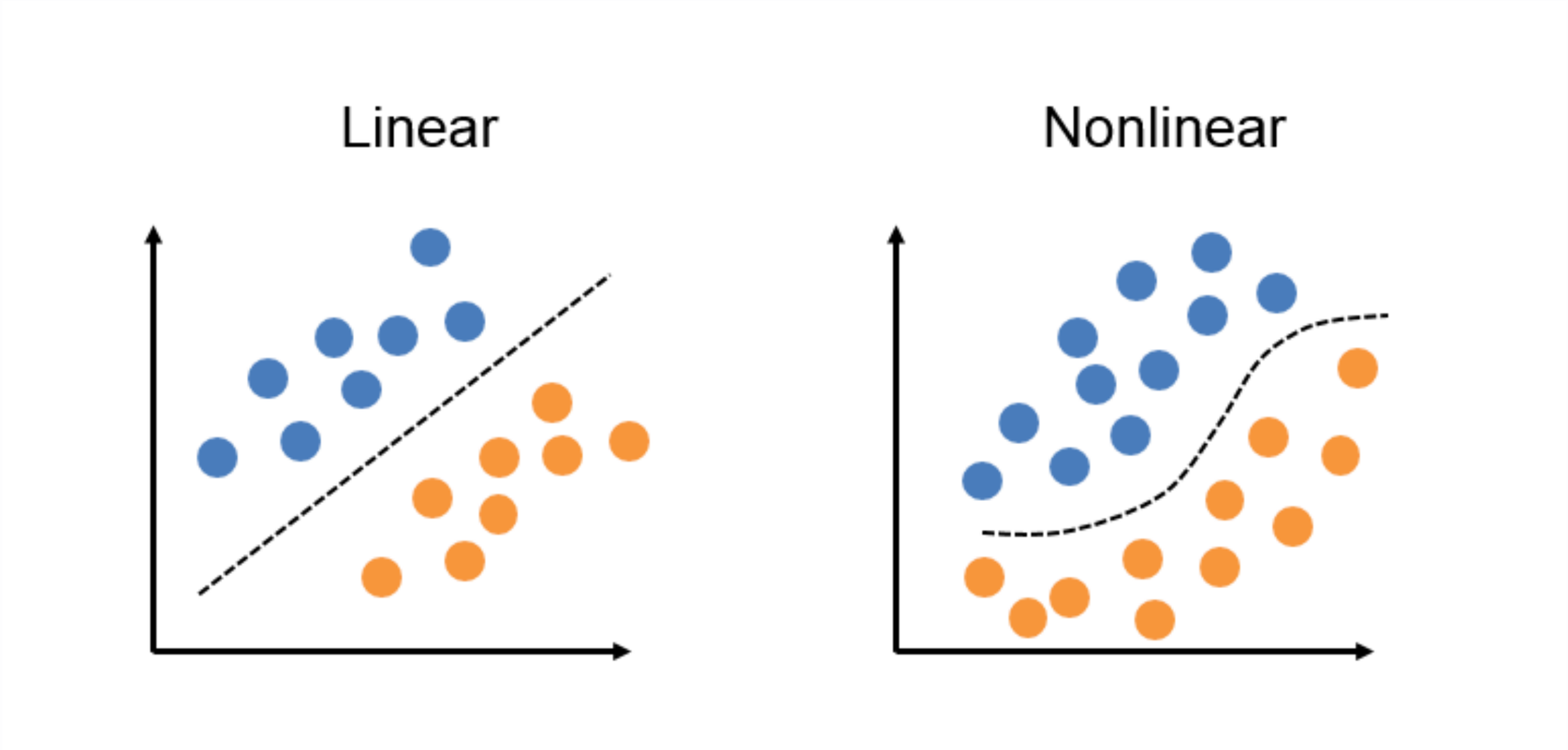
Assume: $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$

Losses:

- Square loss: linear regression $\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (\mathbf{w}^\top \mathbf{x}_i - y_i)^2$
- Log loss: logistic regression $\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \ell_{log}(y_i \cdot \mathbf{w}^\top \mathbf{x}_i)$
- Hinge loss: SVMs $\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y_i \cdot \mathbf{w}^\top \mathbf{x}_i\}$

ISSUE

Not all data is linear …



How can we learn more complicated hypotheses?

ONE APPROACH

Neural networks

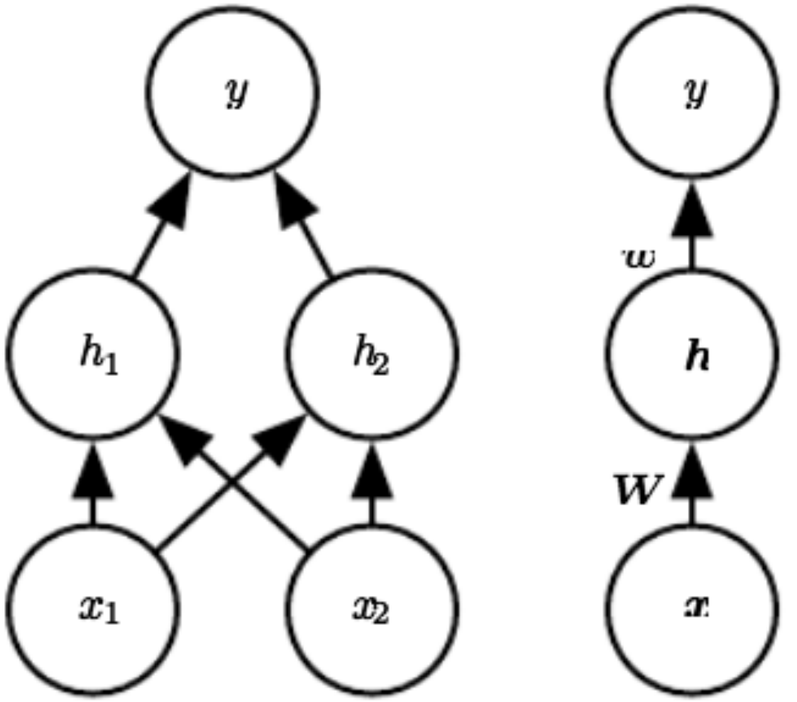
Goal: apply a nonlinear transformation to our input data: $f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}; \theta)$

Neural networks provide a way to learn such a transformation

Simple approach: feedforward neural networks

- E.g., suppose we have: $\phi(\mathbf{x}; \mathbf{W}) = \sigma(\mathbf{W}^\top \mathbf{x})$
- Then: $f(\mathbf{x}) = \mathbf{w}^\top \sigma(\mathbf{W}^\top \mathbf{x})$

$\sigma(\cdot)$ is referred to as an *activation function*



an example of a feedforward network with *one* hidden layer containing *two* units

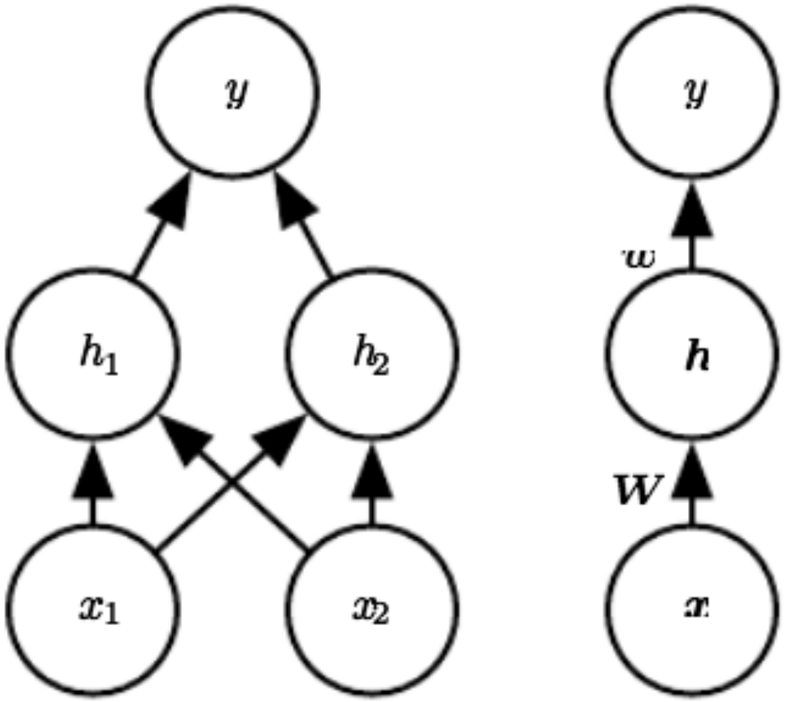
A NOTE ON

Activation functions

Simple approach: feedforward neural networks

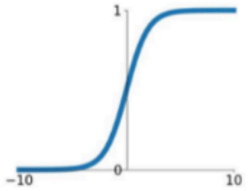
- E.g., suppose we have: $\phi(\mathbf{x}; \mathbf{W}) = \sigma(\mathbf{W}^\top \mathbf{x})$
- Then: $f(\mathbf{x}) = \mathbf{w}^\top \sigma(\mathbf{W}^\top \mathbf{x})$

$\sigma(\cdot)$ is referred to as an *activation function*



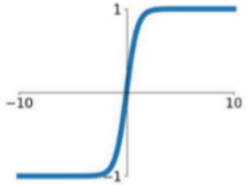
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



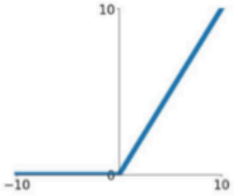
tanh

$\tanh(x)$



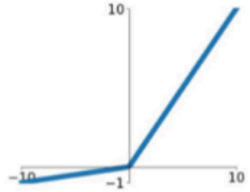
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

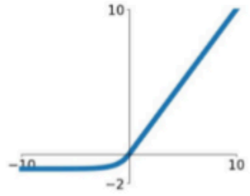


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



some examples of activation functions

typically applied element-wise, i.e.

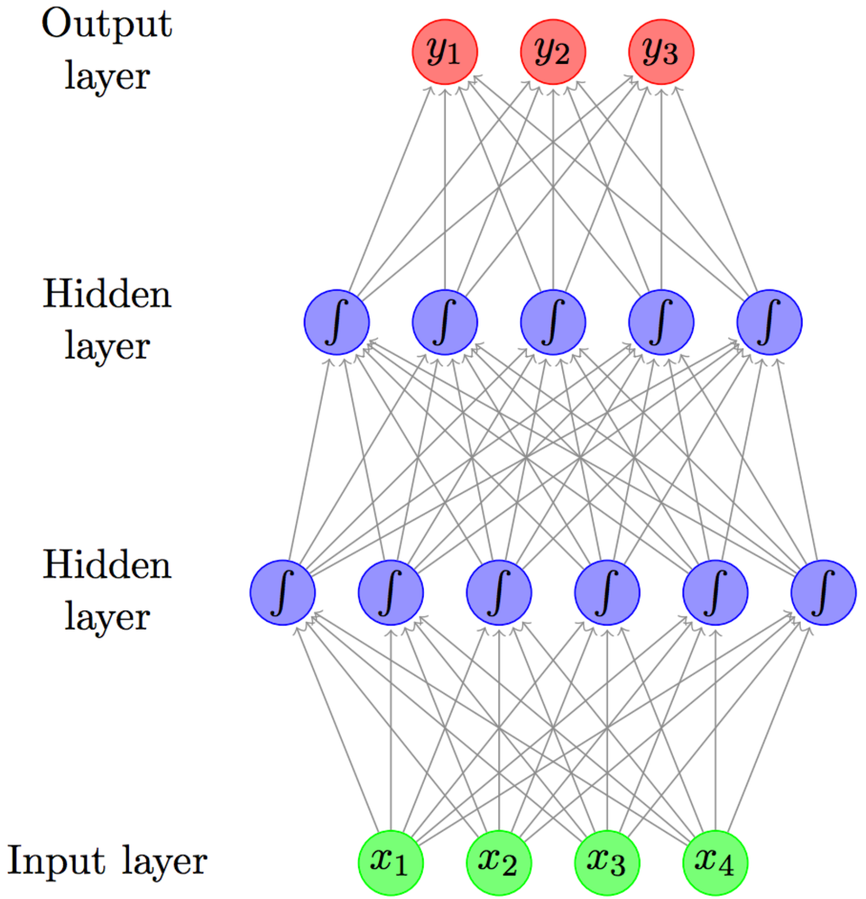
$$\sigma(\mathbf{x})_i = \sigma(\mathbf{x}_i)$$

MORE GENERALLY

Feedforward neural networks

We can consider adding L hidden layers, each with their own activation functions σ_l and corresponding weights $\mathbf{W}_l$

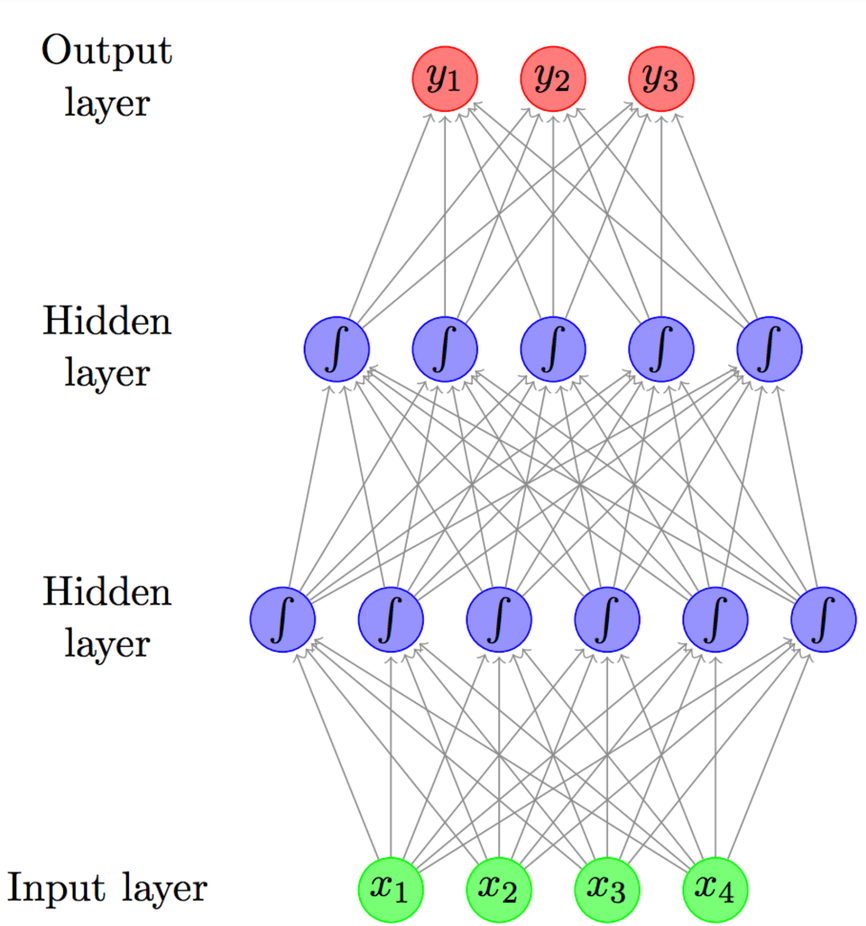
Using this layering technique, the resulting model becomes:



$$f(\mathbf{x}) = \mathbf{w}^\top \sigma_L(\mathbf{W}_L^\top(\sigma_{L-1}(\mathbf{W}_{L-1}^\top \ldots \sigma_2(\mathbf{W}_2^\top \sigma_1(\mathbf{W}_1^\top \mathbf{x}))\ldots)$$

MORE GENERALLY

Feedforward neural networks



How to pick the best architecture? Must select:

- Number of layers (depth)
- Number of nodes per layer (width)
- Activation functions (nonlinearities)
- Loss function

Other variants:

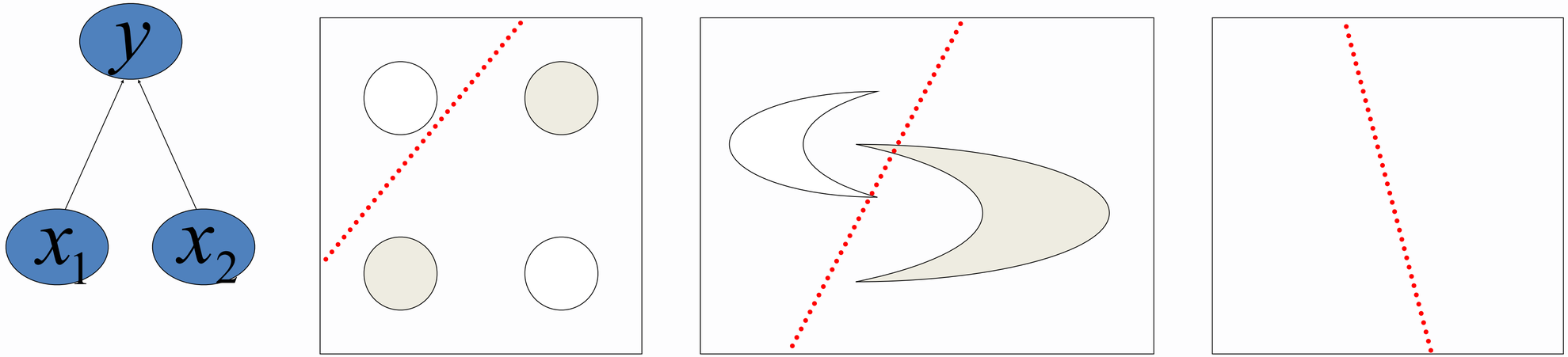
- CNNs: use convolution instead of standard matrix multiply
- RNNs: outputs are fed back into network

WHY WOULD WE DO THIS?

Feedforward neural networks

Neural networks can be very expressive, especially as you add more layers (i.e., ‘deep’ neural networks)

0 hidden layers: linear classifier



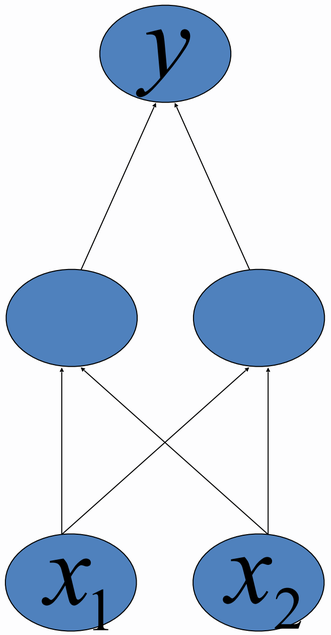
Credit: Matt Gormley

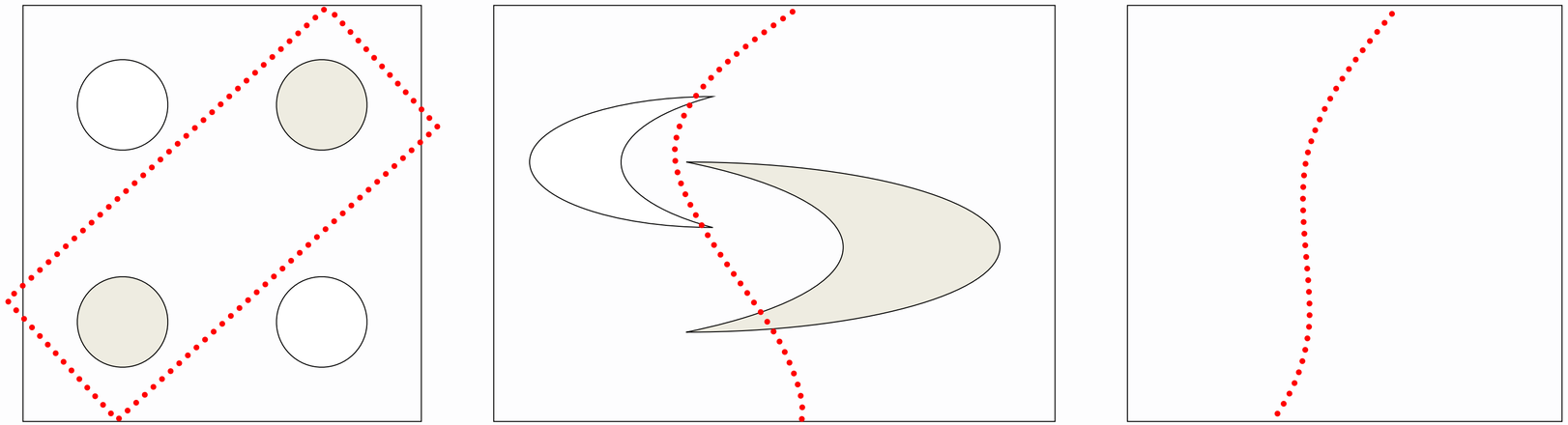
WHY WOULD WE DO THIS?

Feedforward neural networks

Neural networks can be very expressive, especially as you add more layers (i.e., ‘deep’ neural networks)

1 hidden layer: boundary of convex region (open or closed)



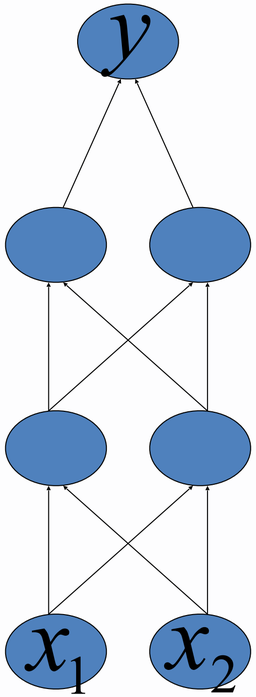


Credit: Matt Gormley

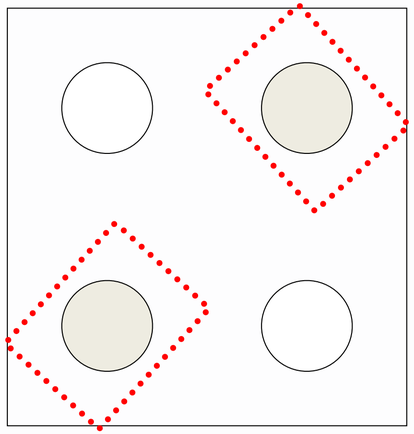
WHY WOULD WE DO THIS?

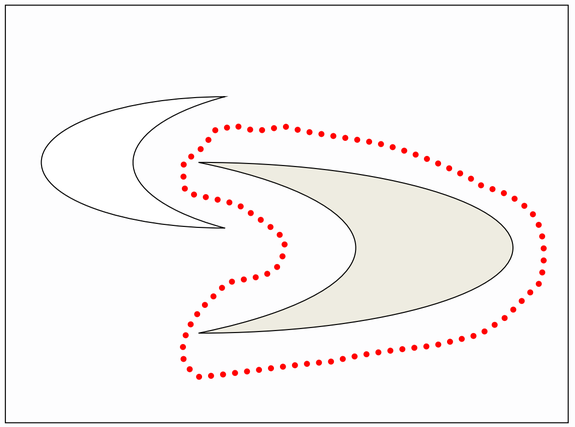
Feedforward neural networks

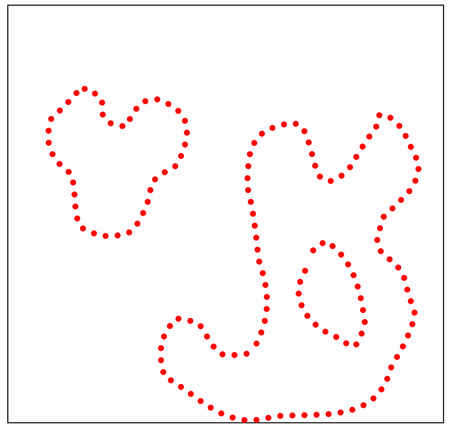
Neural networks can be very expressive, especially as you add more layers (i.e., ‘deep’ neural networks)



2 hidden layers: combinations of convex regions







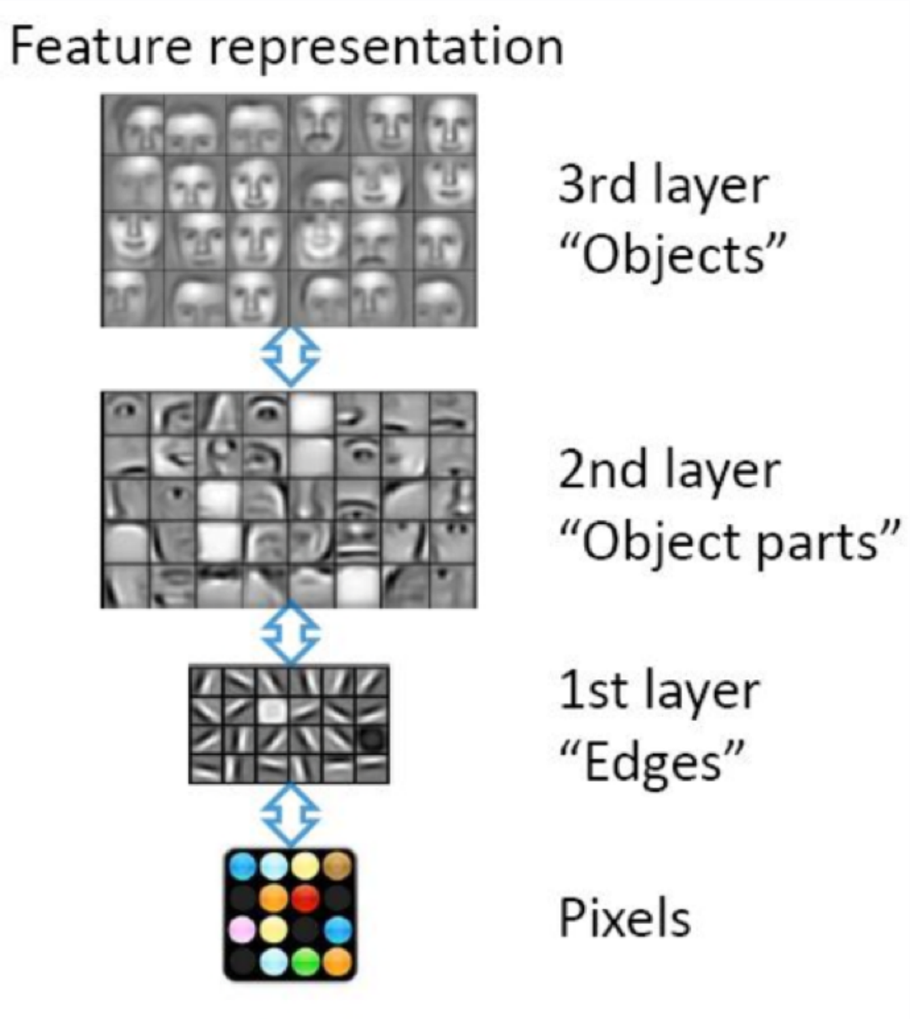
Credit: Matt Gormley

Feedforward neural networks

Neural networks can be very expressive, especially as you add more layers (i.e., 'deep' neural networks)

Pro: Can allow the model to learn a feature representation and the appropriate levels of abstraction from your data

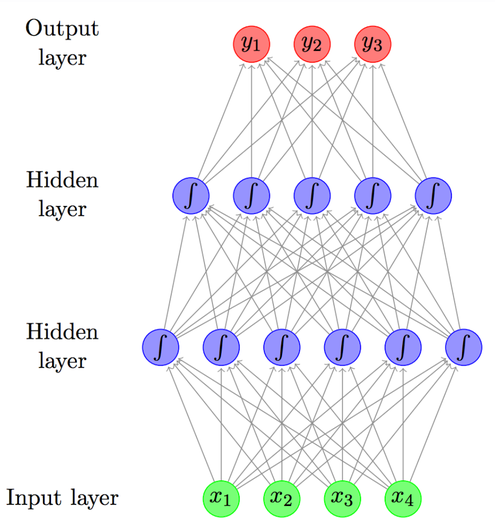
Challenge: this can make DNNs expensive …



Credit: Matt Gormley

COST OF THE 'FORWARD PASS'

Feedforward neural networks



Can express NN model with the following recurrence:

$$\mathbf{o}_0 = \mathbf{x}$$

$$\forall l \in \{1,\ldots,L\},\ \mathbf{h}_l = \mathbf{W}_l^\top \mathbf{o}_{l-1}$$

$$\forall l \in \{1,\ldots,L\},\ \mathbf{o}_l = \sigma_l(\mathbf{h}_l)$$

$$\phi(\mathbf{x}) = \mathbf{o}_L$$

What dominates the cost?

Matrix(-vector) multiplies

O(nm) for n x m matrix, m x 1 vector

MOVING FORWARD

What makes deep learning expensive?

Training requires lots of computation

- GPUs can help with matrix computations [*more on this later in lecture*]
- Can use advanced iterative optimization methods [*more on this Thursday*]
- Can parallelize training [*more on this in a few weeks*]

Hyperparameter tuning and neural architecture search (NAS) make this worse

- Lots of knobs to tune! [*more on this in a few weeks*]

Resulting models can be large!

- Can be expensive to store model, perform inference [*more on this in a few weeks*]

Outline

1. Neural networks
2. Backprop
3. ML Frameworks
4. DL Hardware

How to train DNNs?

Same idea as before: can use first-order optimization methods, e.g., SGD

Issue: need to compute the gradient …

How do we do this? Need to differentiate a composition of functions

$$f(\mathbf{x}) = \mathbf{w}^\top \sigma_L(\mathbf{W}_L^\top(\sigma_{L-1}(\mathbf{W}_{L-1}^\top \dots \sigma_2(\mathbf{W}_2^\top \sigma_1(\mathbf{W}_1^\top \mathbf{x}))\dots)$$

^ will have to use the *chain rule*

RECALL

Chain rule

The chain rule is used to differentiate composite functions, e.g.:

$$\frac{d}{dx} f(g(h(x))) = f'(g(h(x))) \cdot g'(h(x)) \cdot h'(x)$$

We can use this rule to calculate the gradients for neural networks

However, this expression doesn't provide an *algorithm* for computing the gradient efficiently

How can we compute the chain rule efficiently?

RECALL

Chain rule

$$\frac{d}{dx} f(g(h(x))) = f'(g(h(x))) \cdot g'(h(x)) \cdot h'(x)$$

Simple example: suppose we input the above equation into python …

```
def grad_fgh(x):
        return grad_f(g(h(x))) * grad_g(h(x)) * grad_h(x)
```

Do you see any inefficiencies? *h(x) is calculated twice …*

Instead, consider:

```
def better_grad_fgh(x):
        hx = h(x)
        return grad_f(g(hx)) * grad_g(hx) * grad_h(x)
```

Credit: Chris De Sa

Chain rule

How much computation can this actually save?

Suppose we have: $f_k(f_{k-1}(\cdots f_2(f_1(x))\cdots))$

Using the chain rule, the derivative is:

$$\frac{d}{dx} f_k(f_{k-1}(\cdots f_2(f_1(x))\cdots))$$

$$= f_k'(f_{k-1}(\cdots f_2(f_1(x))\cdots)) \cdot f_{k-1}'(f_{k-2}(\cdots f_2(f_1(x))\cdots))\cdots f_2'(f_1(x)) \cdot f_1'(x)$$

Suppose f_i and f_i' take $O(1)$ time to compute

What's the complexity for naively computing this derivative? $\sum_{i=1}^{k} i \rightarrow O(k^2)$

What if we avoid redundant computation (as in previous slide)? $k + (k-1) \rightarrow O(k)$

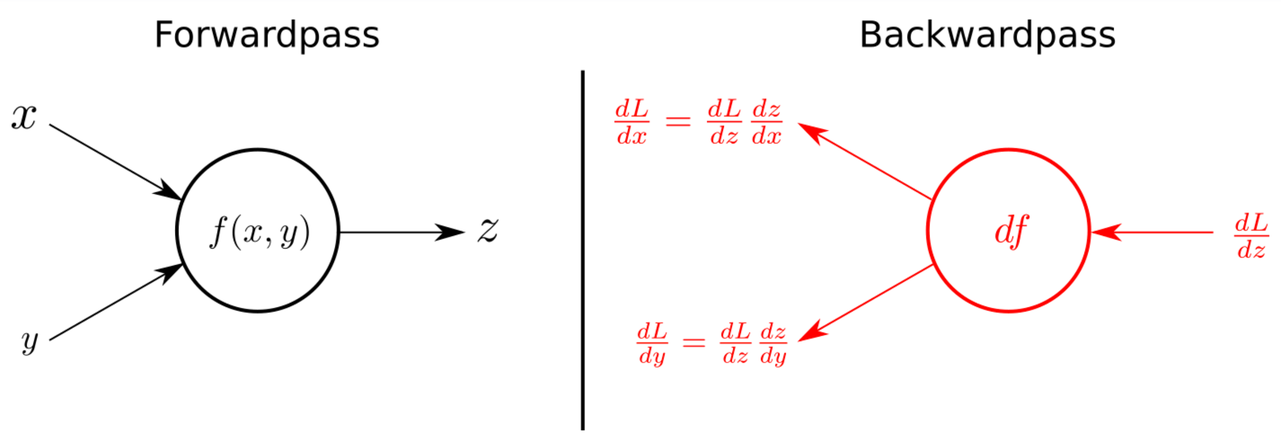
Backpropagation

This is the main idea / cost savings in “backpropagation”, which efficiently computes the gradient of a feedforward neural network

Two main steps:

- *forward pass*: traverse through the network normally, as if we were computing the function value, and save all intermediate values
- *backward pass*: proceed backwards through the network, computing gradients of the function value with respect to intermediates

You will gain hands on experience in HW5 …



Additional Resources

Feedforward neural networks & backprop

- Deep Learning Book, Chapter 6: https://www.deeplearningbook.org/

Autodiff

- William Cohen Notes: http://www.cs.cmu.edu/~wcohen/10-605/notes/autodiff.pdf
- Blog post: https://justindomke.wordpress.com/2009/02/17/automatic-differentiation-the-most-criminally-underused-tool-in-the-potential-machine-learning-toolbox/

Outline

1. Neural networks
2. Backprop
3. ML Frameworks
4. DL Hardware

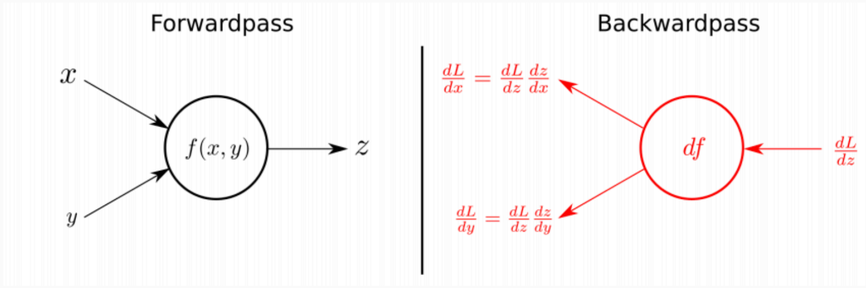
TRAINING DNNS

Backpropagation

This is the main idea / cost savings in “backpropagation”, which efficiently computes the gradient of a feedforward neural network

Two main steps:

- *forward pass*: traverse through the network normally, as if we were computing the function value, and save all intermediate values
- *backward pass*: proceed backwards through the network, computing gradients of the function value with respect to intermediates



🤔

Recall backpropagation

Slides Credit: Heather Miller

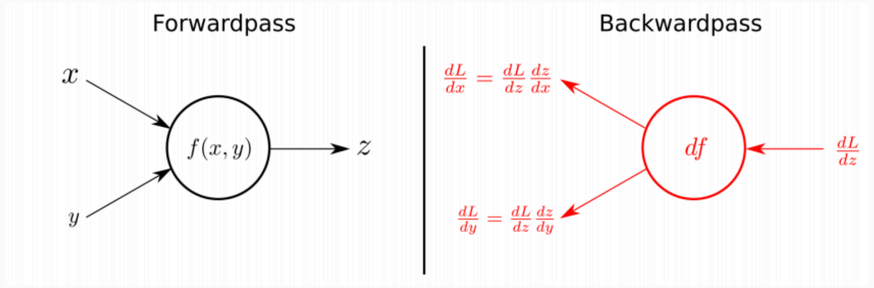
TRAINING DNNS

Backpropagation

This is the main idea / cost savings in “backpropagation”, which efficiently computes the gradient of a feedforward neural network

Two main steps:

- *forward pass*: traverse through the network normally, as if we were computing the function value, and save all intermediate values
- *backward pass*: proceed backwards through the network, computing gradients of the function value with respect to intermediates



IMPLEMENTING THIS...

It would be a pain to have to implement this all from scratch for a network of potentially 1000s of units!

THOUGHT EXERCISE

Imagine designing a ML system from scratch

It's **easy to start with basic SGD** in C++

Implement objective function, gradient function, then make a loop

But there's **so much more to be done** with our C++ program

- *Need to manually code a **step size scheme***
- *Need to modify code to add **mini-batching***
- *Need to completely rewrite code to run in **parallel** or with **low-precision***
- *Impossible to get it to run on a **GPU** or on an **ASIC***
- *And at each step we have to **debug** and **validate** the program*

Clearly, in a software engineering sense, we should be striving for reuse!

SOLUTION

Machine Learning Frameworks

Machine Learning frameworks have exploded in popularity in an effort to…

Goal: Make ML easier

- From a software engineering perspective
- Make the computations more reliable, debuggable, and robust

Goal: Make ML scalable

- To large datasets running on distributed heterogeneous hardware

Goal: Make ML accessible

- So that even people who aren't ***ML systems experts*** can get good performance

SOLUTION

Machine Learning Frameworks

Machine Learning frameworks **make it so that we don't have to reinvent the wheel** each time we would like to experiment with a technique!

Goal: Make ML easier

- From a software engineering perspective
- Make the computations more reliable, debuggable, and robust

Goal: Make ML scalable

- To large datasets running on distributed heterogeneous hardware

Goal: Make ML accessible

- So that even people who aren't ***ML systems experts*** can get good performance

SOLUTION

Machine Learning Frameworks: Deep Learning

This extends to **deep learning** frameworks

Why we have frameworks for deep learning:

1. Allow us to quickly prototype ideas
2. Automatically compute gradients
3. Run efficiently on CPUs/GPUs/TPUs

Machine Learning frameworks **make it so that we don't have to reinvent the wheel** each time we would like to experiment with a technique!

QUICK ASIDE: CONTEXT

Kinds of ML frameworks

There are many *kinds of* **reusable** frameworks that are useful for machine learning:

General machine learning frameworks

- *Goal: make a wide range of ML workloads and applications easy for users*

General big data processing frameworks

- *Focus: computing large-scale parallel operations quickly*
- *Typically has machine learning as a major, but not the only, application*

Deep learning frameworks

- *Focus: fast scalable backpropagation and inference*
- *Although typically supports other applications as well*

QUICK ASIDE: CONTEXT

A quick sampling of some such ML frameworks

General machine learning frameworks

scikit-learn

NumPy

SciPy

Together provide a great ecosystem for prototyping systems on single machine / CPU!

QUICK ASIDE: CONTEXT

A quick sampling of some such ML frameworks

General machine learning frameworks

Everything mentioned so far runs on top of basic linear algebra operations! Those come in frameworks too!

BLAS: Basic Linear Algebra Subroutines

- Also has support on GPUs with NVIDIA cuBLAS

LAPACK: Linear Algebra PACKage

QUICK ASIDE: CONTEXT

A quick sampling of some such ML frameworks

General big data processing frameworks

MapReduce/Hadoop

- The first widely-used framework for distributed machine learning
- Mostly been superseded by other frameworks

QUICK ASIDE: CONTEXT

A quick sampling of some such ML frameworks

General big data processing frameworks

MapReduce/Hadoop 

Apache Spark 

- Popular cluster-computing framework, 100x faster than Hadoop

Apache Spark MLlib

- Scalable machine learning library built on top of Spark
- Supports most of the same algorithms scikit-learn supports
  - *Not primarily a deep learning library*
- Major benefit: interaction with other processing in Spark

QUICK ASIDE: CONTEXT

A quick sampling of some such ML frameworks

Deep learning frameworks

Caffe

MXNet

- Early DL frameworks

QUICK ASIDE: CONTEXT

A quick sampling of some such ML frameworks

Deep learning frameworks

Caffe

MXNet

TensorFlow

- End-to-end deep learning system, developed by Google Brain
- Framework automatically schedules computations in computation graph on the available resources

QUICK ASIDE: CONTEXT

A quick sampling of some such ML frameworks

Deep learning frameworks

Caffe

MXNet

TensorFlow

PyTorch

- Tensor computation (like numpy) with strong GPU acceleration
- Eager execution (now TensorFlow 2.0 also includes this …)

+ many others … and new frameworks are always being developed!

What DL Frameworks Typically Support

Basic tensor operations

- *Provides the low-level math behind all the algorithms*

Automatic differentiation

- *Used to make it easy to run backprop on any model*

Simple-to-use implementations of **common techniques**

- *Like mini-batching, SVRG, Adam, etc.*
- *automatic hyperparameter optimization*

A GENERAL IDEA

Tensors

Tensors can be thought of as a **multidimensional array** (CS perspective) or a **multilinear map** (math perspective).

$$T : \mathbb{R}^{d_1} \times \mathbb{R}^{d_2} \times \cdots \times \mathbb{R}^{d_n} \to \mathbb{R}$$

$T(x_1, x_2, \cdots, x_n)$ is linear in each x_i, with other inputs fixed

n is the order of the tensor.

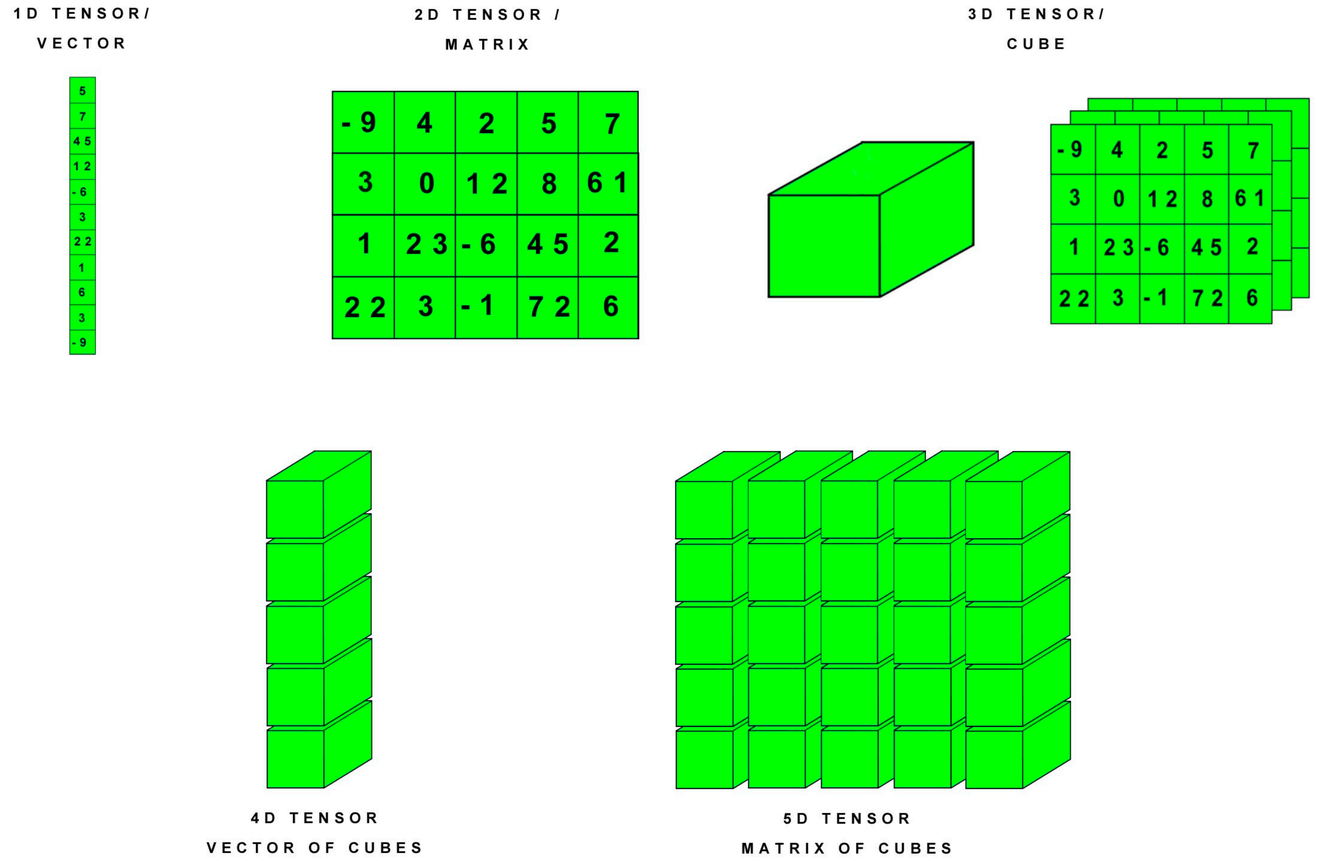
Examples:

- A matrix is a 2nd-order tensor
- A vector is a 1st-order tensor

A GENERAL IDEA

Tensors

Tensors can be thought of as a **multidimensional array** (CS perspective) or a **multilinear map** (math perspective).



EXAMPLES

Tensors

The **CIFAR10 dataset** consists of 60000 32x32 color images,

We can write the training set as a tensor:

$$T_{\text{CIFAR10}} \in \mathbb{R}^{32 \times 32 \times 3 \times 60000}$$

Gradients for deep learning can also be tensors.

Example: fully-connected layer with 100 input and 100 output neurons, and minibatch size b=32

$$G \in \mathbb{R}^{100 \times 100 \times 32}$$

SYSTEMS PERSPECTIVE

Tensors

Benefits of tensors from a systems perspective…

Loads of data parallelism

- Tensors are in some sense the structural embodiment of data parallelism
- Multiple dimensions -> not always obvious which one best to parallelize over

Predictable linear memory access patterns

- Great for locality

Many different ways to organize the computation

- Creates opportunities for frameworks to automatically optimize

Automatic Differentiation

Imagine you write up an SGD algorithm with some objective and some gradient.

You hand-code the computation of the objective and gradient.

What happens when you differentiate incorrectly?

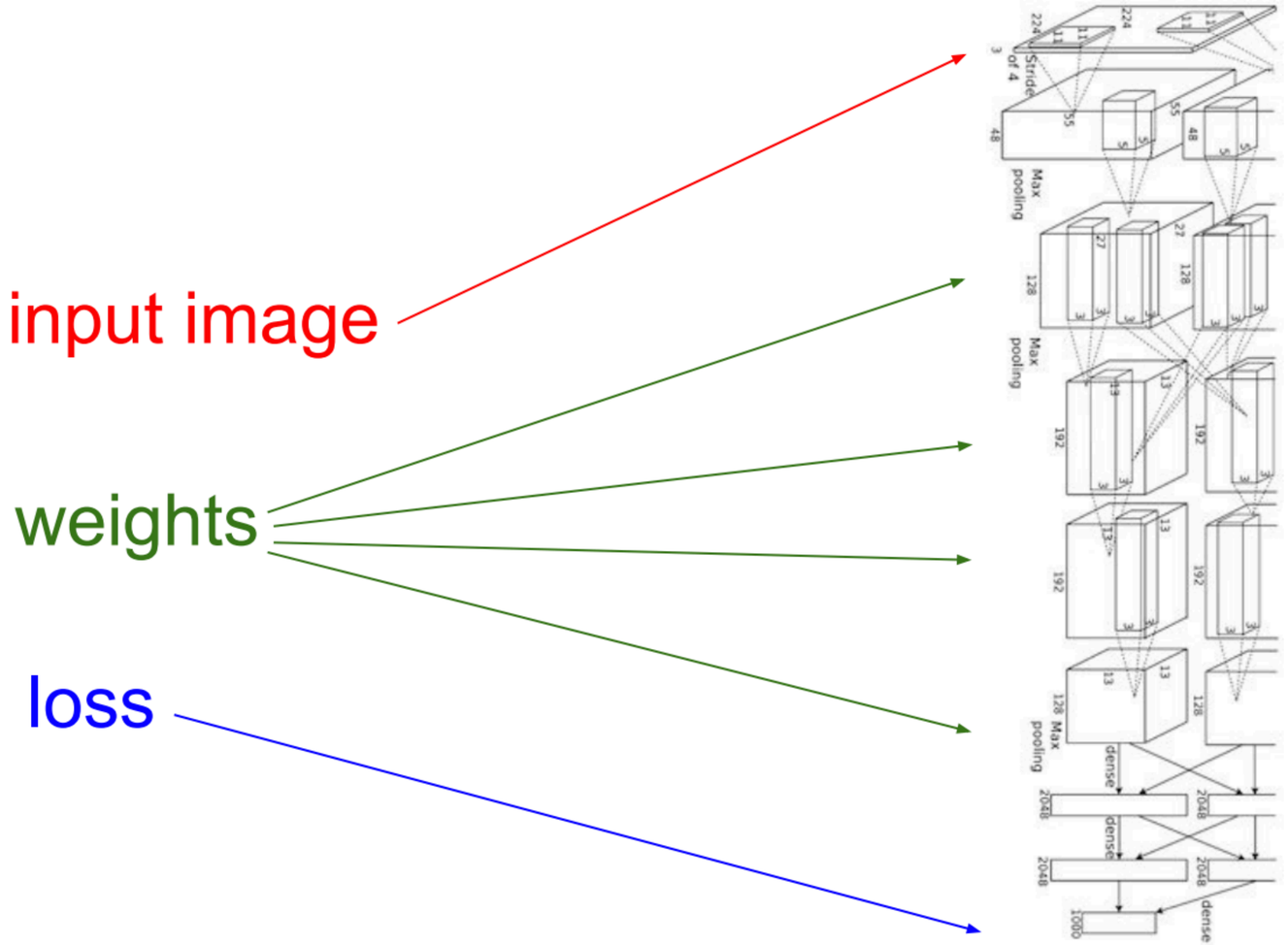
(Everybody will make this mistake eventually if they hand-write objectives. And it's really difficult and increasingly intractable to debug as models become complex)

SOLUTION: generate the gradient automatically from the objective via backprop!

MOTIVATION

Computational Graphs

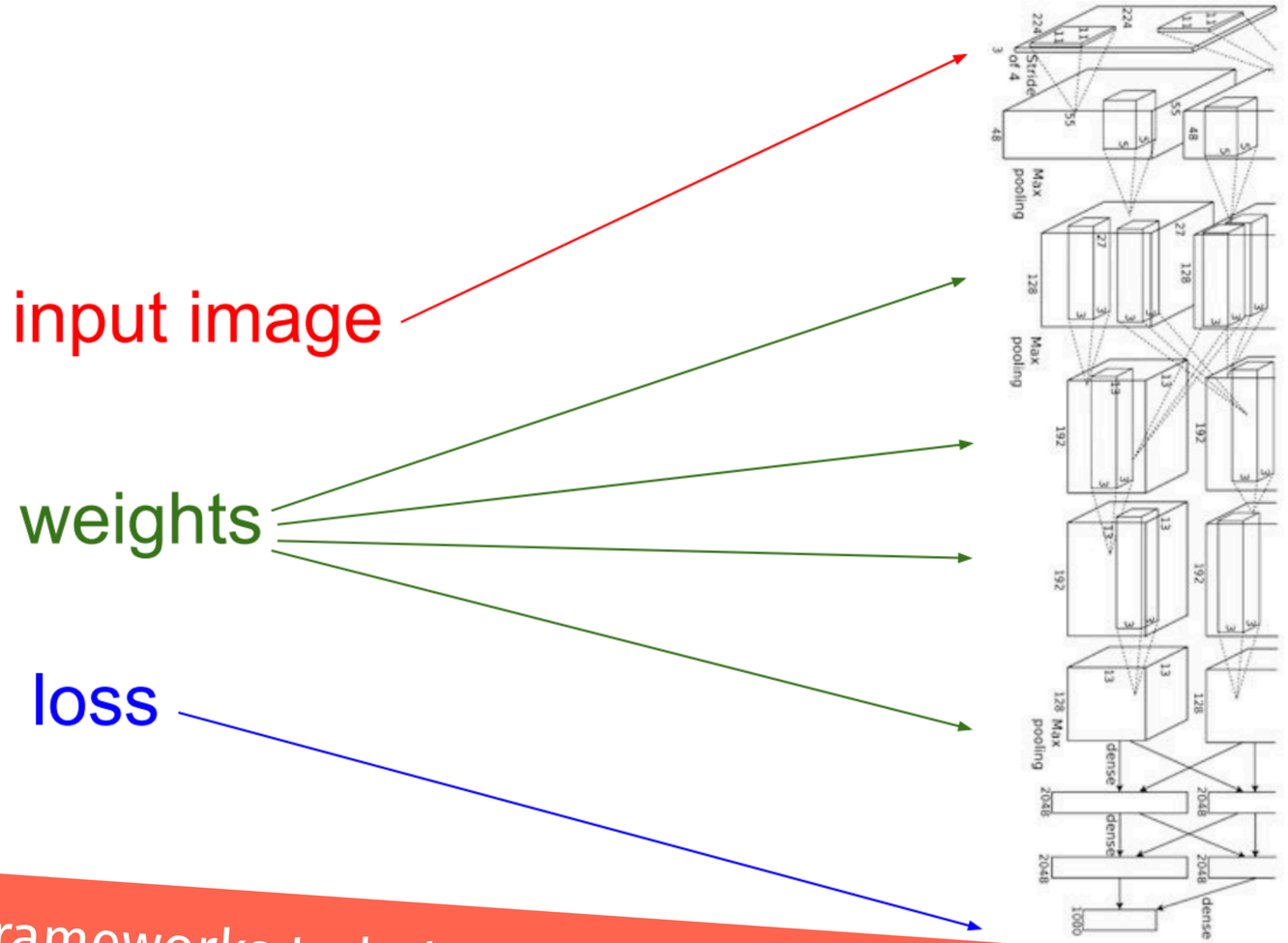
These graph structures can get pretty complex when working with big neural networks.



MOTIVATION

Computational Graphs

These graph structures can get pretty complex when working with big neural networks.



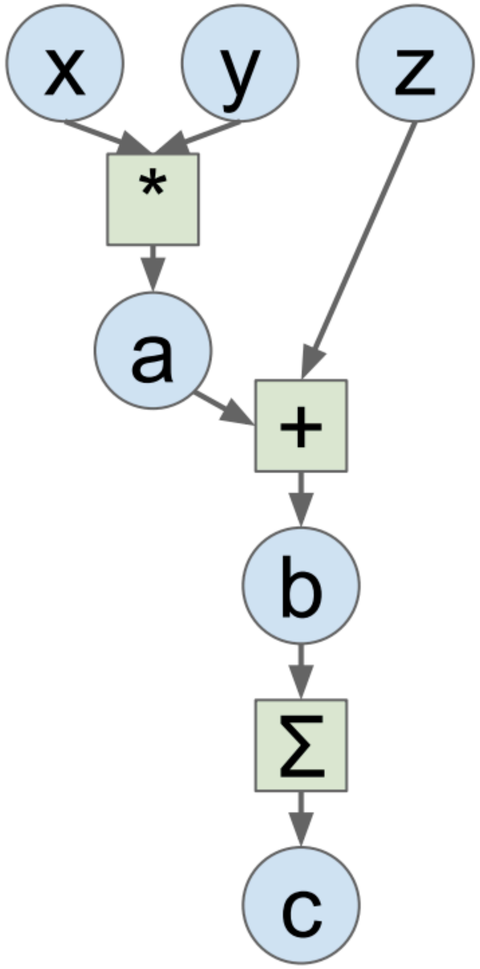
We want Deep Learning frameworks to help us manage this complexity!

MOTIVATION

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)
```



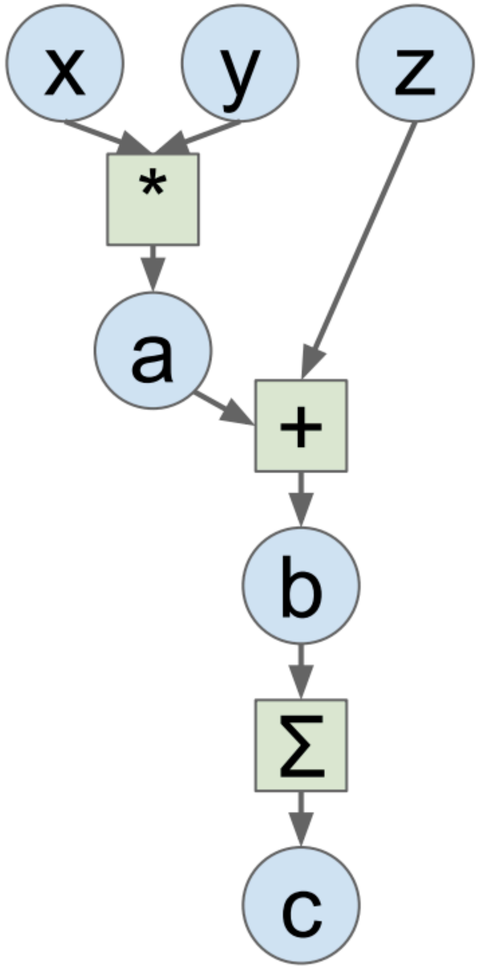
MOTIVATION

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
```



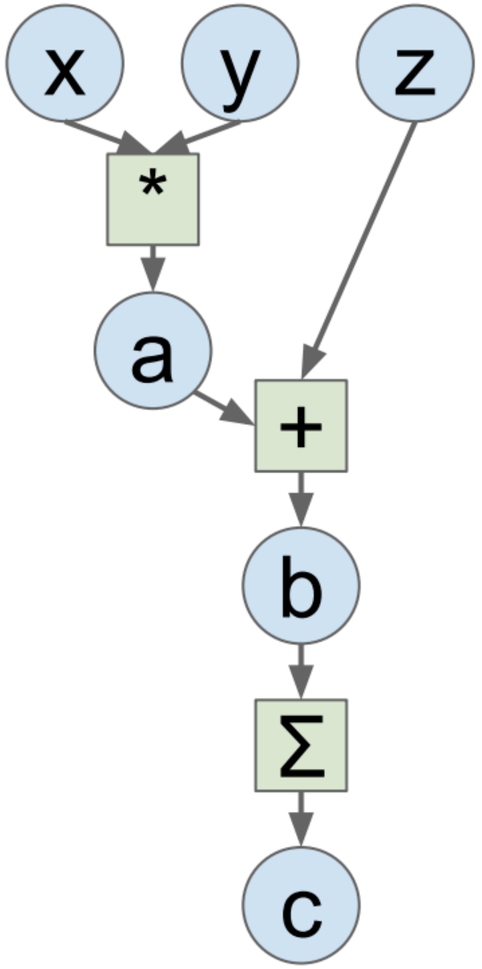
MOTIVATION

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
```



Issues with this:

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

Goal of most DL frameworks is to let you write code in the forward pass that looks like NumPy, AND automatically compute gradients & run on GPU.

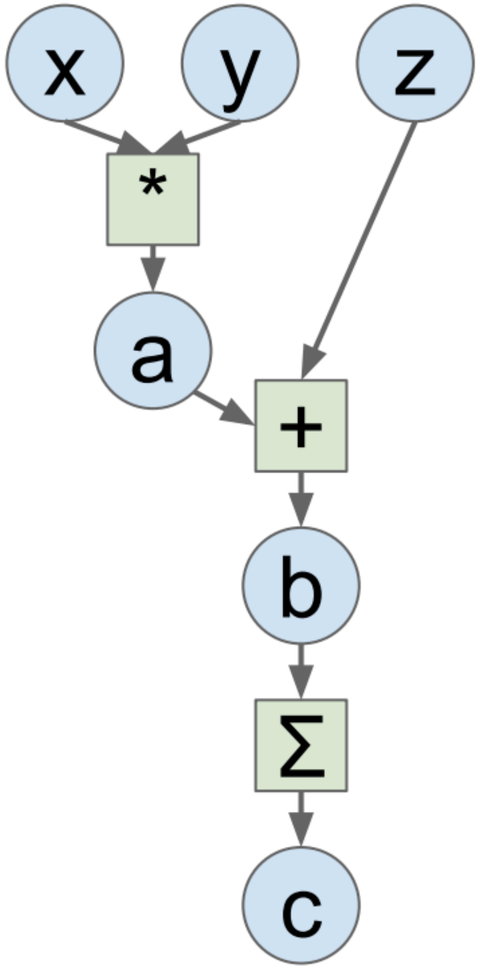
MOTIVATION

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
```

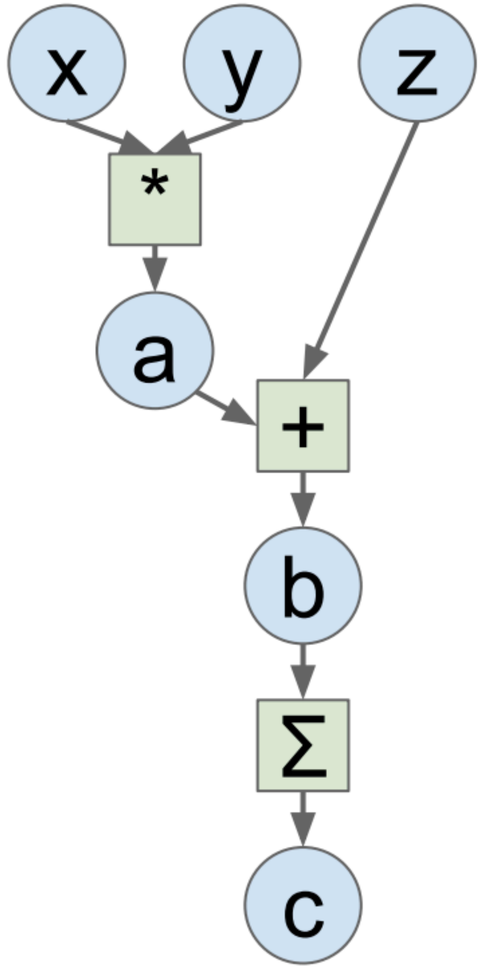


TensorFlow 1.0

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

MOTIVATION

Computational Graphs



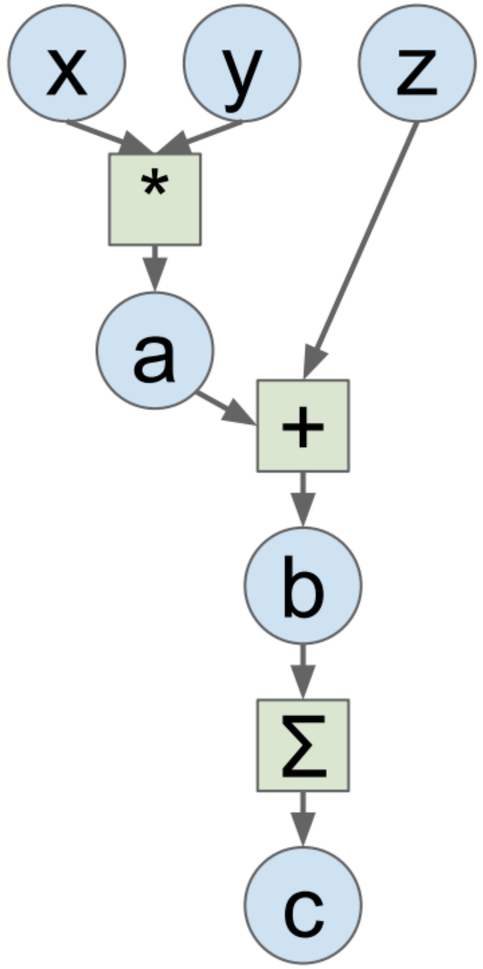
Forward pass

TensorFlow 1.0

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

MOTIVATION

Computational Graphs



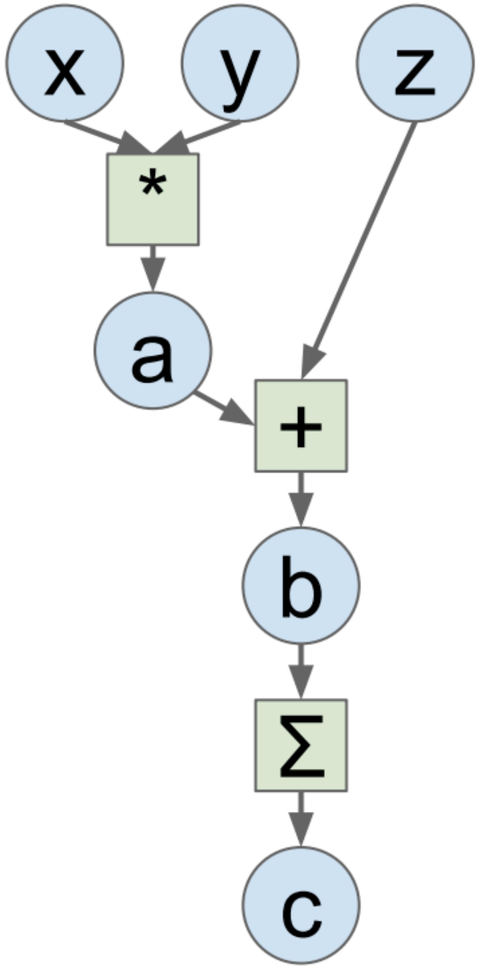
TensorFlow 1.0

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Compute gradients

MOTIVATION

Computational Graphs



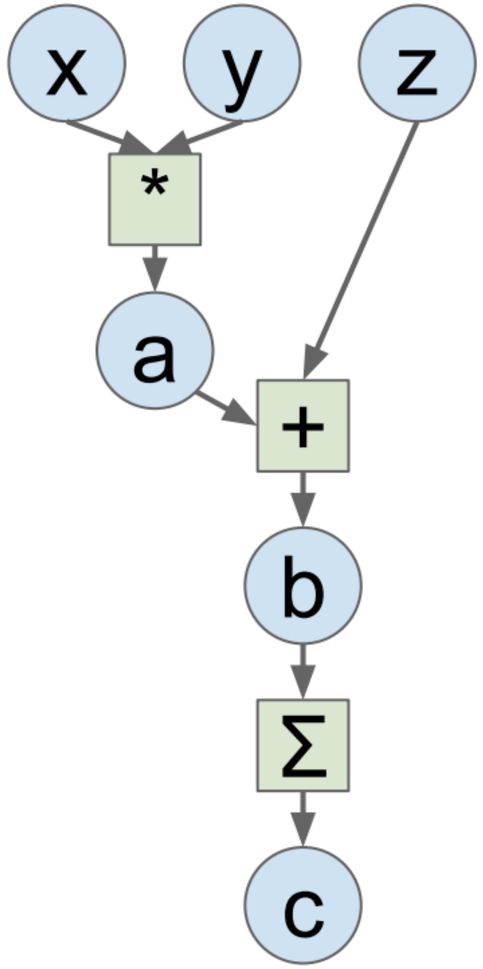
Tell TensorFlow to run on CPU

TensorFlow 1.0

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

N, D = 3000, 4000

with tf.device('/cpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

MOTIVATION

Computational Graphs



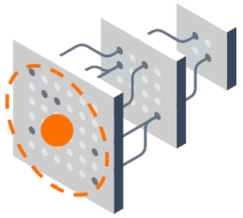
Tell TensorFlow to run on GPU

Now, we're:

- Running on a GPU
- Having framework compute the gradients for us

TensorFlow 1.0

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

N, D = 3000, 4000

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```



Easy model building

Build and train ML models easily using intuitive high-level APIs like Keras with eager execution, which makes for immediate model iteration and easy debugging.

Robust ML production anywhere

Easily train and deploy models in the cloud, on-prem, in the browser, or on-device no matter what language you use.

Powerful experimentation for research

A simple and flexible architecture to take new ideas from concept to code, to state-of-the-art models, and to publication faster.

We'll be using TensorFlow 2.0 in this class

Carnegie Mellon University
School of Computer Science

TensorFlow

- We will use TensorFlow (2.0) in HW5
- For additional resources on TensorFlow, please check out:

  - Quickstart: https://www.tensorflow.org/tutorials/quickstart/beginner
  - Guide: https://www.tensorflow.org/guide
  - Tutorials: https://www.tensorflow.org/tutorials
  - **Recitation video released by Friday!**

Outline

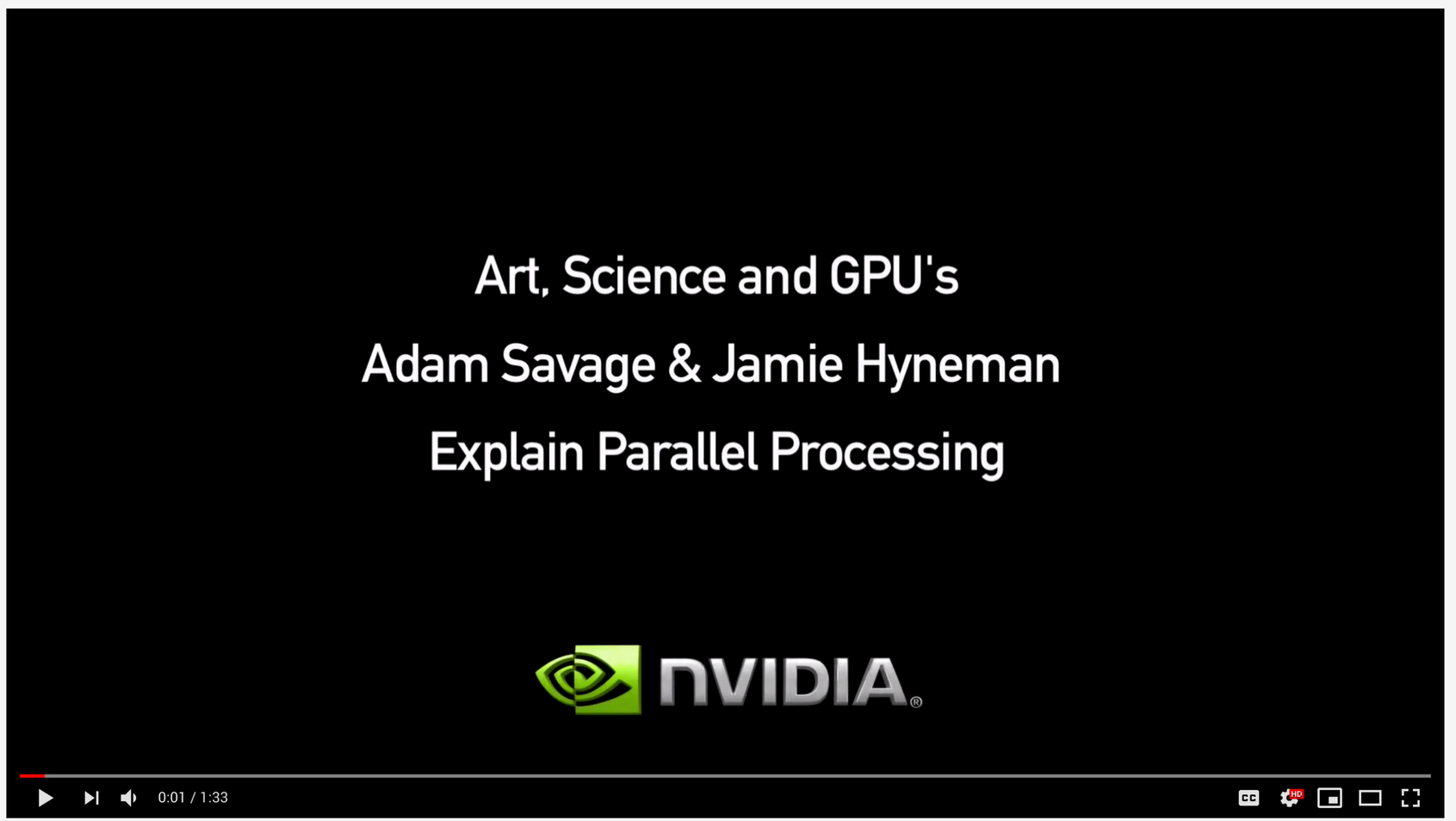
1. Neural networks
2. Backprop
3. ML Frameworks
4. DL Hardware

OK. LET'S CHANGE GEARS A BIT

What about the hardware? TensorFlow is supposed to be good on GPUs.

WHY GPUS?

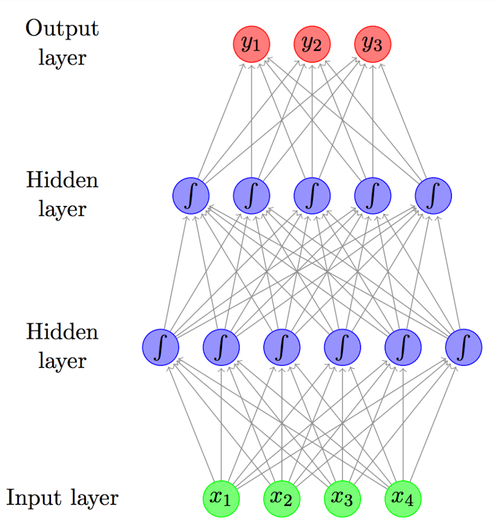
Parallelism



https://www.youtube.com/watch?v=-P28LKWTzrI

RECALL: COST OF THE 'FORWARD PASS'

Feedforward neural networks



Can express NN model with the following recurrence:

$$\mathbf{o}_0 = \mathbf{x}$$

$$\forall l \in \{1,\ldots,L\},\ \mathbf{h}_l = \mathbf{W}_l^\top \mathbf{o}_{l-1}$$

$$\forall l \in \{1,\ldots,L\},\ \mathbf{o}_l = \sigma_l(\mathbf{h}_l)$$

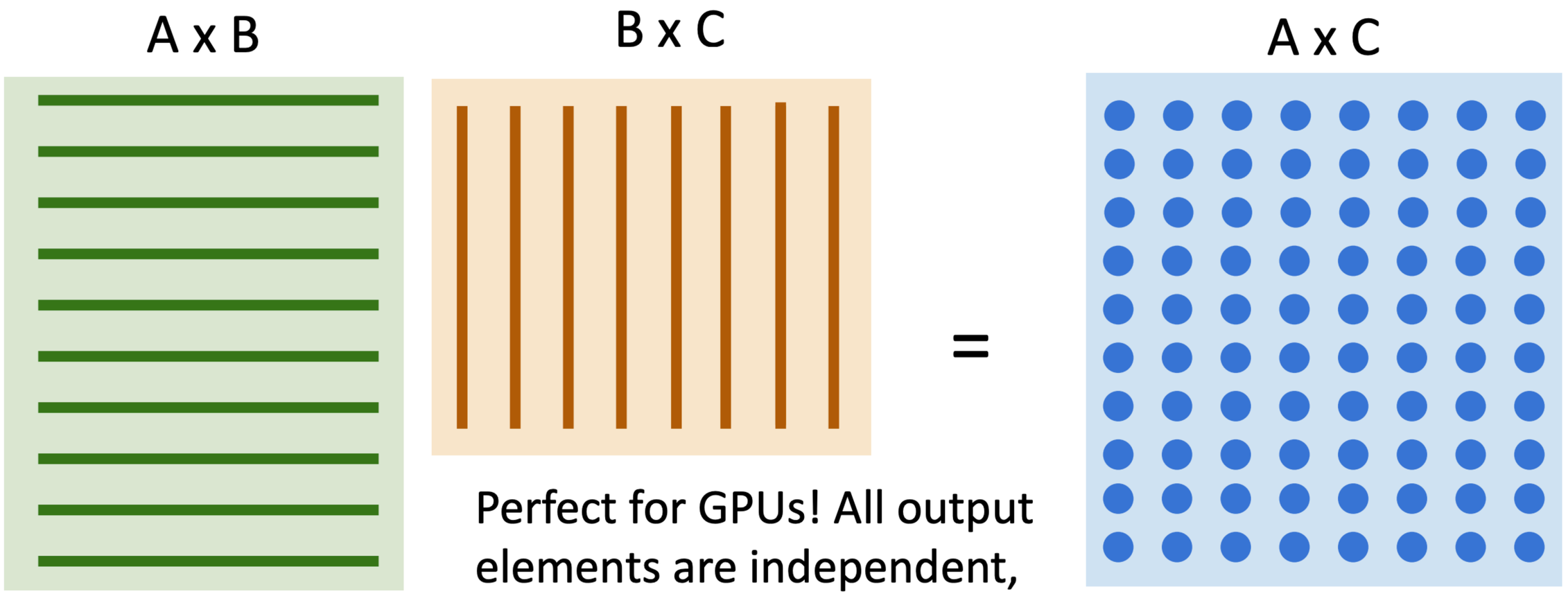
$$\phi(\mathbf{x}) = \mathbf{o}_L$$

What dominates the cost?

Matrix(-vector) multiplies

O(nm) for n x m matrix, m x 1 vector

Matrix Multiplication



Perfect for GPUs! All output elements are independent, can be trivially parallelized

What do we already know about GPUs?

What do we already know about GPUs?

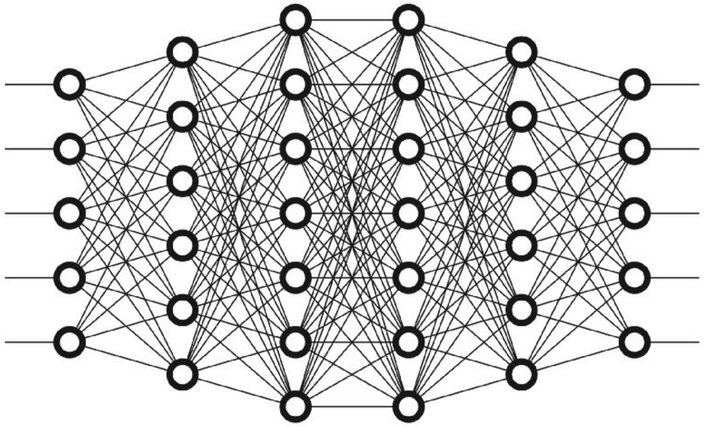
If you’re a gamer, there is a bit of a debate between NVIDIA and AMD.

What do we already know about GPUs?

If you're a gamer, there is a bit of a debate between NVIDIA and AMD.

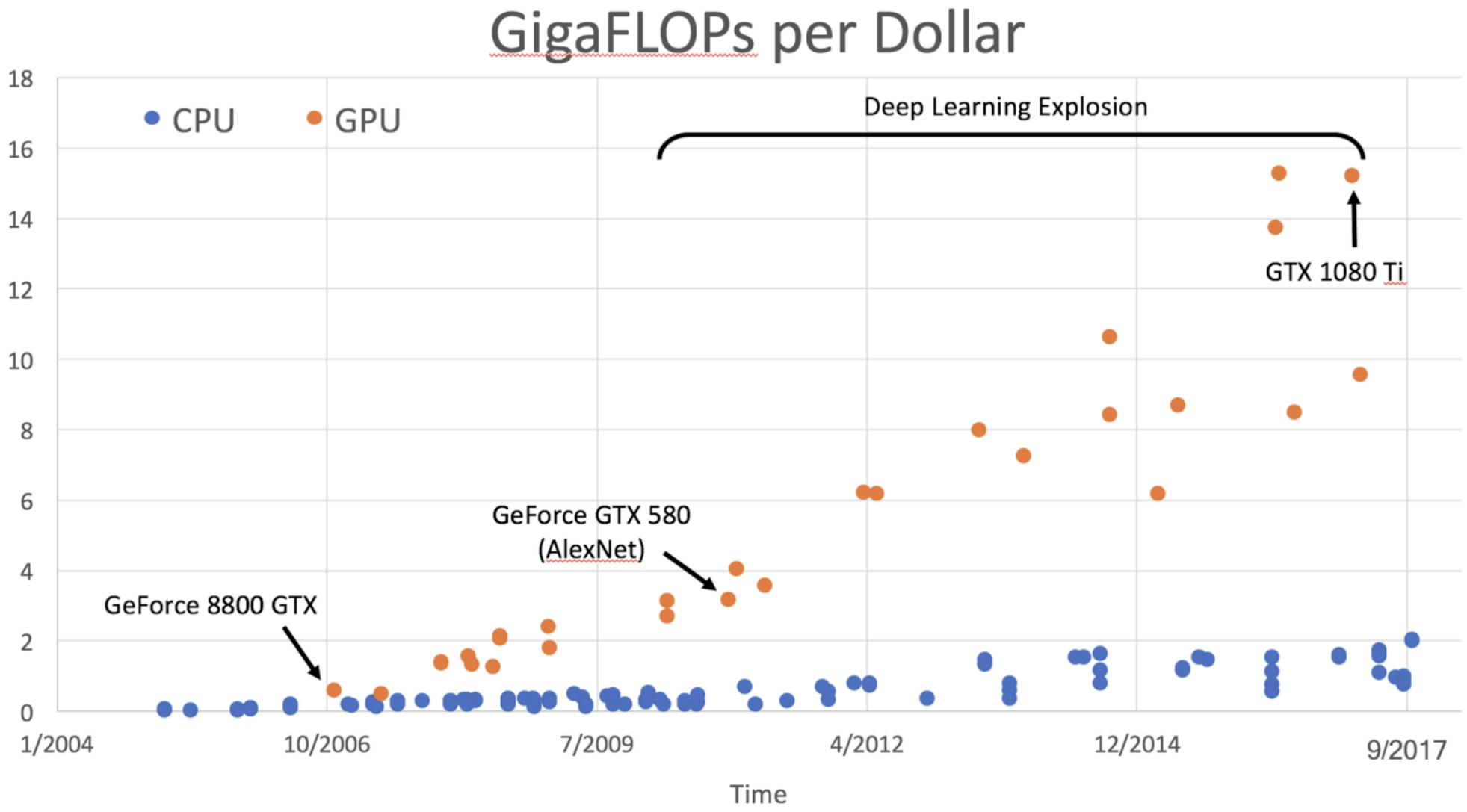
However, in the context of deep learning,

NVIDIA wins.



ML HARDWARE

GigaFLOPs per Dollar



https://web.eecs.umich.edu/~justincj/slides/eecs498/498_FA2019_lecture09.pdf

CPUs vs GPUs

	Cores	Clock Speed (GHz)	Memory	Price	TFLOP/sec
CPU Ryzen 9 3950X	16 (32 threads with hyperthreading)	3.5 (4.7 boost)	System RAM	$749	~4.8 FP32
GPU NVIDIA Titan RTX	4608	1.35 (1.77 boost)	24 GB GDDR6	$2499	~16.3 FP32

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

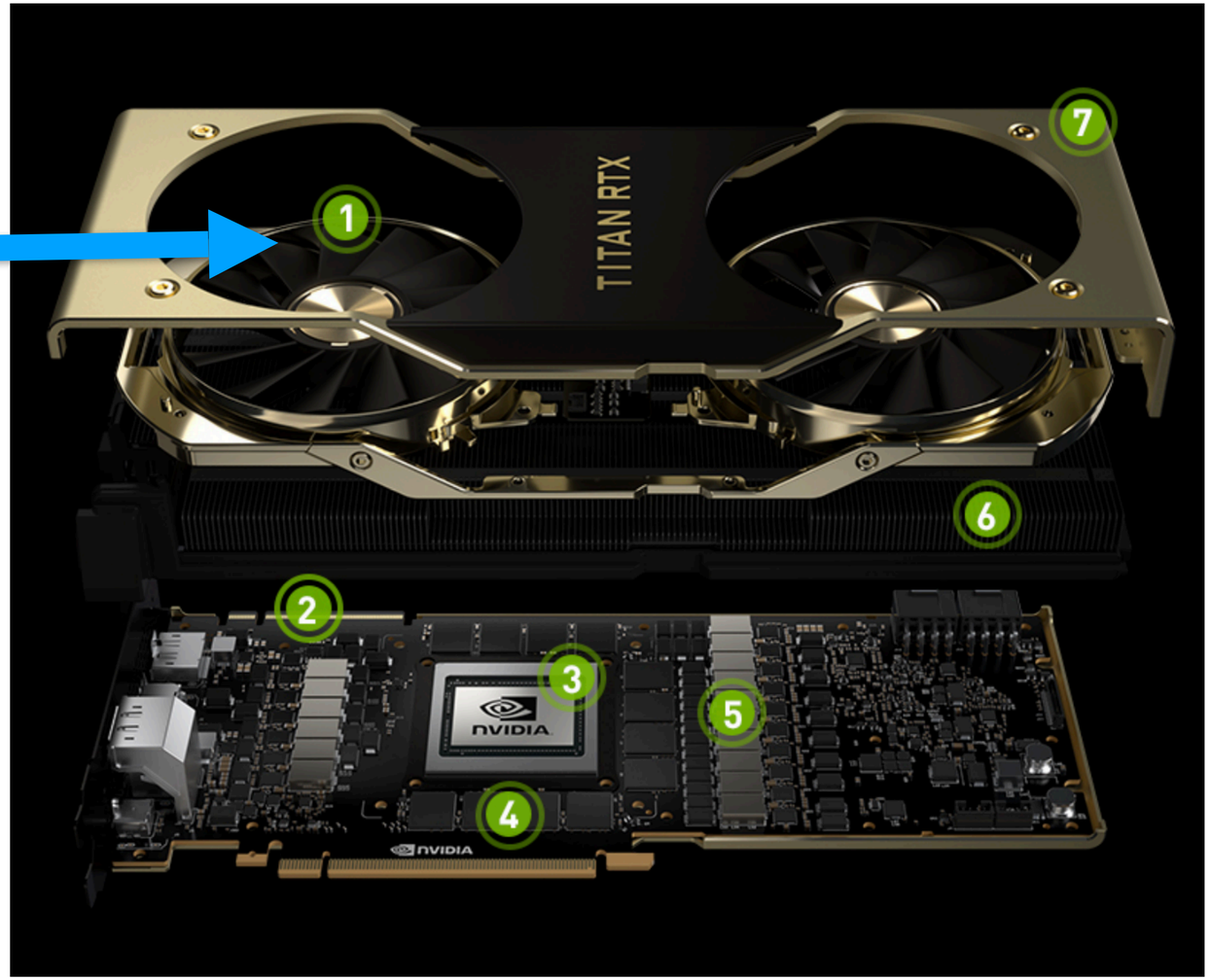
GPU: More cores, but each core is much slower and "dumber"; great for parallel tasks

https://web.eecs.umich.edu/~justincj/slides/eecs498/498_FA2019_lecture09.pdf

Inside A GPU: RTX Titan

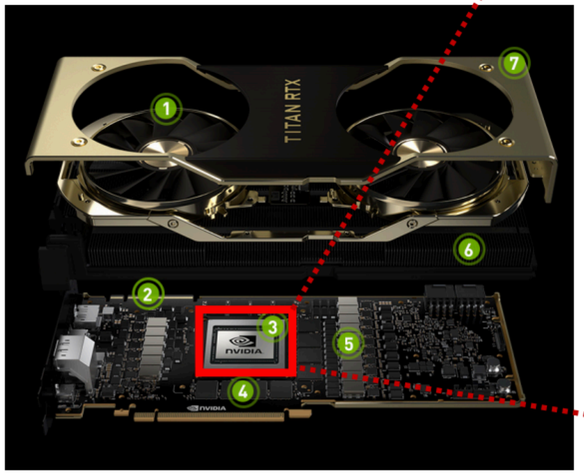
HOT! — Fans!

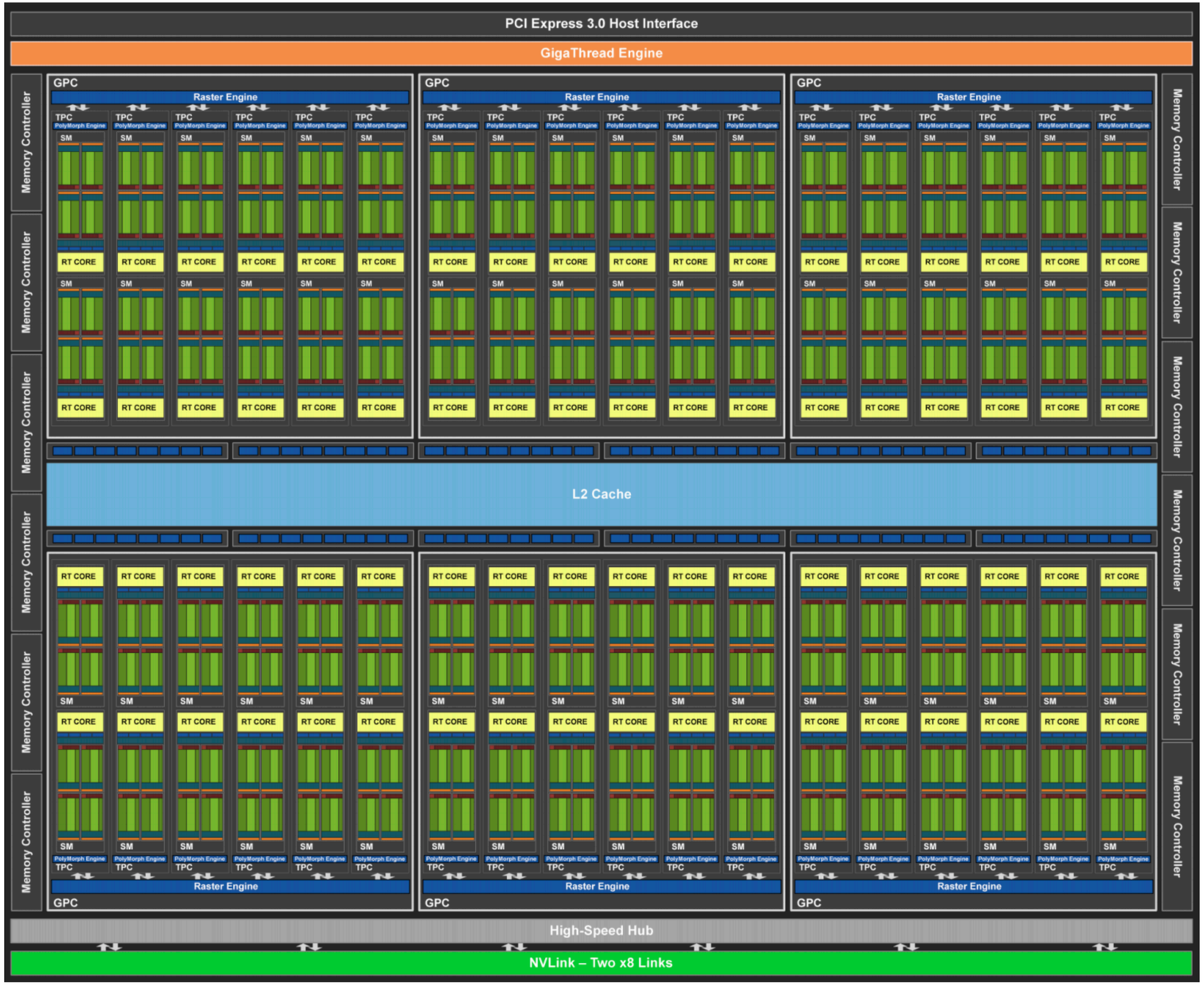
Lot’s of energy!



ML HARDWARE

Inside A GPU:

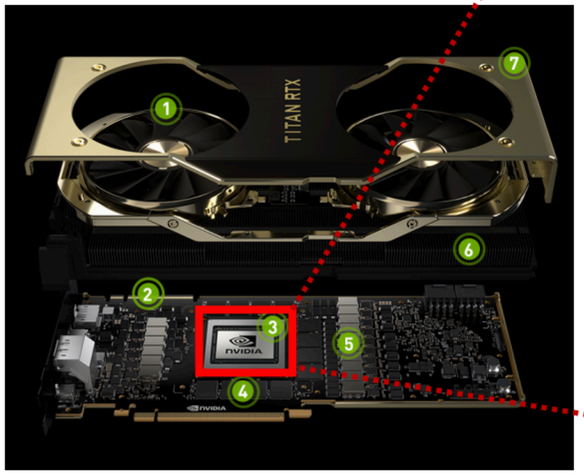


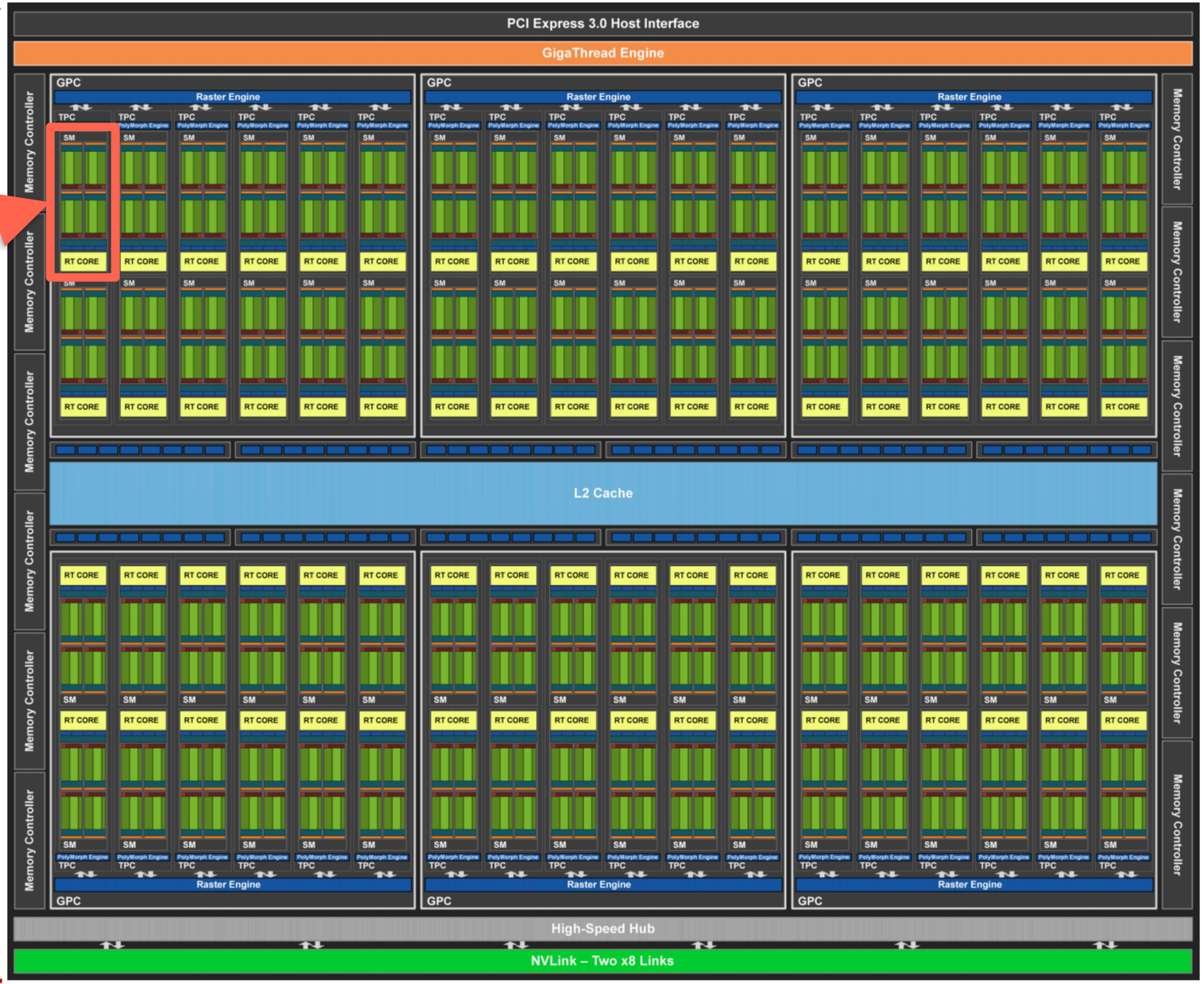


ML HARDWARE

Inside A GPU:

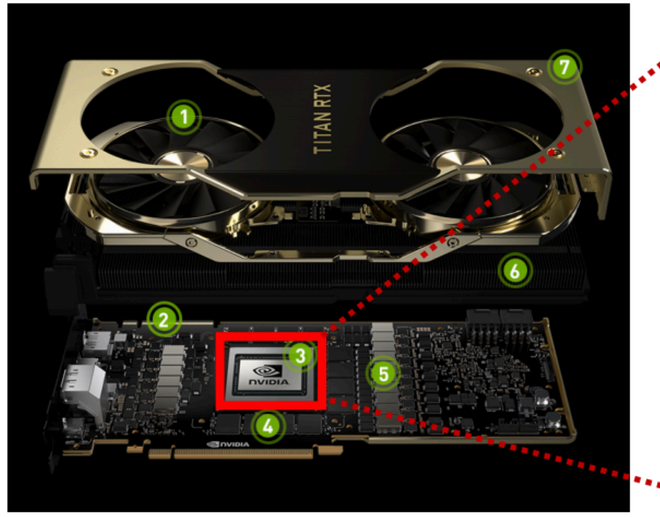
72 streaming multiprocessors (SMs)



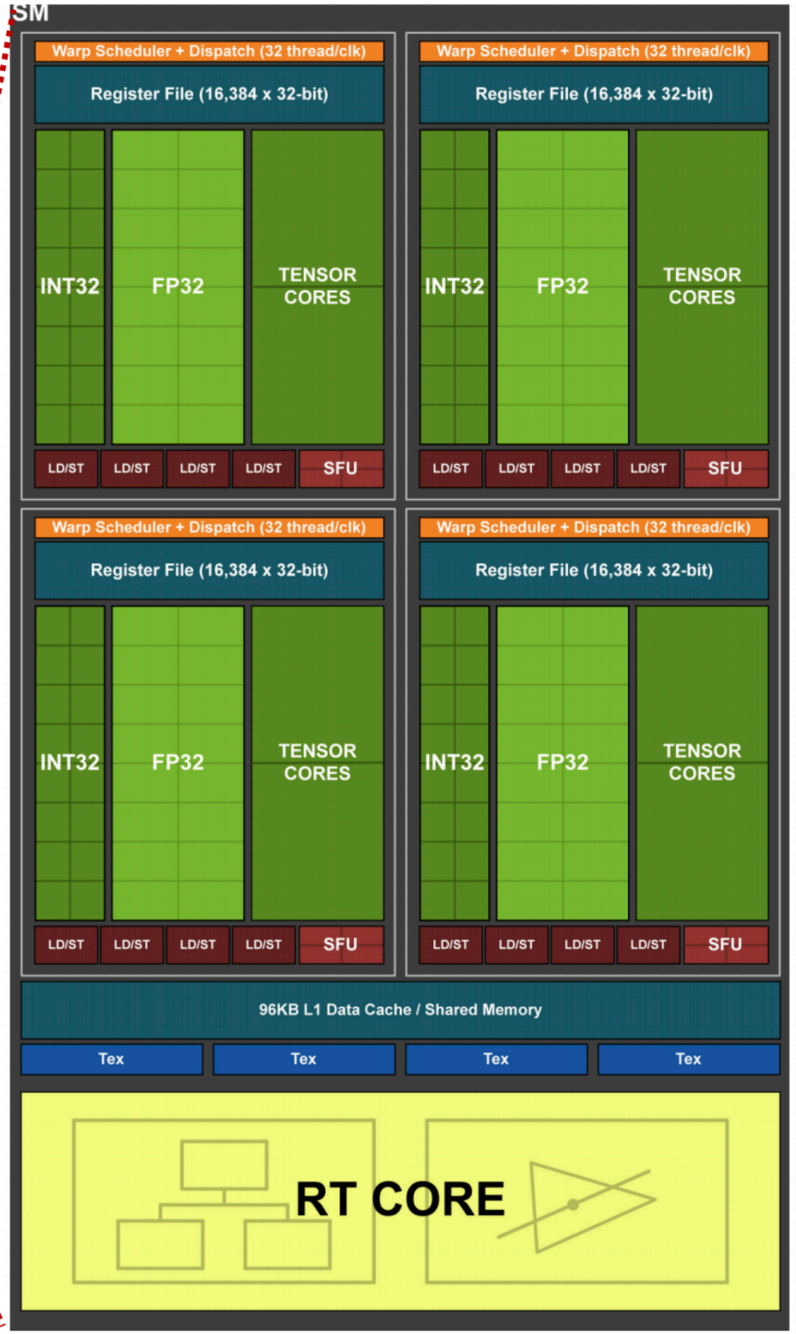


ML HARDWARE

Inside A GPU: RTX Titan



72 Streaming multiprocessors (SMs)

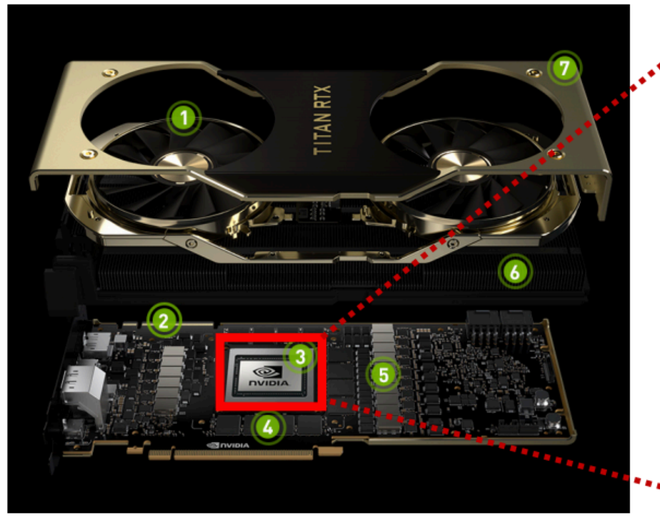


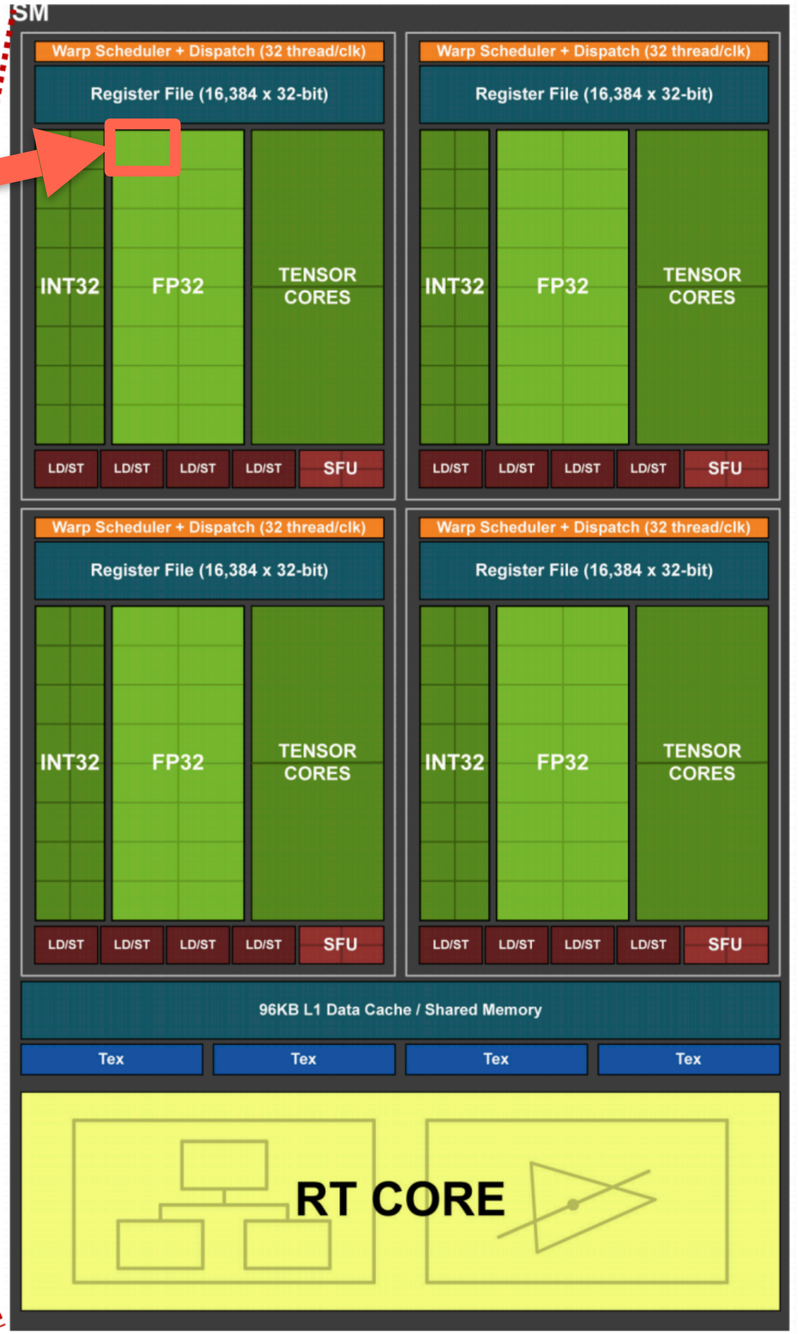
ML HARDWARE

Inside A GPU: RTX Titan

64 FP32 cores per SM

72 Streaming multiprocessors (SMs)





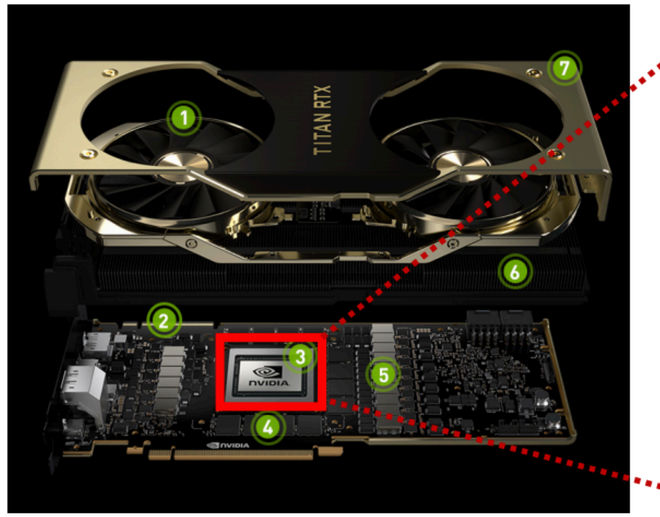
Inside A GPU: RTX Titan

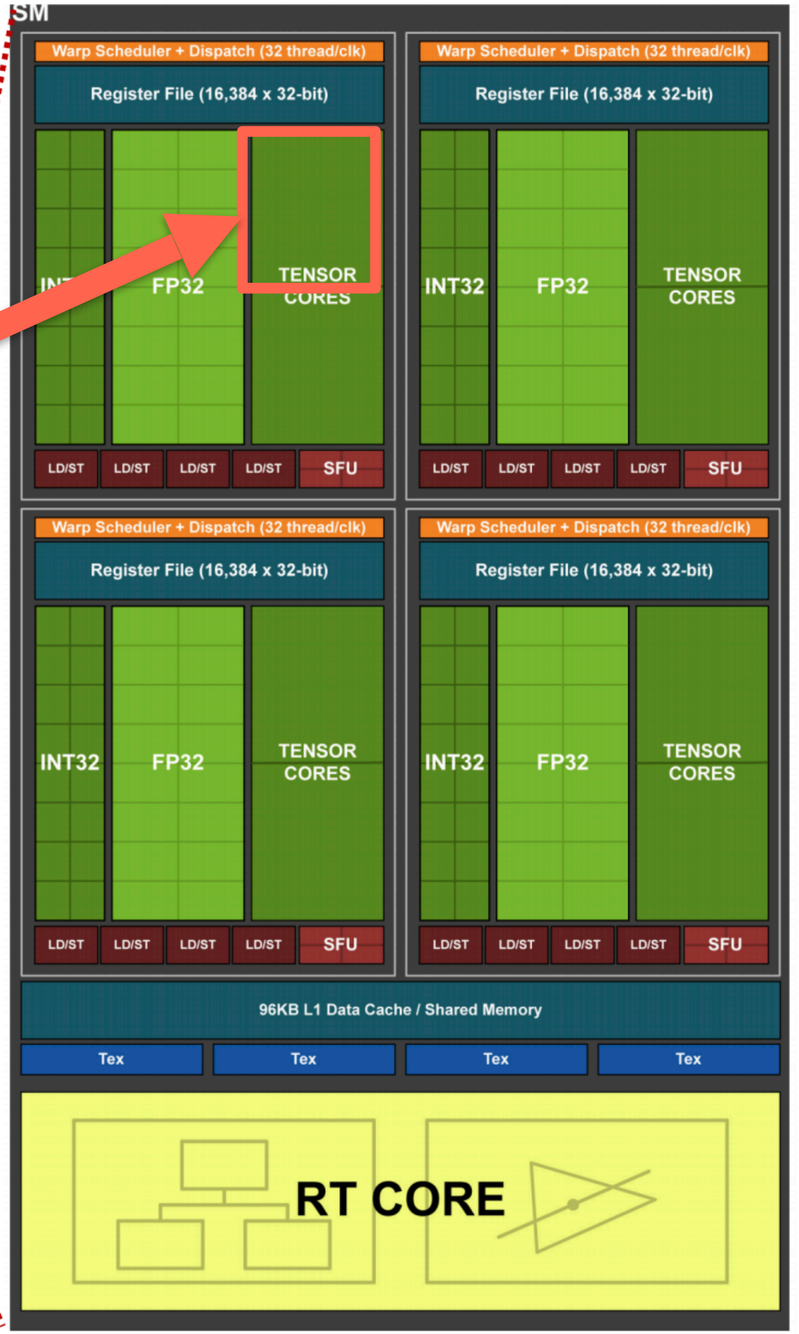
Tensor Core:
Special hardware for DL!

Let A,B,C be 4x4 matrices; computes AB+C in one clock cycle! (128 FLOP)

8 Tensor Core per SM

72 Streaming multiprocessors (SMs)





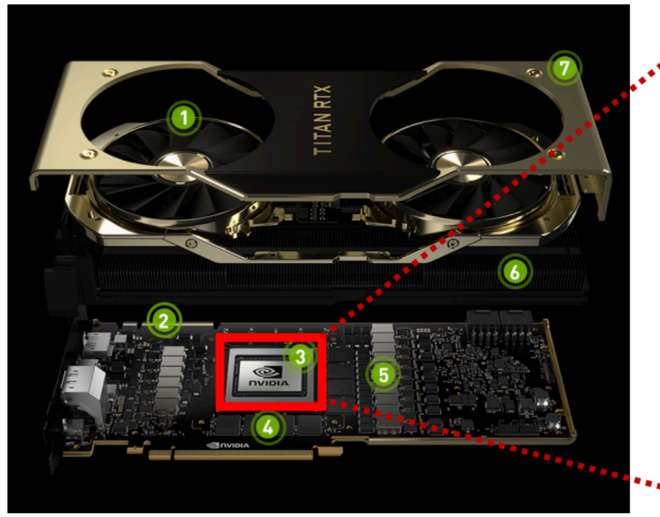
ML HARDWARE

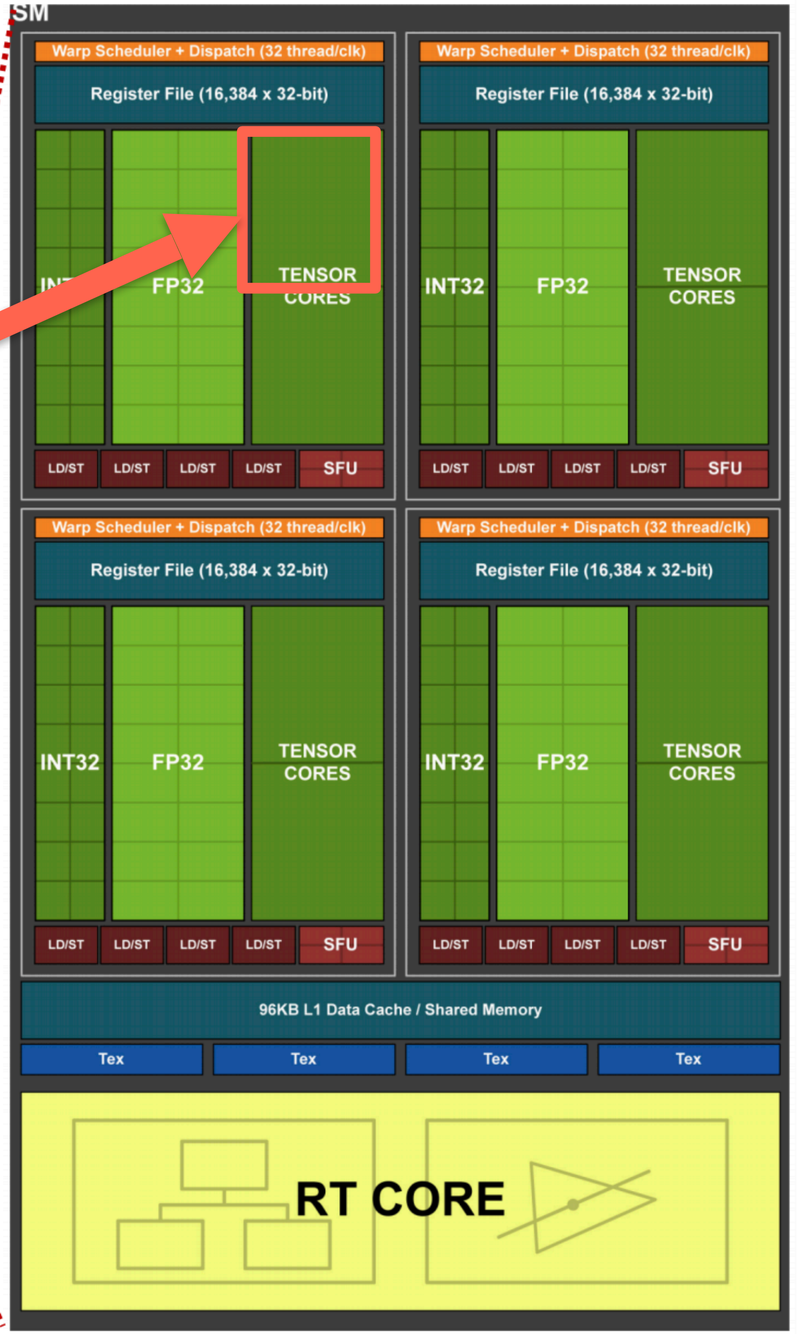
Inside A GPU: RTX Titan

Tensor cores use mixed precision: Multiplication is done in FP16, and addition is done in FP32

8 Tensor Core per SM

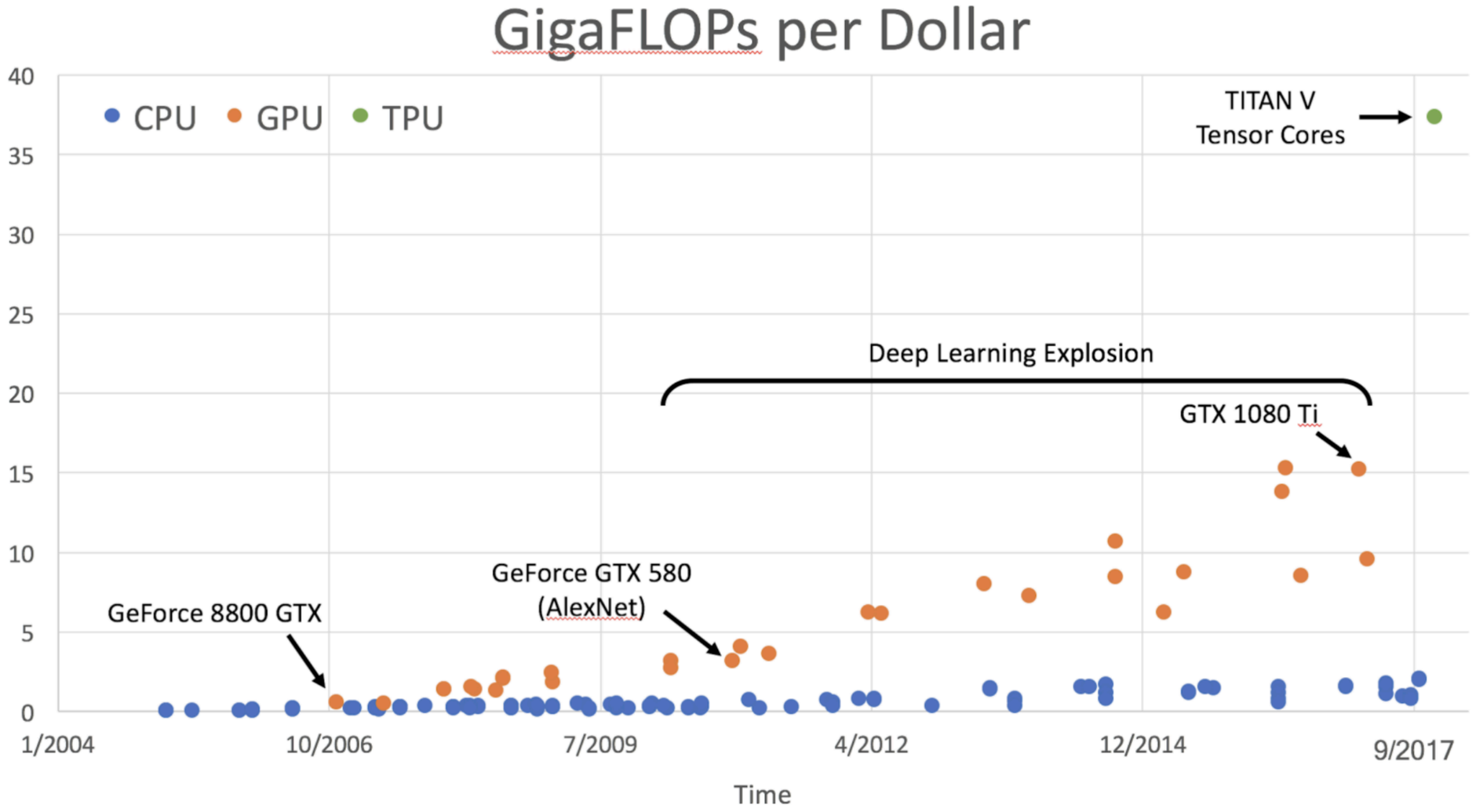
72 Streaming multiprocessors (SMs)





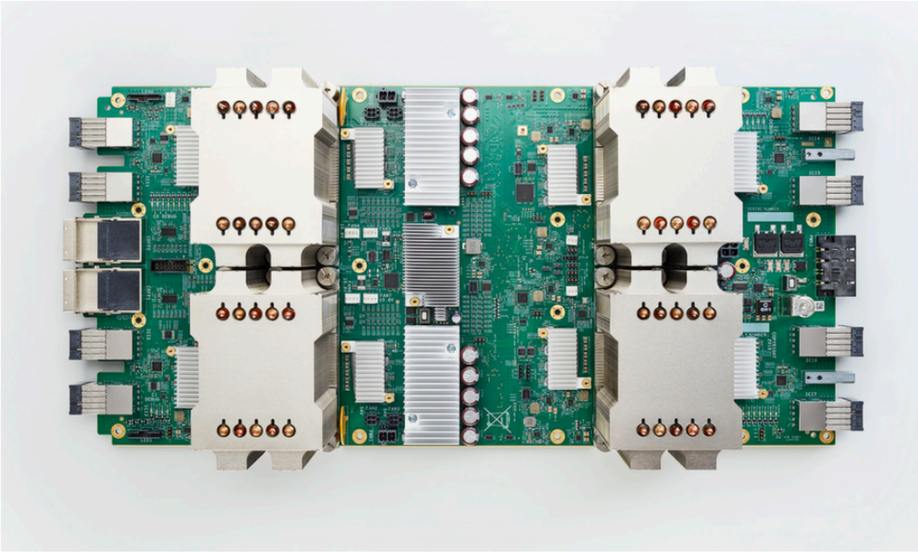
Scaling up: 8 GPUs per server

GigaFLOPs per Dollar



ML HARDWARE

Google's Tensor Processing Units (TPUs)



Special hardware for matrix multiplication, similar to NVIDIA Tensor Cores; also runs in mixed precision (bfloat16)

Cloud TPU v2

180 TFLOPs

64 GB HBM memory

$4.50 / hour

(free on Colab!)

ML HARDWARE

Google's Tensor Processing Units (TPUs)

Model with TPU Modifications

```
def model_fn(features, labels, mode, params):
  input_layer = tf.reshape(features, [-1, 28, 28, 1])
  conv1 = tf.layers.conv2d(inputs=input_layer, ...)
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2,2],
    strides=2)
  # ...
  loss = tf.losses.softmax_cross_entropy(
    onehot_labels=onehot_labels, logits=logits)
  optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
  optimizer = tpu_optimizer.CrossShardOptimizer(optimizer)
  train_op = optimizer.minimize(loss)
  return tpu_estimator.EstimatorSpec(mode=mode, loss=loss,
    train_op=train_op)
```

No further change required for TPU Pod

So, the whole TPU Pod works just like a single computer without the tedious manual clustering setup.

Google Cloud

19 / 26

PRESENTATION ANIMATION DEMO SUBTITLES EN JP

niversity
uter Science

Google's Tensor Processing Units (TPUs)

https://storage.googleapis.com/nexttpu/index.html (slides 3-6, 1min)

Summary

CPUs

- General purpose
- Cheap
- Can work well with some DL models (e.g., RNNs), but generally slower than GPUs/ TPUs

GPUs

- Initially designed for graphics, now re-purposed/specialized for DL
- Highly parallel → fast for DL
- Expensive, though typically cost-efficient for DL jobs relative to CPUs

TPUs

- First chip specifically designed for DL
- Fast and cost-efficient for many DL jobs
- Less flexible / worse for ‘irregular’ computations (small batches, non-matrix mul)

Additional Resources

- Podcast with Dave Patterson: https://determined.ai/blog/dave-patterson-podcast-new/
- TPUs: https://storage.googleapis.com/nexttpu/index.html
- Energy Considerations: https://ai.googleblog.com/2022/02/good-news-about-carbon-footprint-of.html