MSE | MASTER OF SCIENCE IN ENGINEERING

Deep Learning

TSM-DeLearn

7. Deep Learning Hardware & Software

Recap 1 - Gradient descent

- Update any parameters of your model in the opposite direction of the gradient of the loss w.r.t. weights

$$\text{param} \leftarrow \text{param} - \alpha \frac{\partial J}{\partial \text{param}}$$

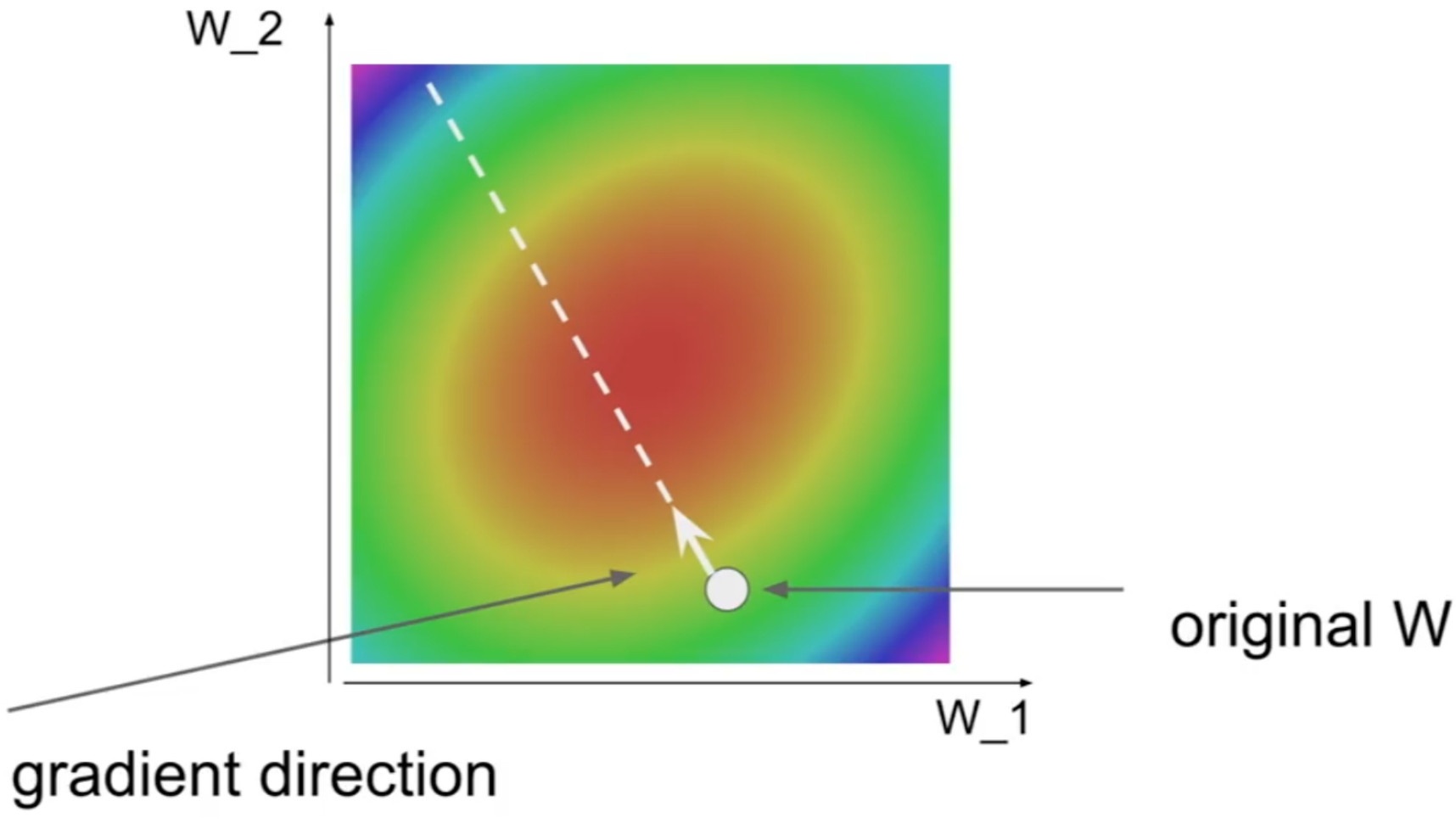
In practice we have stopping criteria, e.g. epoch < max_epochs or loss_gain < ε

1 sample = stochastic gradient descent
B samples = mini-batch gradient descent
N samples = (full) batch gradient descent

Current weighs

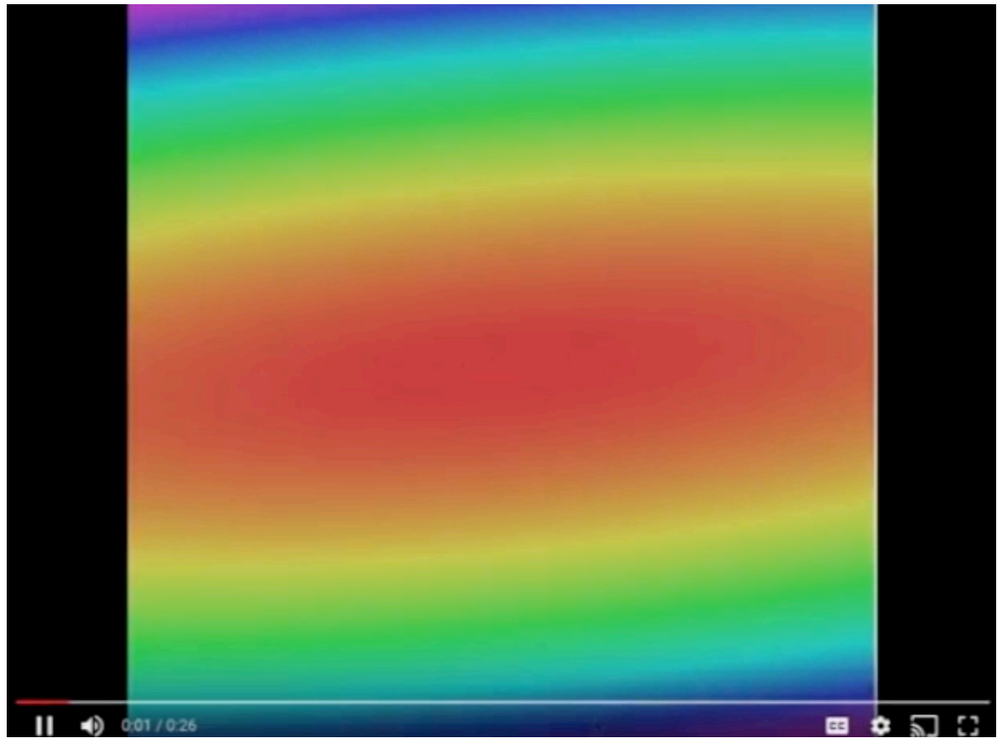
```
# Vanilla Gradient Descent

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

The loss function usually includes a **performance** term and a **regularisation** term



negative gradient direction

From Fei-Fei Li & Justin Johnson & Serena Yeung,
April 2018: CS231n Stanford



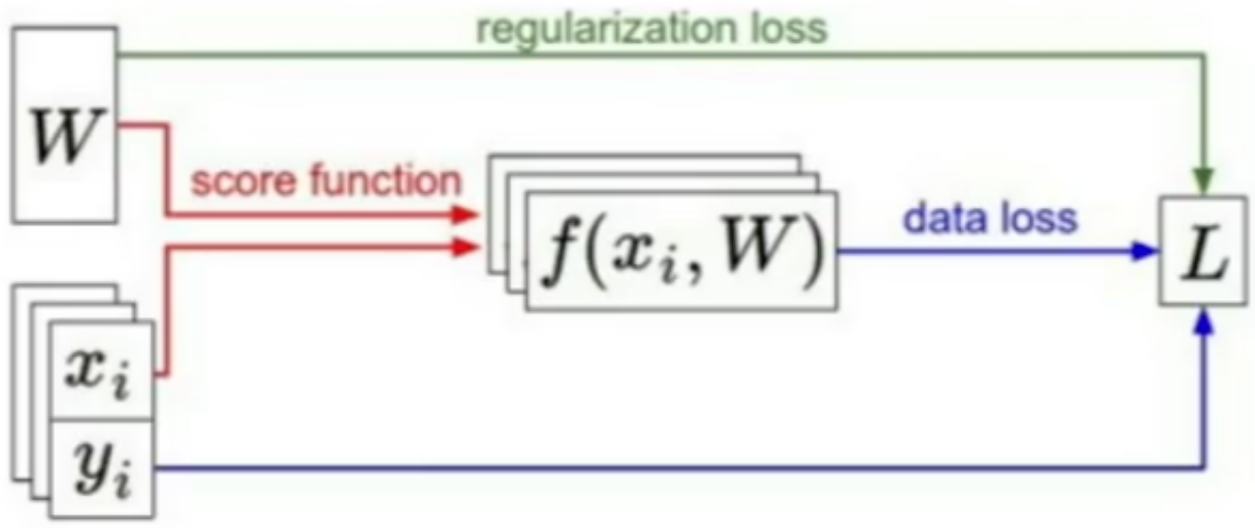
Recap 2 - Regularisation methods

- Add a term to the loss function to impeach the model to become too “complex”

$$J = J_{data} + J_{reg}$$

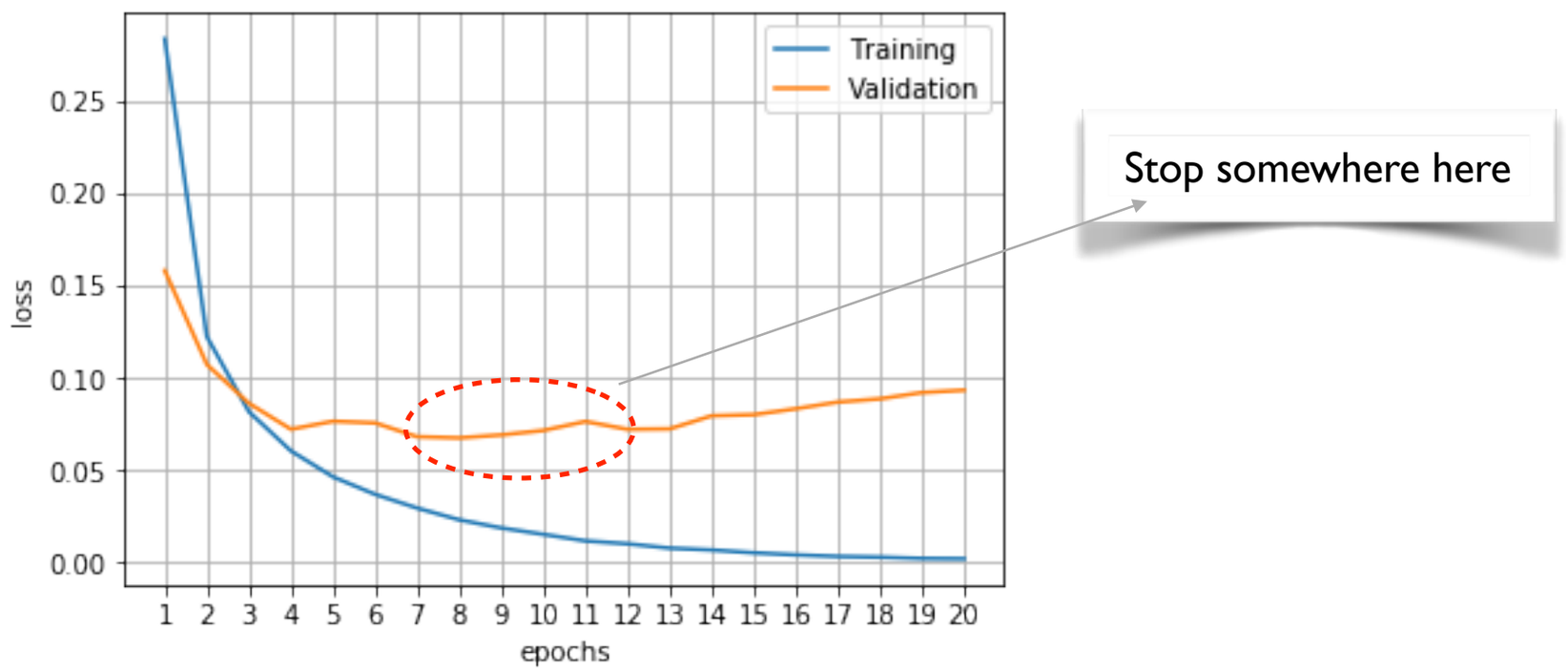
Performance term: on the data, how far are we predicting from the ground truth?
E.g. MSE, hinge or cross-entropy loss
see https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

Regularisation term: we should not let our model become too complex.
E.g. weight decay with L1 or L2



Recap 3 - Other regularisation methods

- **Dropout** - randomly drop neurons during training to "share the knowledge" and make the solution less dependent on individual neurons
- **Early stopping** - stop training when loss is increasing on a validation set



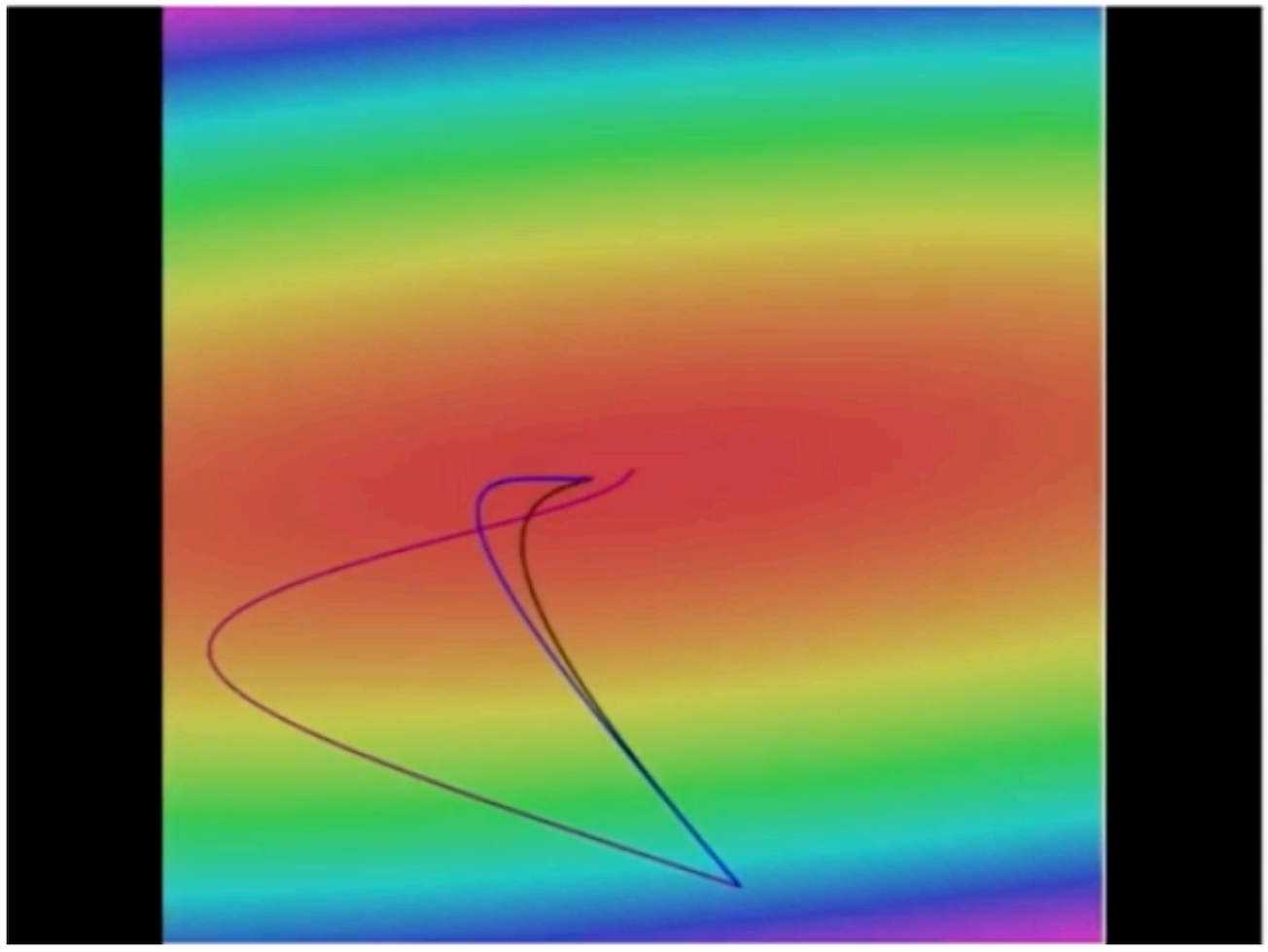
- Data augmentation

Recap 4 - optimisation strategies

- Why : problems of vanishing gradient, dead neurons, convergences
- **Normalisation** of the input samples, e.g. re-scaling, z-norm
- Good parameter initialisation, e.g. random init, **Xavier**
- Normalisation of the activations of neurons: **batch normalisation**
- Use adequate activations for your problem, e.g. using non-saturating activations

Recap 4 - optimisation strategies

- Use advanced learning rate evolution strategies, e.g. learning schedule, momentum with Nesterov, RMS prop, **Adam**
- See also : https://distill.pub/2017/momentum/



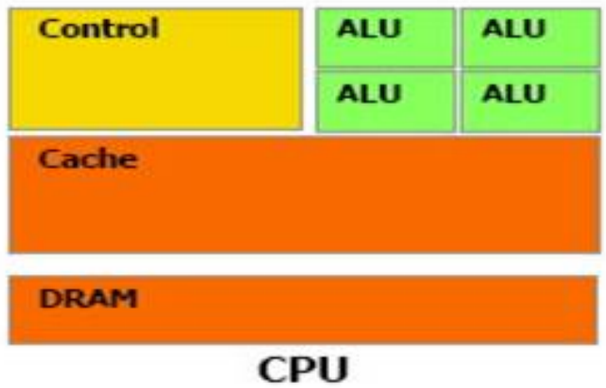
Plan

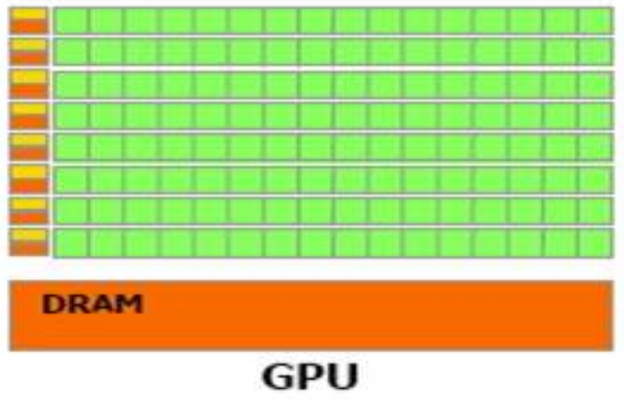
1. CPU vs GPU
2. Computational graphs implementation
3. Deep Learning Frameworks
    1. Overview of the “zoo”
    2. Tensorflow overview
4. PW on Tensorflow

CPU vs GPU

Inside a server - spot the cpu and gpu

CPU vs GPU



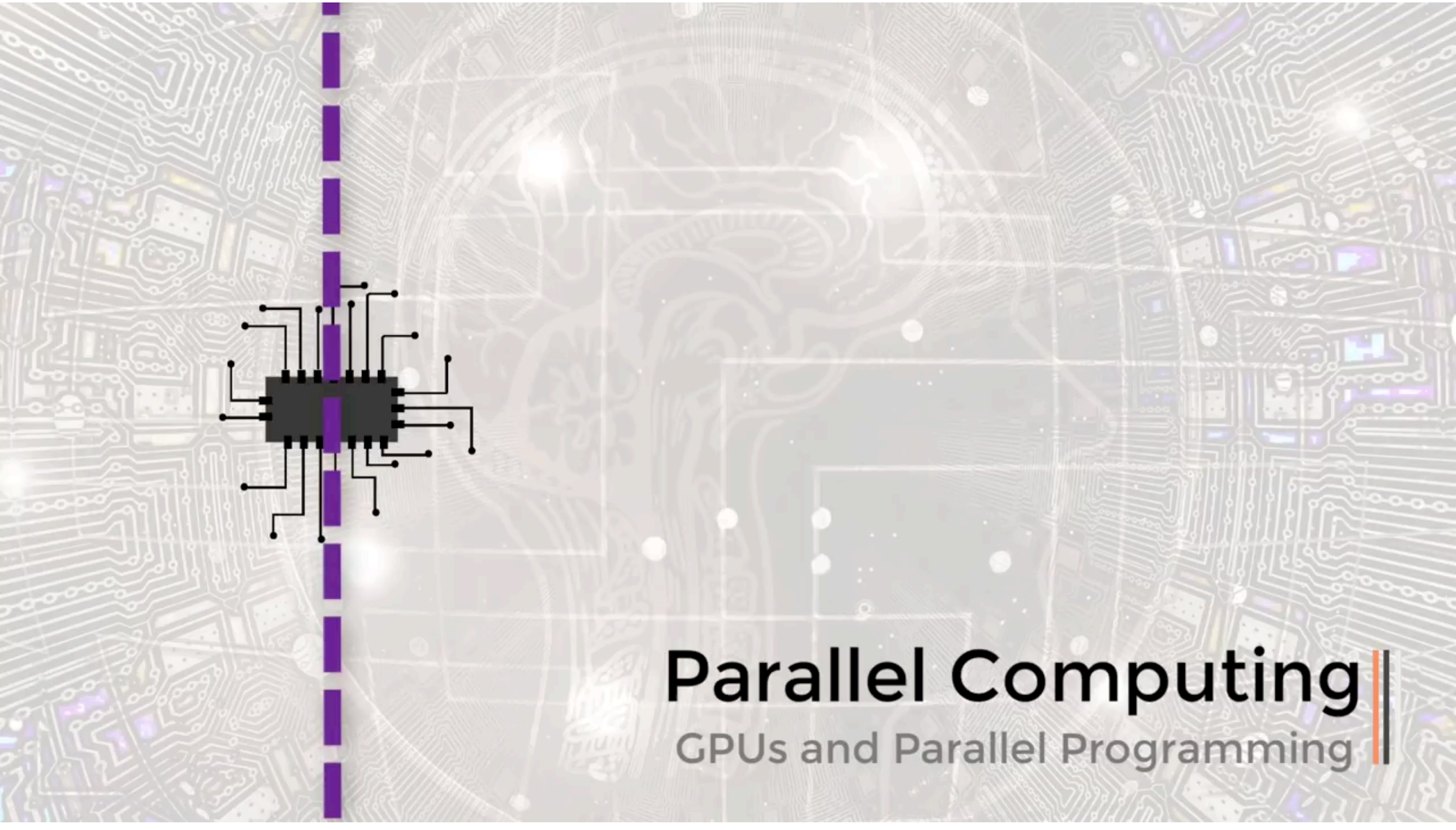


CPU - Central Processing Unit

- Few cores (~10), fast (~4 GHz), lots of cache, few parallel processes
- Great at sequential tasks
- Providers: Intel, AMD

GPU - Graphical Processing Unit

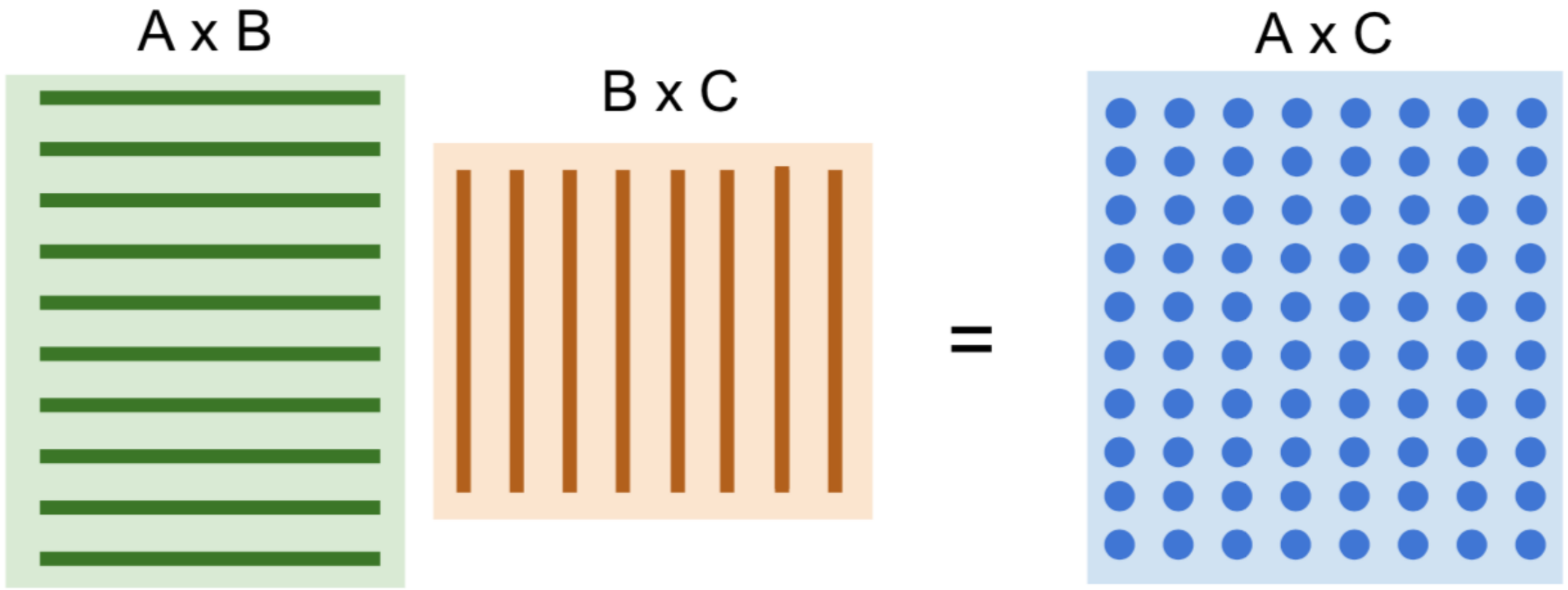
- Many cores (~1’000), slow (~1.5 GHz)), few cache, many parallel processes
- Great at parallel tasks
- Providers: **NVIDIA**, AMD



Why Deep Learning uses GPUs

https://www.youtube.com/watch?v=6stDhEA0wFQ

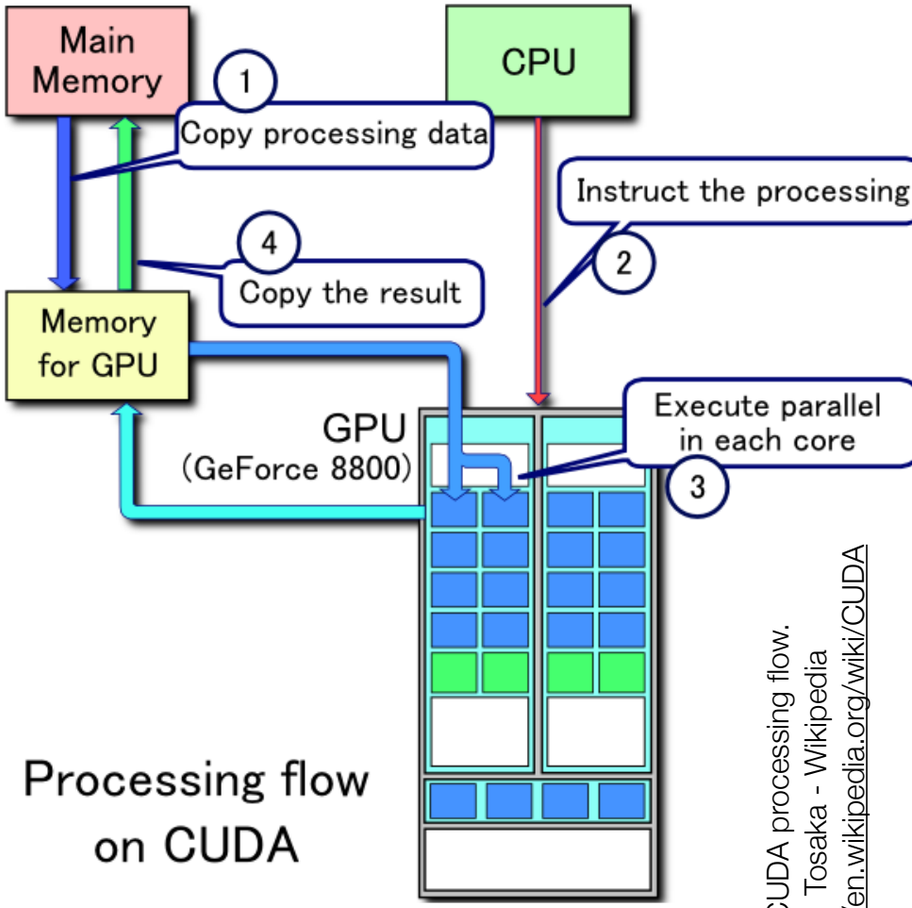
Typical problem for GPU: matrix multiplication



- This operation is very common for ANN:
  - Ex: A = num of samples in your batch, B = dim of input vector, C = num of neurons, AxC = matrix of logits
- Each blue dot of the result matrix can actually be computed independently, so easily “parallelizable”
- ANNs are “embarassingly parallel”

Programming GPUs

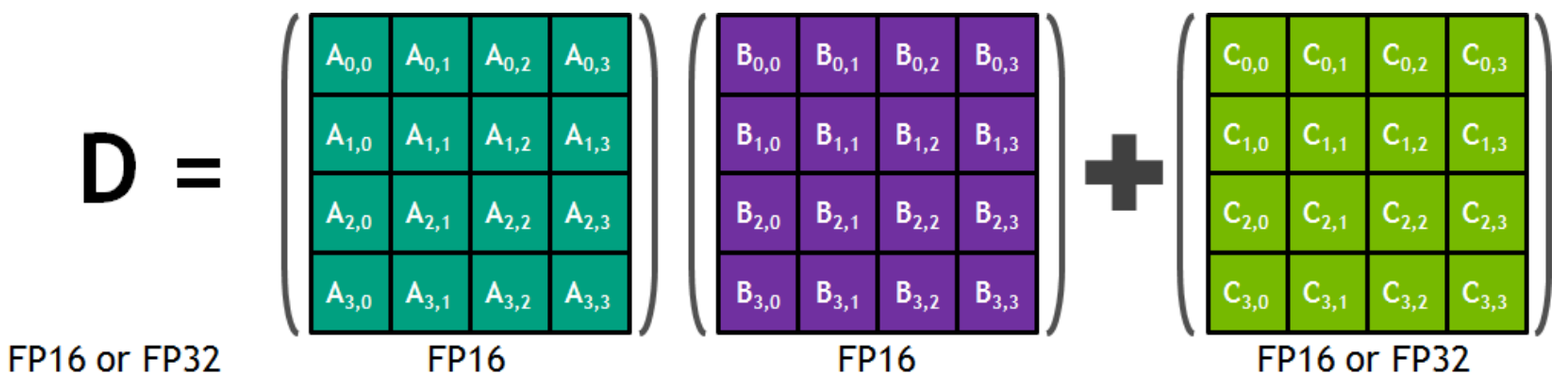
- CUDA - for NVIDIA only
  - Low-level API for programming GPUs in C-like code - see Figure
  - Optimized higher-level APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
  - Similar to CUDA, but runs on anything
  - Usually slower on NVIDIA hardware
- HIP (*new*)
  - https://github.com/ROCm-Developer-Tools/HIP
  - Write once in HIP C++ and port it on NVIDIA and AMD platforms
  - “hipify” tool converts CUDA code to something that can run on AMD GPUs
- Generally in deep learning: you should use existing GPU libraries that are called from other languages



CUDA processing flow. Tosaka - Wikipedia https://en.wikipedia.org/wiki/CUDA

Tensor cores vs. general purpose cpu cores

- New NVIDIA Volta GPU cards, e.g. the Tesla V100, came with the notion of tensor cores
  - Even more specialised hardware to perform matrix and convolution operations used in deep learning



- Similar story for Google's proprietary Tensor Processing Unit TPU

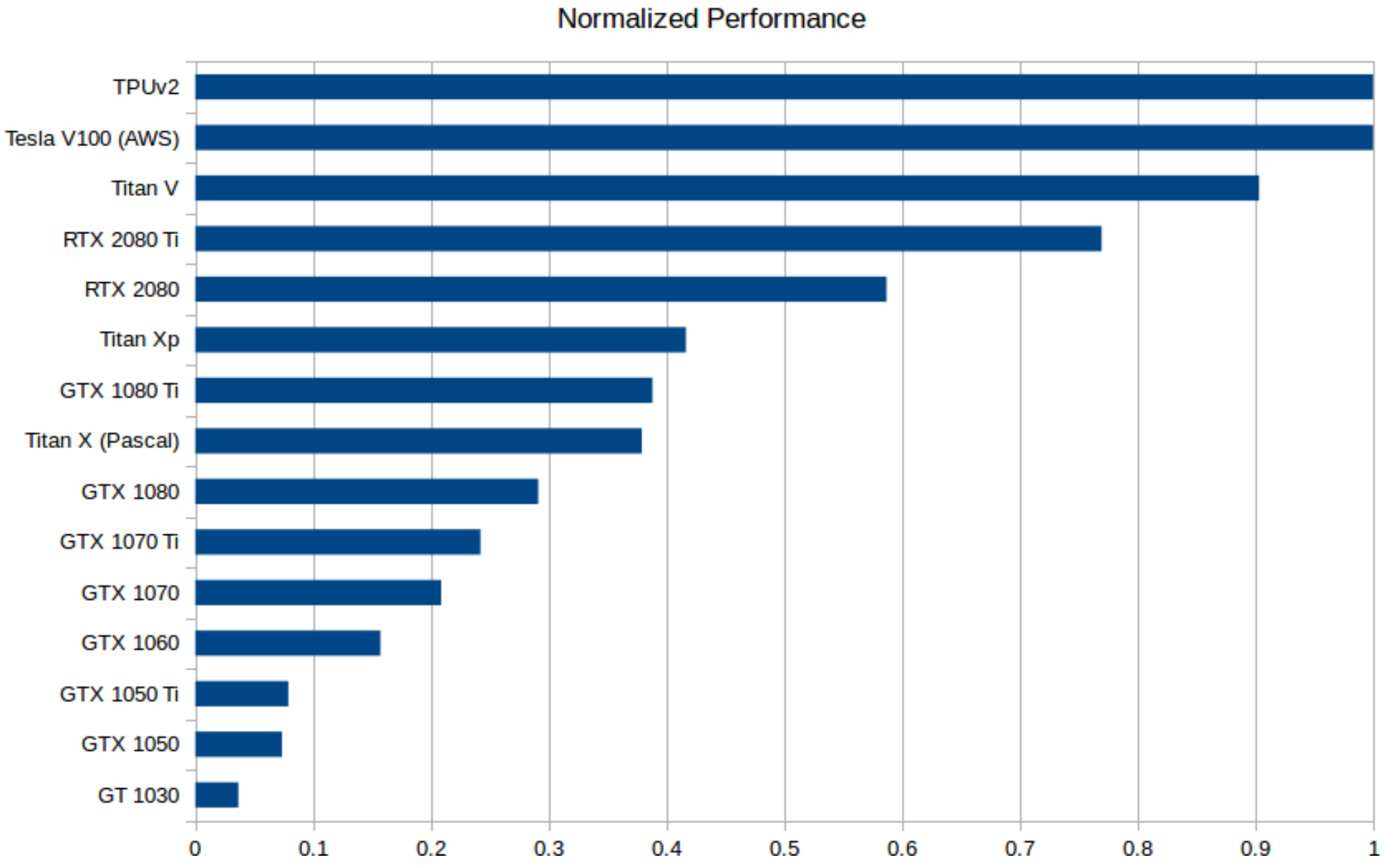
Performance comparison

	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

Comparison chart CPU, GPU and TPU.

From Fei-Fei Li & Justin Johnson & Serena Yeung, April 2018: CS231n Stanford

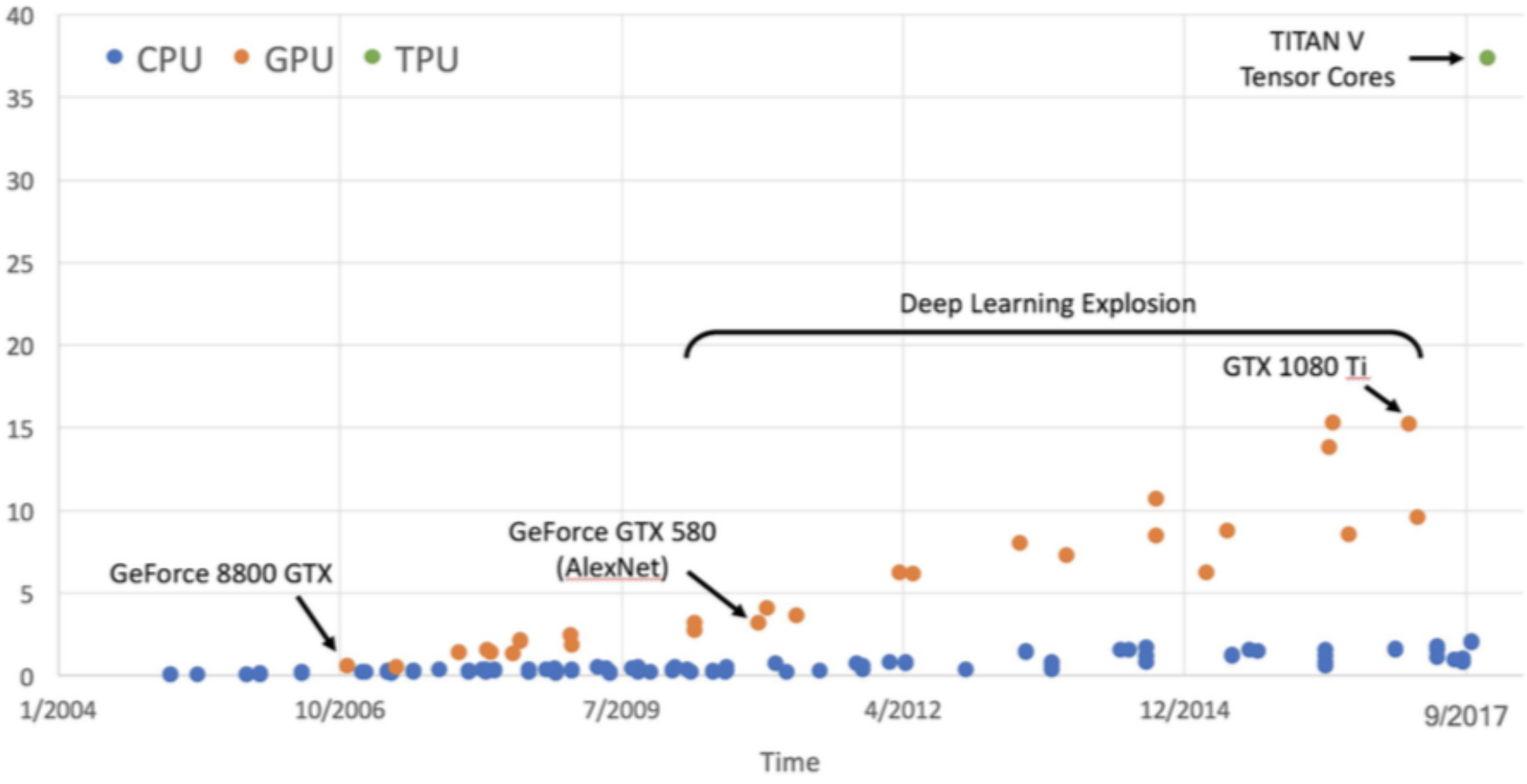
Performance comparison



Normalized Raw Performance Data of GPUs and TPU. Higher is better.

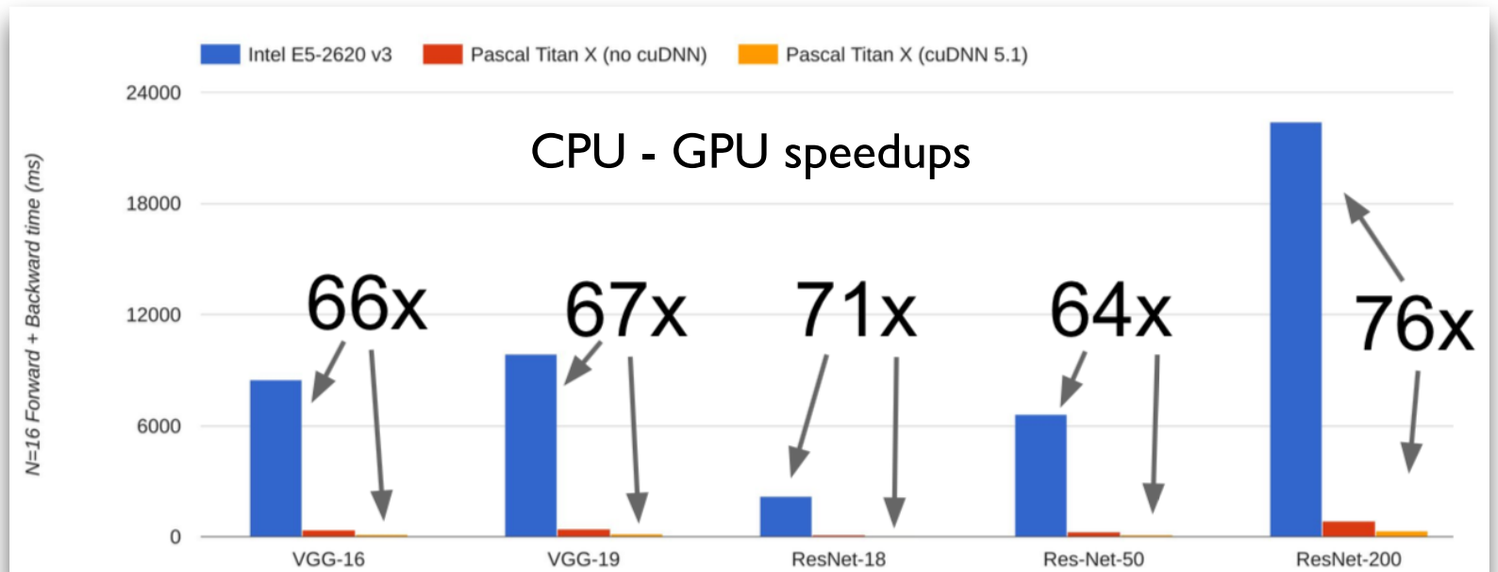
From Tim Dettmers, August 2018: **Which GPU(s) to Get for Deep Learning**
http://timdettmers.com/2018/08/21/which-gpu-for-deep-learning/

Performance comparison

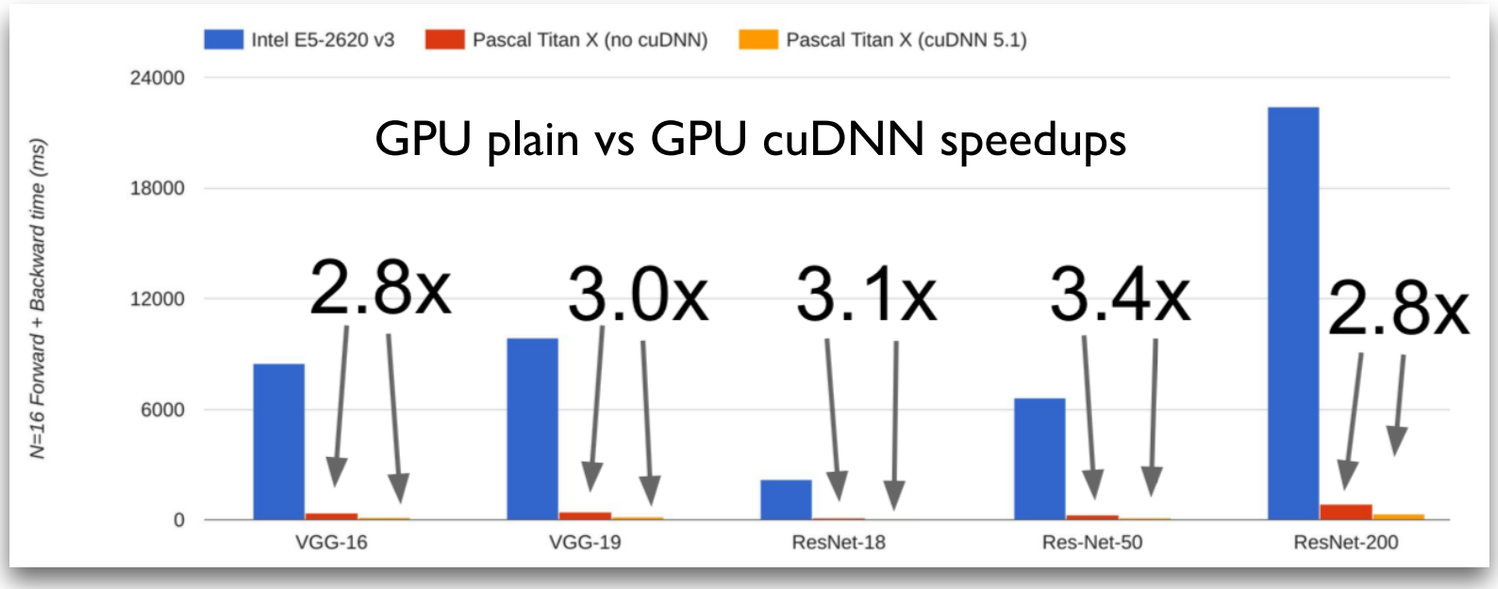


GigaFLOPS per dollar - based on raw performance data and market price
From Fei-Fei Li & Justin Johnson & Serena Yeung, April 2018: CS231n Stanford

Is it worth to move from CPU to GPU?



Quoting Justin Johnson: In this benchmark, CPU performances are not optimised, a little "unfair". See next slide for more optimised cpu implementations and fairer comparisons.



The GPU libraries offered by NVIDIA like cuDNN are really well optimised. No need to fight to write your own CUDA code.

CNN Benchmarks CPU/GPU using different models on the same image recognition task Imagenet 2012
Justin Johnson, 2017 https://github.com/jcjohnson/cnn-benchmarks

Is it worth to move from CPU to GPU?

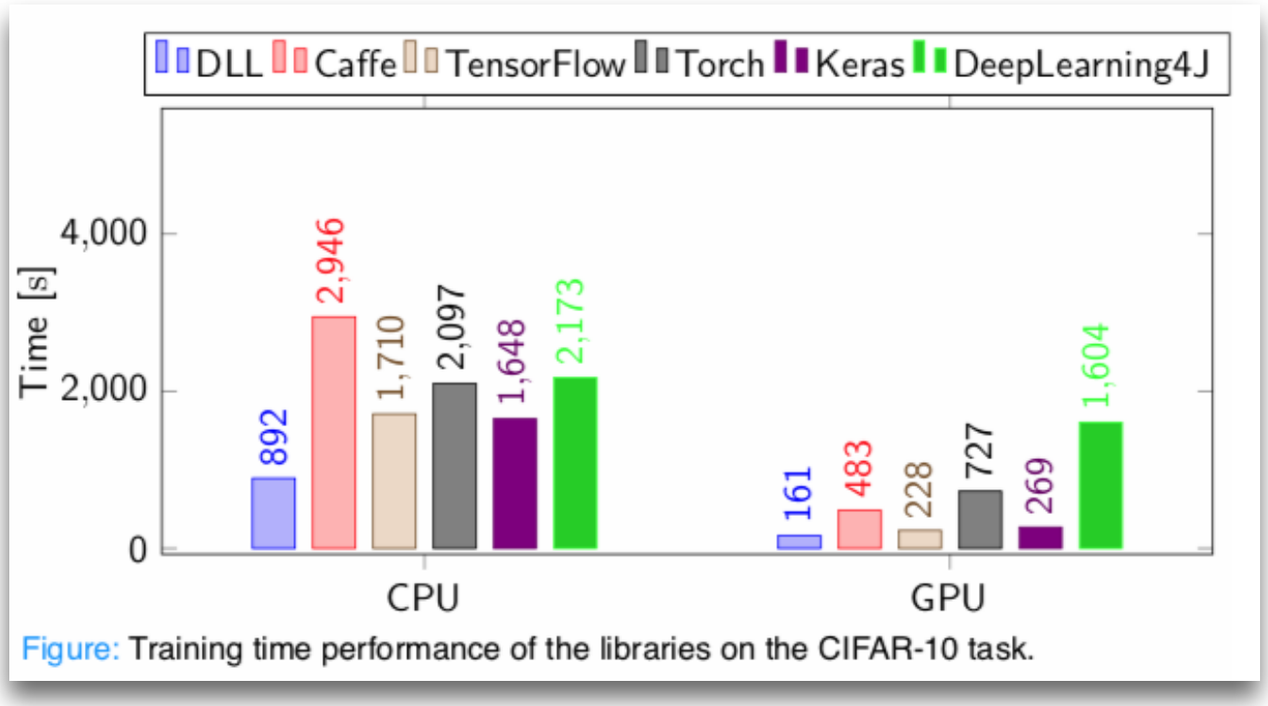


Figure: Training time performance of the libraries on the CIFAR-10 task.

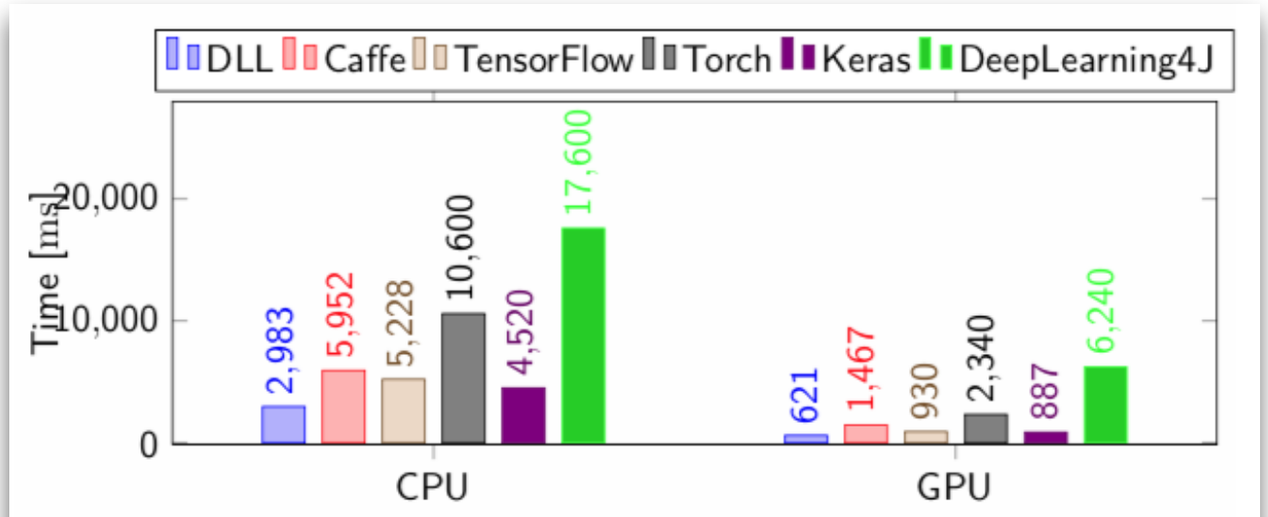


Figure: Training time performance of the libraries, on ImageNet. The time is the average time necessary for the training of one batch of 128 elements.

In this benchmark, we compared our own carefully optimised CPU implementation - DLL against the CPU versions of major frameworks. We also compared CPU-GPU on a recent processor Intel CoreTM i7-2600.

Findings

- CPU speedups can be gained relying on Intel Math Kernel Libraries and other optimisations such as data memory management.
- GPU-CPU speedups are not so impressive as the ones of previous page when using optimised CPU implementations.

Torch shows slower perf than TF. This is probably due to the nature of the models used (CNN). The static graphs strategy of TF is better on feedforward networks, vs dynamic graphs of Torch.

From B. Wicht, A. Fischer and J. Hennebert, “DLL: A Fast Deep Neural Network Library”, pp. 54-65, ANNPR 2018, Springer

What GPU to buy?

- Difficult to answer as technology evolves very quickly and as it depends to the types of inputs and models you are using most.

From Tim Dettmers, August 2018:
Which GPU(s) to Get for Deep Learning
http://timdettmers.com/2018/08/21/which-gpu-for-deep-learning/

Best GPU overall: RTX 2080 Ti
Cost-efficient but expensive: RTX 2080, GTX 1080
Cost-efficient and cheap: GTX 1070, GTX 1070 Ti, GTX 1060
I work with datasets > 250GB: RTX 2080 Ti or RTX 2080
I have little money: GTX 1060 (6GB)
I have almost no money: GTX 1050 Ti (4GB) or CPU (prototyping) + AWS/TPU (training)
I do Kaggle: GTX 1060 (6GB) for prototyping, AWS for final training; use fastai library
I am a competitive computer vision researcher: GTX 2080 Ti; upgrade to RTX Titan in 2019
I am a researcher: RTX 2080 Ti or GTX 10XX -> RTX Titan — check the memory requirements of your current models
I want to build a GPU cluster: This is really complicated, you can get some ideas here
I started deep learning and I am serious about it: Start with a GTX 1060 (6GB) or a cheap GTX 1070 or GTX 1070 Ti if you can find one. Depending on what area you choose next (startup, Kaggle, research, applied deep learning) sell your GPU and buy something more appropriate
I want to try deep learning, but I am not serious about it: GTX 1050 Ti (4 or 2GB)

GPU bottleneck

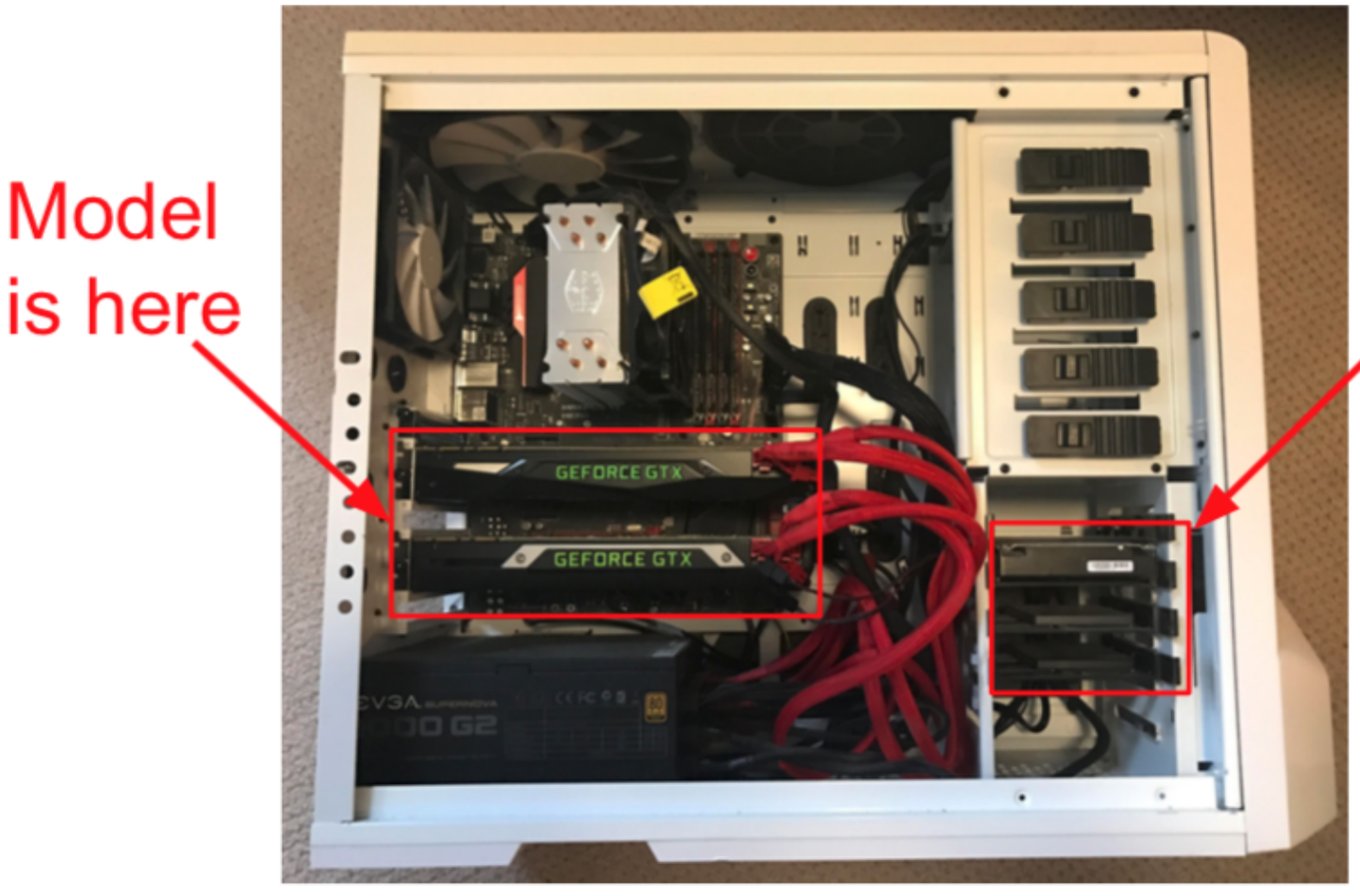
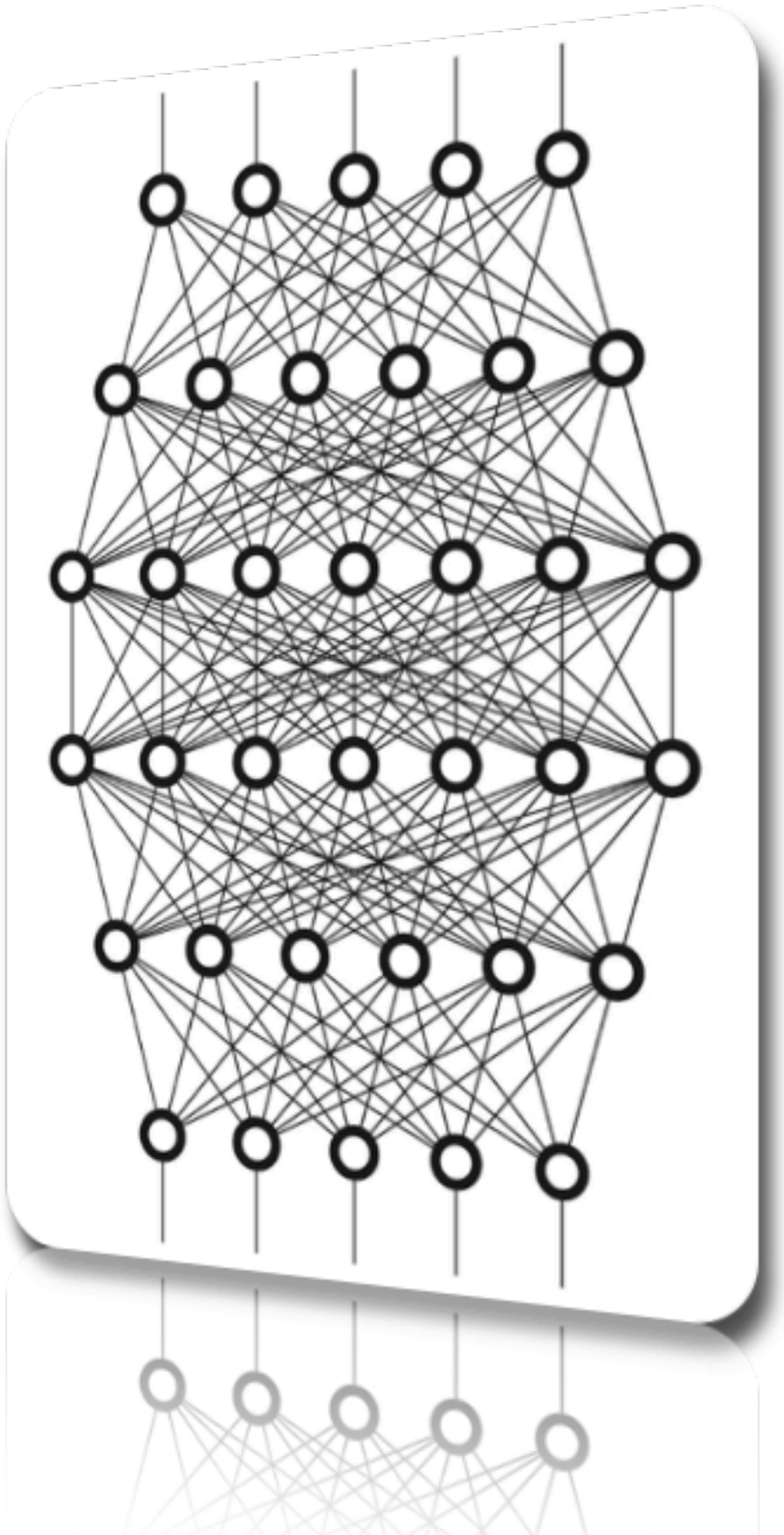


Photo from Justin Johnson, April 2018: CS231n Stanford

- In deep learning, the transfer of data from disk to GPU can be the bottleneck
- Solutions:
  - Read all data into GPU/CPU RAM
  - Use SSDs
  - Use CPU threads to prefetch the data
    - This is done for you in all good frameworks

Computational Graph Implementations



Recaps

Example

Implementation strategies

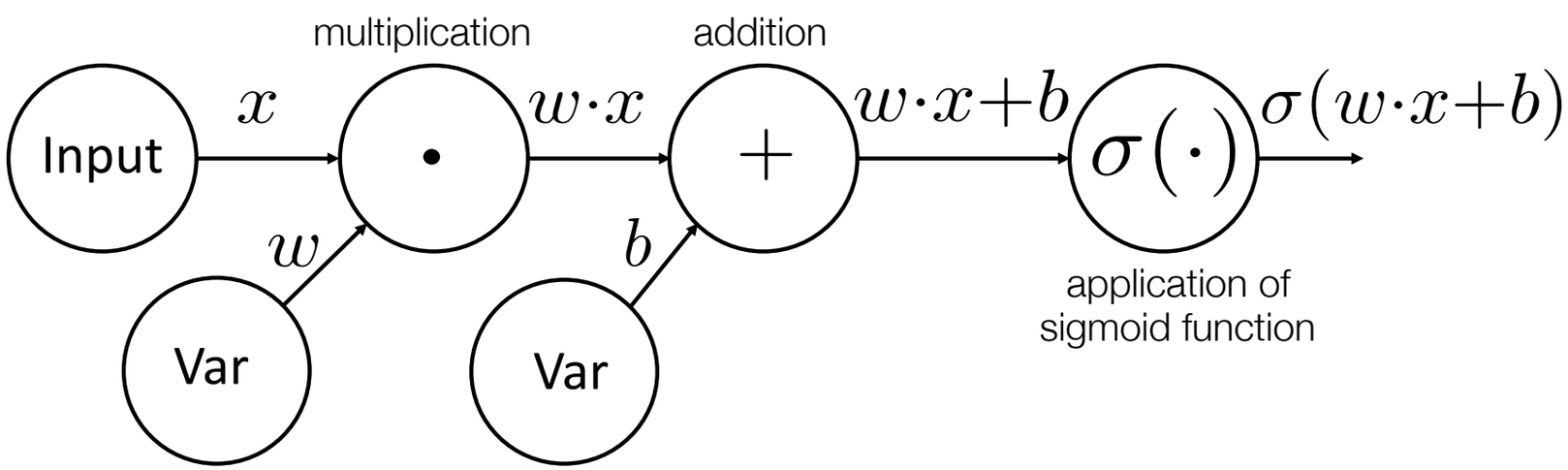
This Section inspired from CS231 Stanford class
https://youtu.be/d14TUNcbn1k

Computational Graphs

Recap from backprob chapter

A **computational graph** is a directed graph where the **nodes** correspond to operations or variables. Variables can feed their value into operations, and operations can feed their output into other operations. This way, every node in the graph defines a function of the variables.

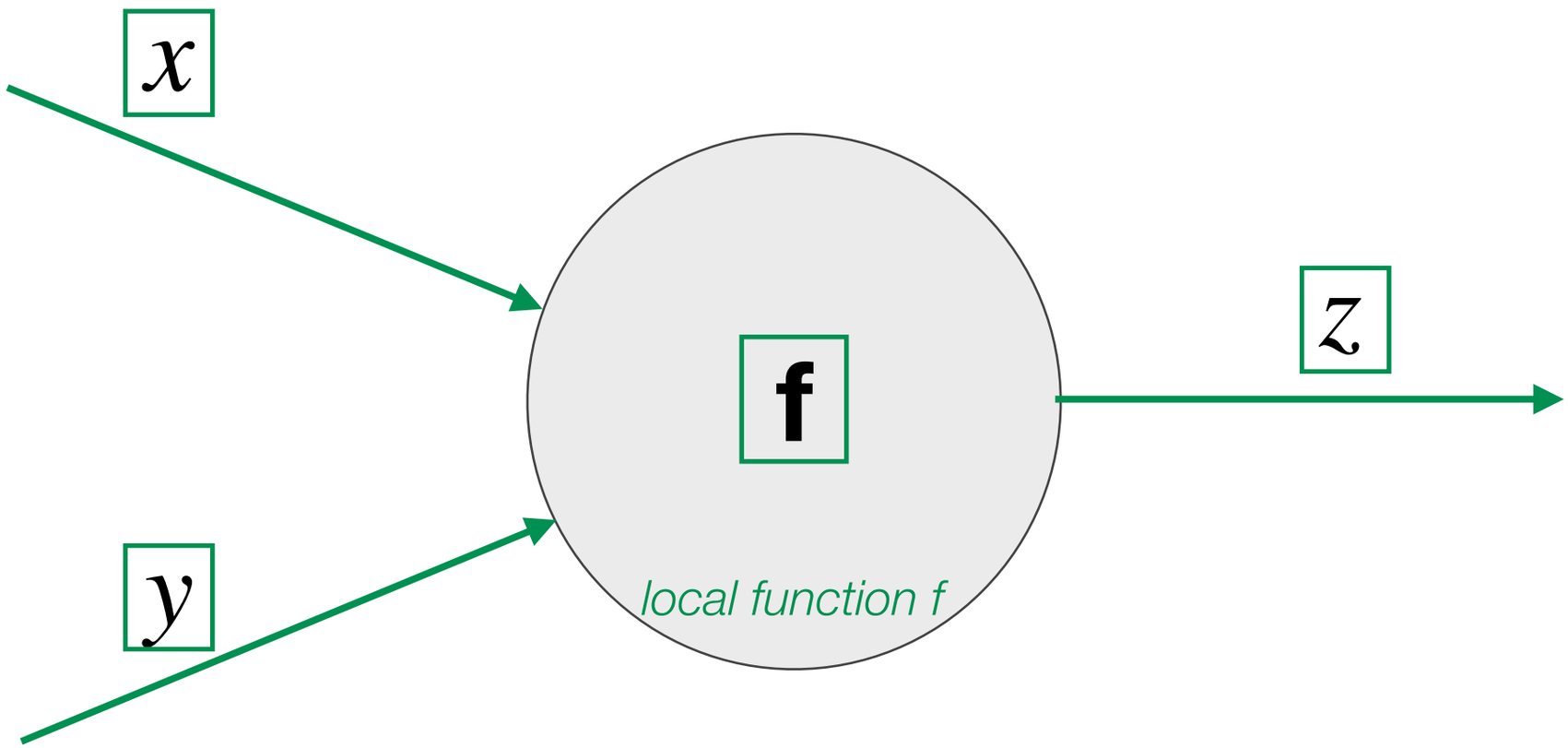
Example: $g(x; w, b) = \sigma(w \cdot x + b)$



Level of function performed at the nodes is not precisely defined: Simple addition, sigmoid function vector operations, …

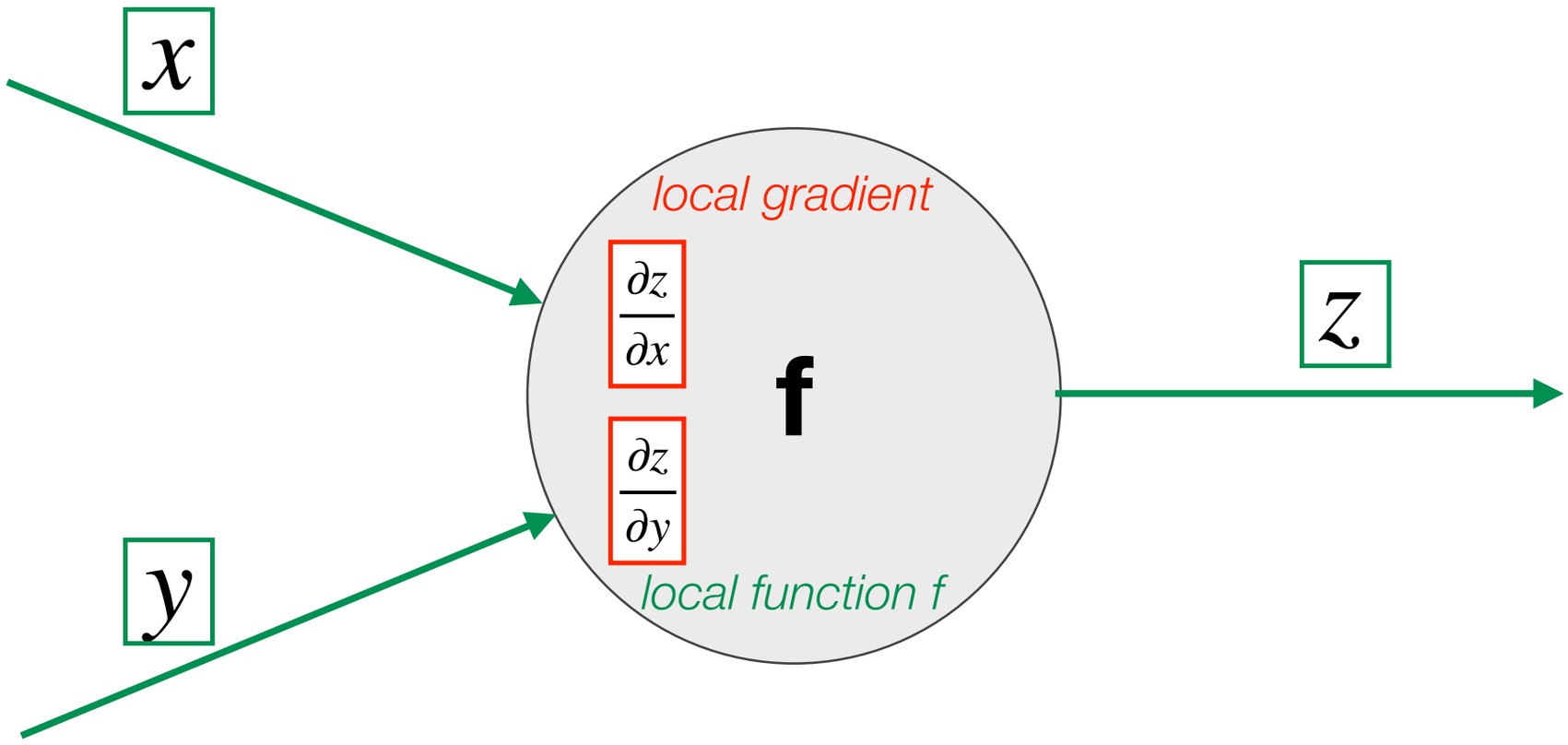
See http://www.deepideas.net/deep-learning-from-scratch-i-computational-graphs/

Node - forward / backward principles

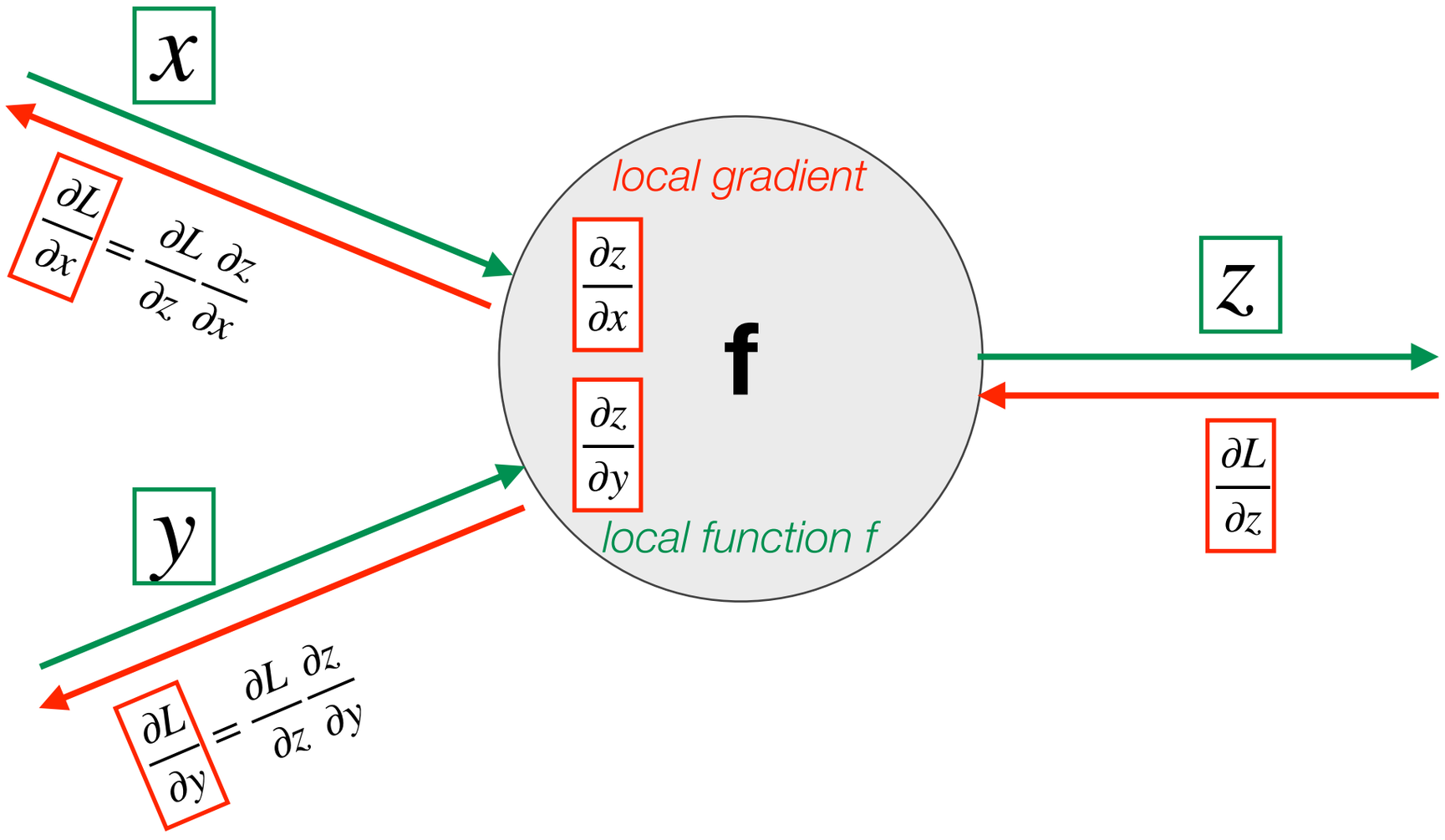


Forward = propagate the signal in the graph

Node - forward / backward principles



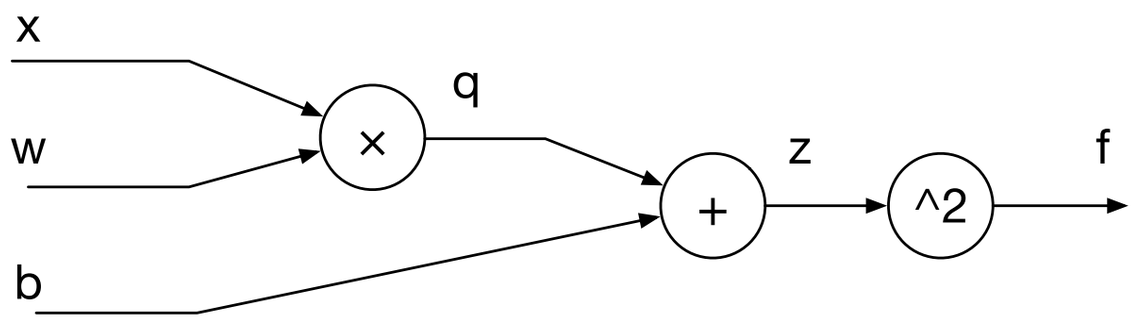
Node - forward / backward principles



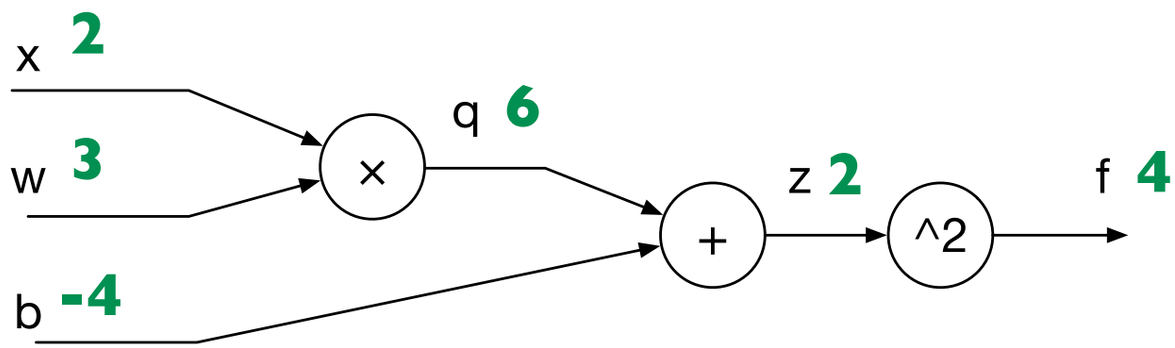
Backward = back-propagate the gradient in the graph with the chain rule

A simple example

$$f = (wx + b)^2$$



- The input can be propagated ***forward*** through the graph
- E.g. x=2, w=3, b=-4



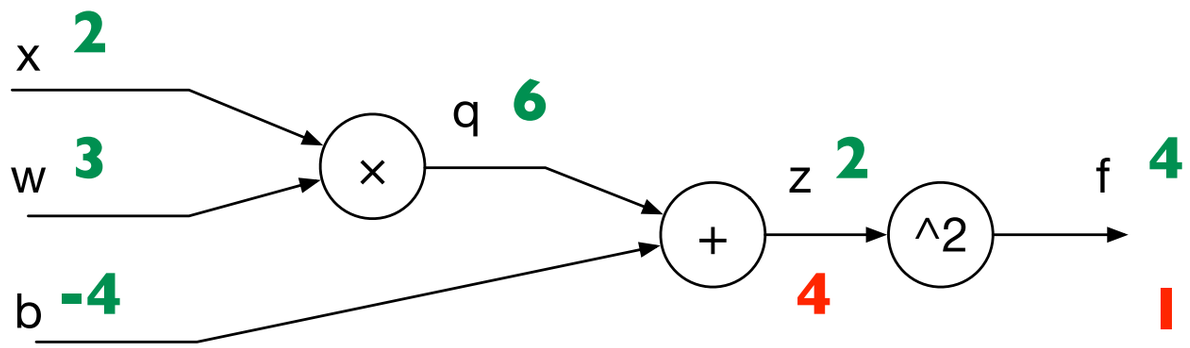
A simple example

$$f = (wx + b)^2$$

$$f = z^2 \quad \frac{\partial f}{\partial z} = 2z$$

$$z = q + b \quad \frac{\partial z}{\partial q} = 1, \frac{\partial z}{\partial b} = 1$$

$$q = wx \quad \frac{\partial q}{\partial x} = w, \frac{\partial q}{\partial w} = x$$



- The chain rule can be applied to each node, e.g. $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial b}$
- The gradient can be propagated ***backward*** by multiplying the gradient at the output of the node with the gradient of the node w.r.t. the input.

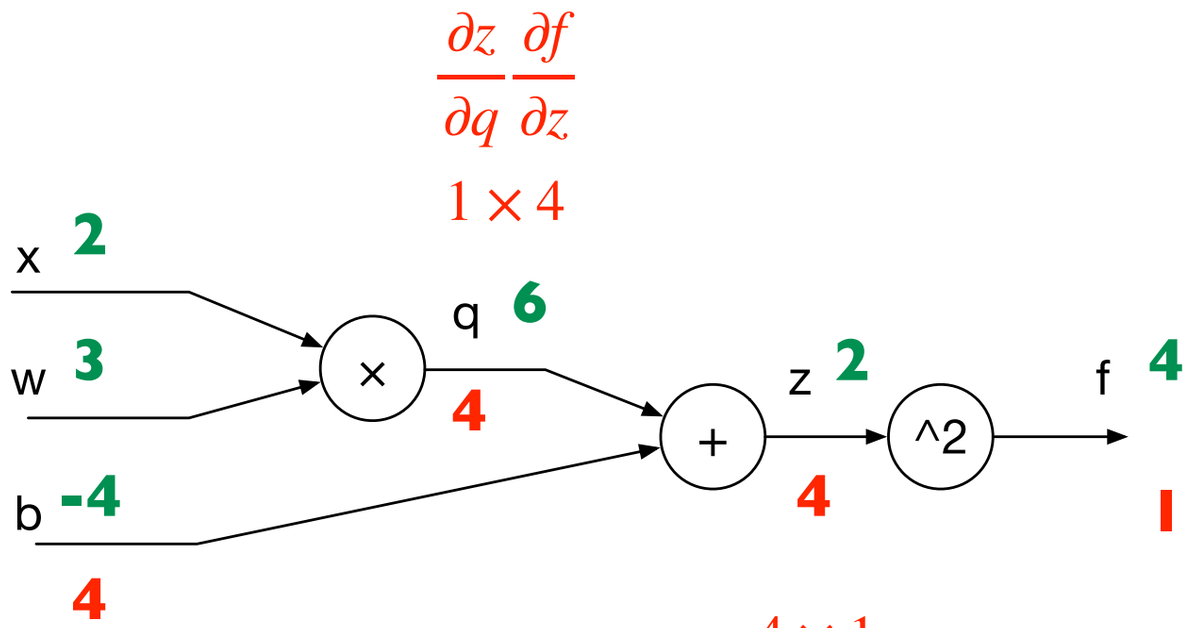
A simple example

$$f = (wx + b)^2$$

$$f = z^2 \quad \frac{\partial f}{\partial z} = 2z$$

$$z = q + b \quad \frac{\partial z}{\partial q} = 1, \frac{\partial z}{\partial b} = 1$$

$$q = wx \quad \frac{\partial q}{\partial x} = w, \frac{\partial q}{\partial w} = x$$



- The chain rule can be applied to each node, e.g. $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial b}$
- The gradient can be propagated ***backward*** by multiplying the gradient at the output of the node with the gradient of the node w.r.t. the input.

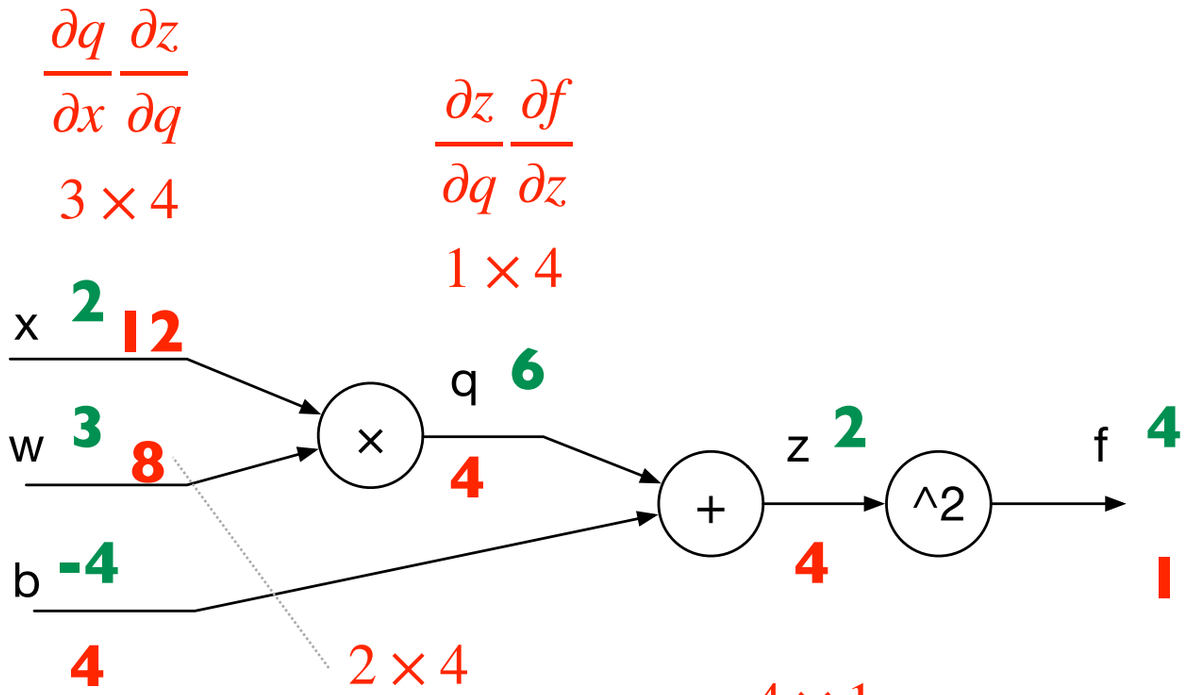
A simple example

$$f = (wx + b)^2$$

$$f = z^2 \quad \frac{\partial f}{\partial z} = 2z$$

$$z = q + b \quad \frac{\partial z}{\partial q} = 1, \frac{\partial z}{\partial b} = 1$$

$$q = wx \quad \frac{\partial q}{\partial x} = w, \frac{\partial q}{\partial w} = x$$



- The chain rule can be applied to each node, e.g. $\frac{\partial f}{\partial b} = \frac{\partial f}{\partial z}\frac{\partial z}{\partial b}$
- The gradient can be propagated ***backward*** by multiplying the gradient at the output of the node with the gradient of the node w.r.t. the input.

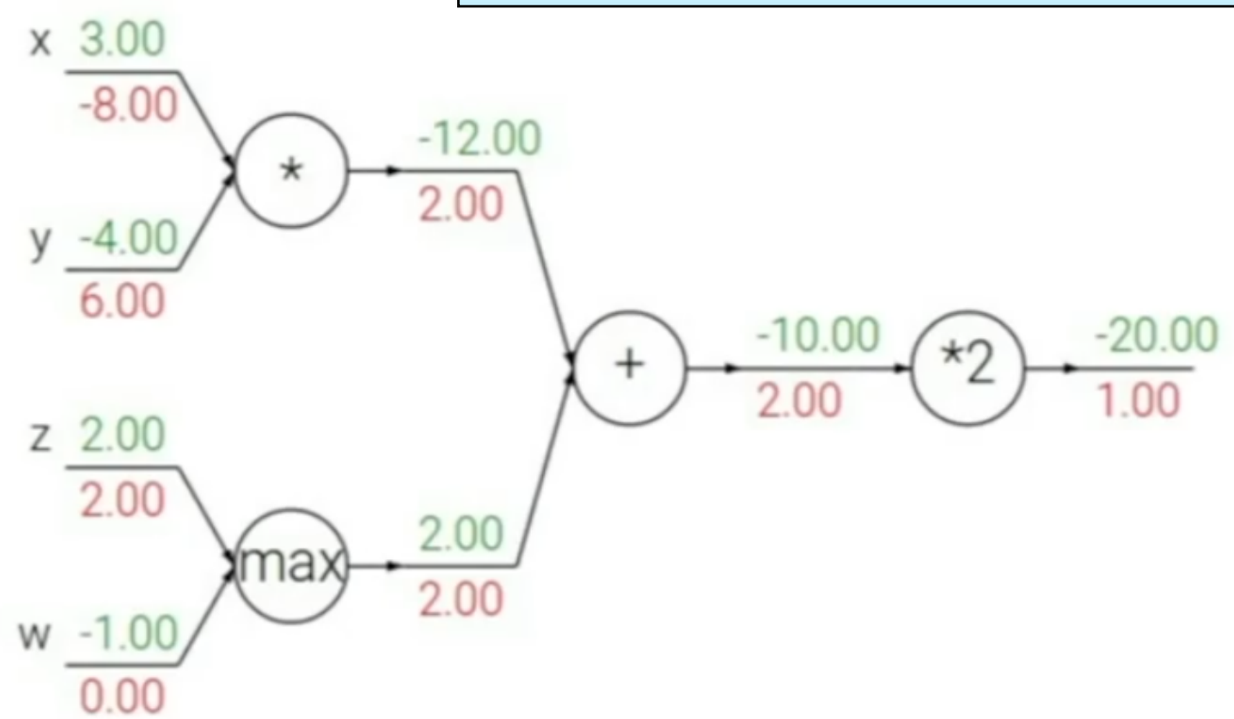
Patterns in backward flow

Advantage I: Intuitive interpretation of gradient backpropagation

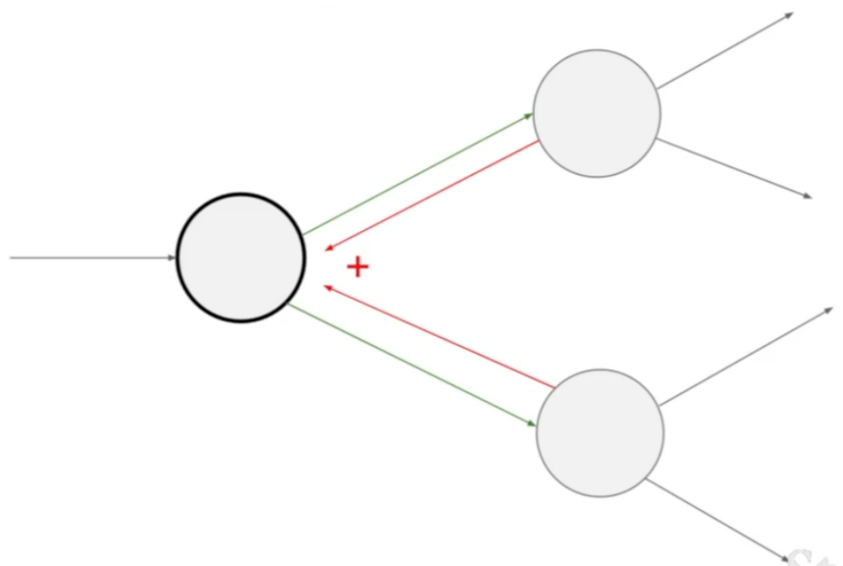
add gate: gradient distributor

max gate: gradient router

mul gate: gradient switcher



gradients add at branches

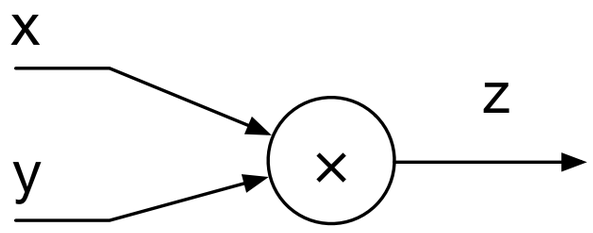


From Fei-Fei Li & Justin Johnson & Serena Yeung, April 2018: CS231n Stanford

Modularized implementation

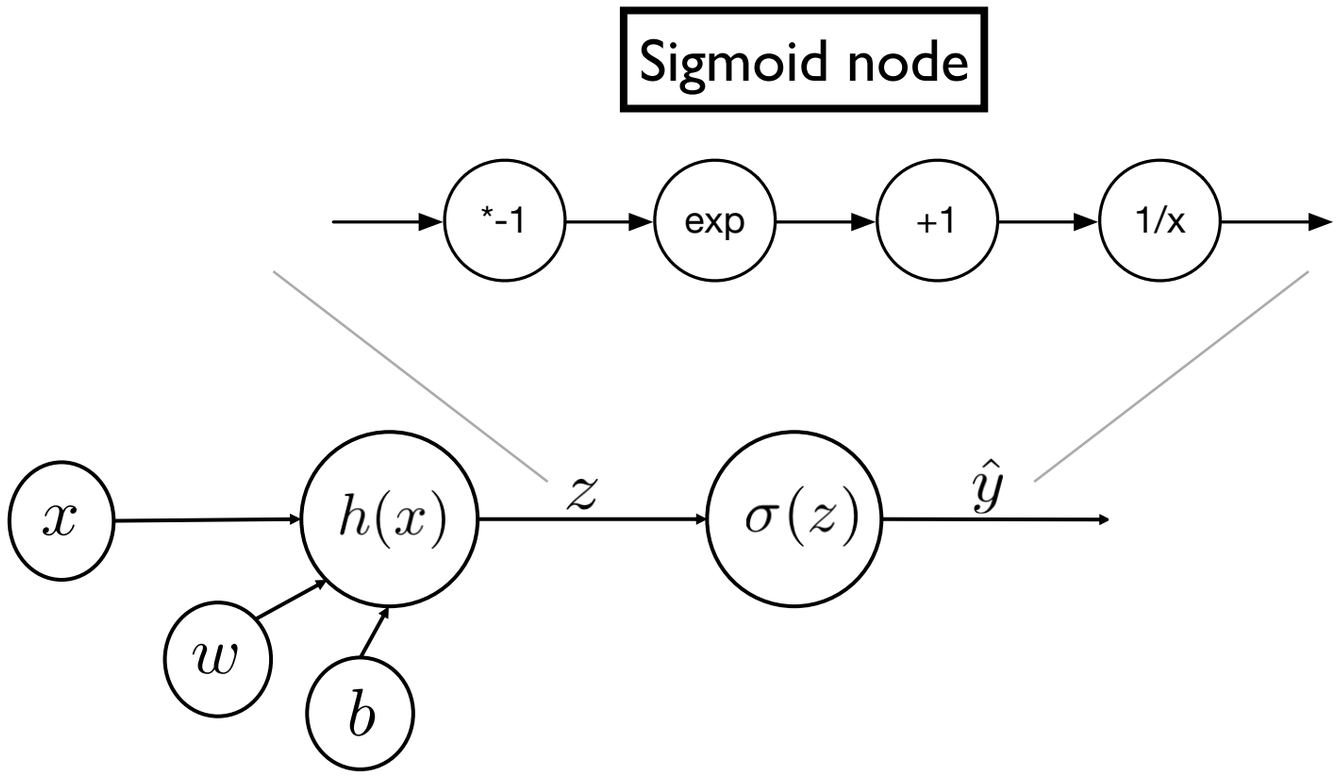
Advantage 2: We can easily define new nodes using the forward/backward pattern

- All nodes will follow the same design pattern




```
class MultiplyNode(object):

    def forward(x, y):
        self.x = x   # must be kept for when backward is called
        self.y = y
        z = x * y
        return z

    def backward(grad_z):
        grad_x = grad_z * self.y   # dL/dz * dz/dx
        grad_y = grad_z * self.x   # dL/dz * dz/dy
        return [grad_x, grad_y]
```

Nodes composition or factorisation



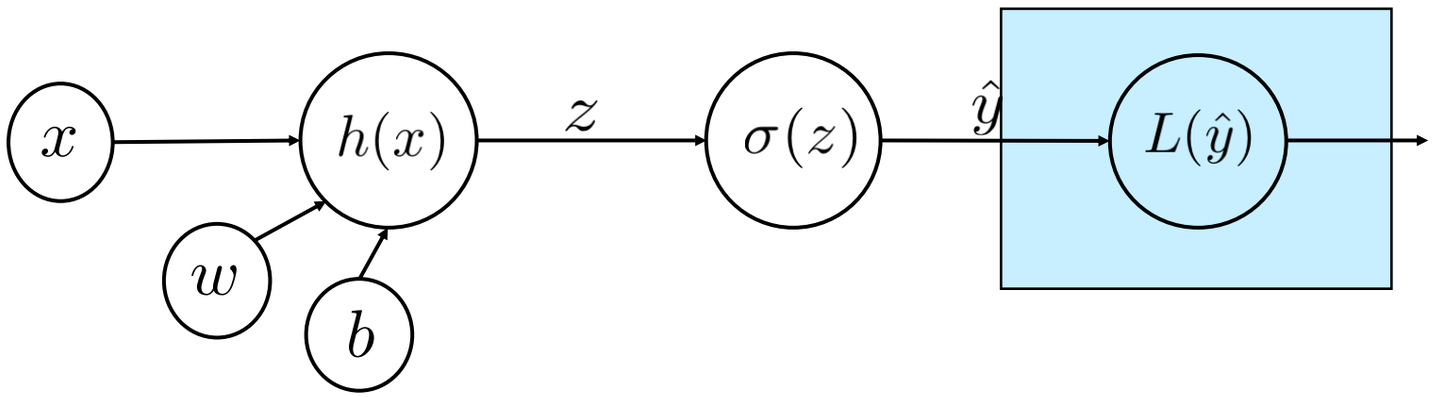
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- Meta-nodes can be composed of nodes
- Or equivalently, a sub-graph composed of nodes can be re-implemented in a single node if we can compute an analytic form of the gradients

$$\frac{\partial \sigma}{\partial z} = (1 - \sigma(z))\sigma(z)$$

Advantages 3 & 4: Node composition or factorisation: any complex learning architecture can be composed from atomic nodes. No need to compute complex global gradient.

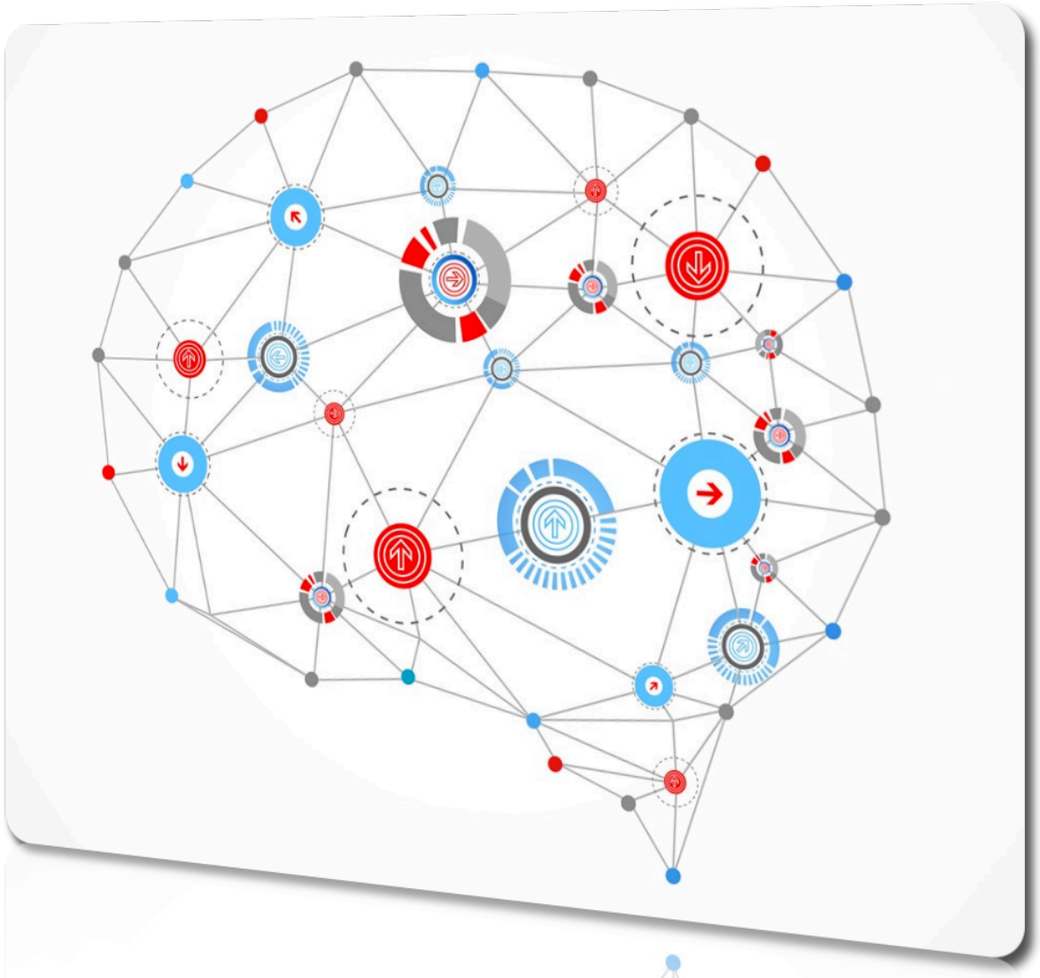
The loss is also composed from nodes



- The loss computation is “plugged” at the end of the graph and constitutes a sub-graph composed of nodes
  - The loss value is computed in the forward pass
  - The gradient is back-propagated from the loss

Advantage 5: The loss functions can actually be seen as extra nodes in the graph (update rules too).

Deep Learning Frameworks



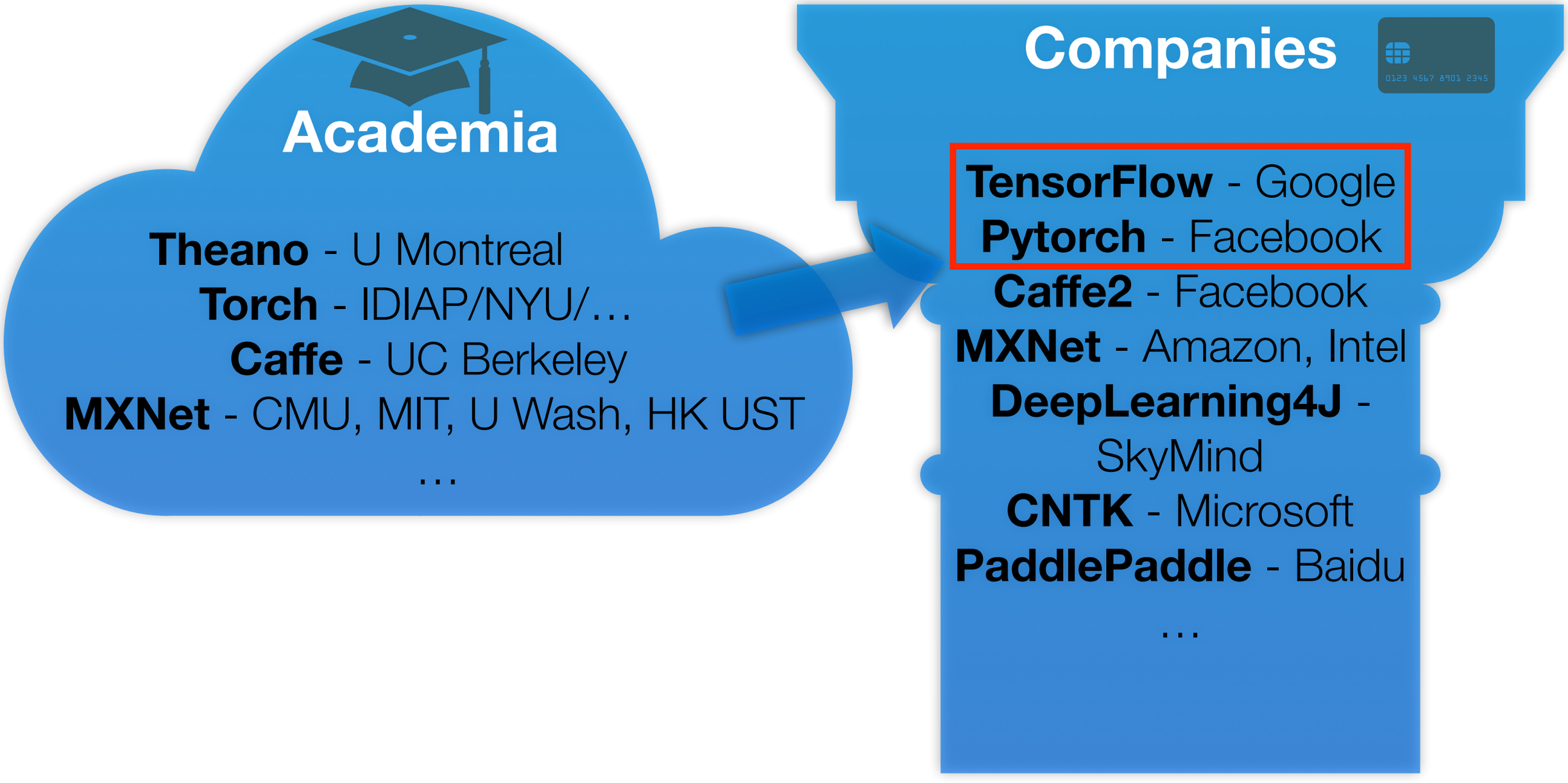
A zoo of frameworks

Our choice

Tensorflow tutorial

A look at the code in other frameworks

A zoo of frameworks for deep learning



See more on https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

Tensorflow vs. Pytorch

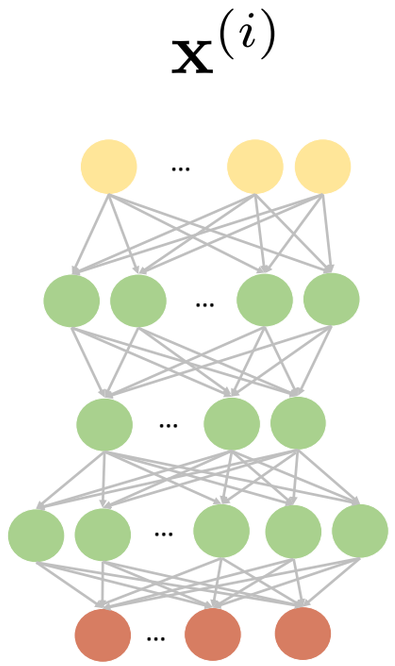
- From Google
- Based on Theano
- **Static** computational graph strategy
  - e.g. faster on CNN, more cumbersome on variable input length such as in RNN
- Steeper learning curve, people usually use wrappers (Keras)
- Bigger community
- Better to go in production mode, seems more accepted in the industry

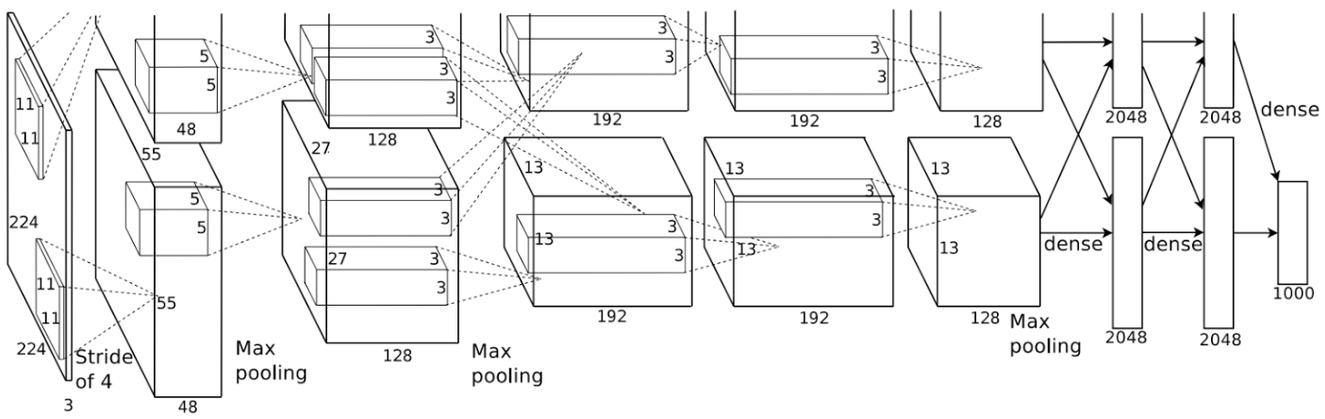
- From Facebook
- Based on Torch
- **Dynamic** computational graph strategy
  - e.g. faster on RNN and slower on static architectures
- Closer to python code, more “pythonic”
- Younger, smaller community
- Better to do rapid prototyping, seems to be well accepted in the research community

For this year’s DL class we will chose Tensorflow + Keras

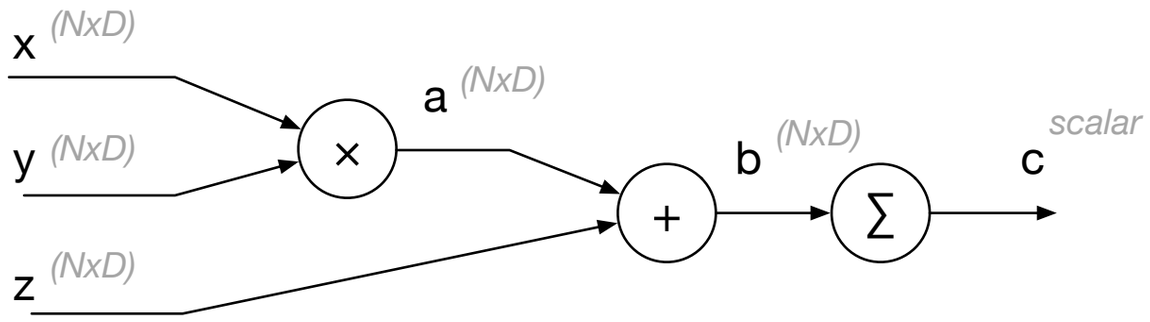
4 main reasons to use DL frameworks

1. Easily build big computational graphs
2. Easily compute losses $L(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$ and gradients in computational graphs for update rules $\text{param} \leftarrow \text{param} - \alpha \frac{\partial L}{\partial \text{param}}$
3. Have at hand all the state-of-the-art strategies for regularisations and optimisations
4. Switch easily from cpu to gpu when needed





TensorFlow - simple example




```
import numpy as np
import tensorflow as tf
```

At the beginning of each example in the following pages.

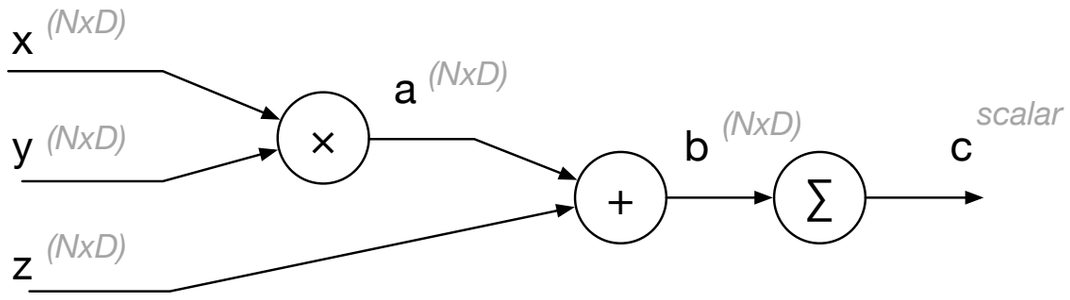
```
# simple computational graph - numpy
N, D = 3, 4
np.random.seed(0)
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y  # shape (N, D)
b = a + z  # shape (N, D)
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
```

We compute ourselves the gradients! Automatic in TensorFlow

```
# simple computational graph with tensorflow
N, D = 3, 4
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=(N, D))

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
```

Example inspired from Fei-Fei Li & Justin Johnson & Serena Yeung, April 2018: CS231n Stanford

TensorFlow - simple example




```python
# simple computational graph - numpy
N, D = 3, 4
np.random.seed(0)
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y  # shape (N, D)
b = a + z  # shape (N, D)
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
```

Just one line to run the graph on gpu

```python
# simple computational graph - tensorflow on gpu
N, D = 3, 4
with tf.device('/gpu:0'): # '/cpu:0' for cpu exec
    x = tf.placeholder(tf.float32, shape=(N, D))
    y = tf.placeholder(tf.float32, shape=(N, D))
    z = tf.placeholder(tf.float32, shape=(N, D))

    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
```

TensorFlow - simple example

Numpy

```python
import numpy as np

# simple computational graph - numpy
N, D = 3, 4
np.random.seed(0)
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y  # shape (N, D)
b = a + z  # shape (N, D)
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
```

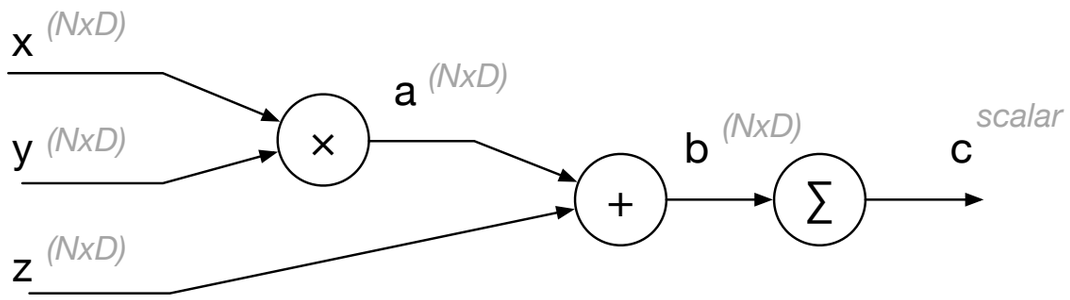
Tensorflow

```python
import numpy as np
import tensorflow as tf

# simple computational graph - tensorflow on gpu
N, D = 3, 4
with tf.device('/gpu:0'): # '/cpu:0' for cpu exec
    x = tf.placeholder(tf.float32, shape=(N, D))
    y = tf.placeholder(tf.float32, shape=(N, D))
    z = tf.placeholder(tf.float32, shape=(N, D))

    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
```

Pytorch

```python
import torch
from torch.autograd import Variable

# simple computational graph - torch on gpu
N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
y = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)
z = Variable(torch.randn(N, D).cuda(),
             requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

Tensorflow - simple example



First define the computational graph

```
# simple computational graph - tensorflow on gpu
N, D = 3, 4
with tf.device('/gpu:0'): # '/cpu:0' for cpu exec
    x = tf.placeholder(tf.float32, shape=(N, D))
    y = tf.placeholder(tf.float32, shape=(N, D))
    z = tf.placeholder(tf.float32, shape=(N, D))

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

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

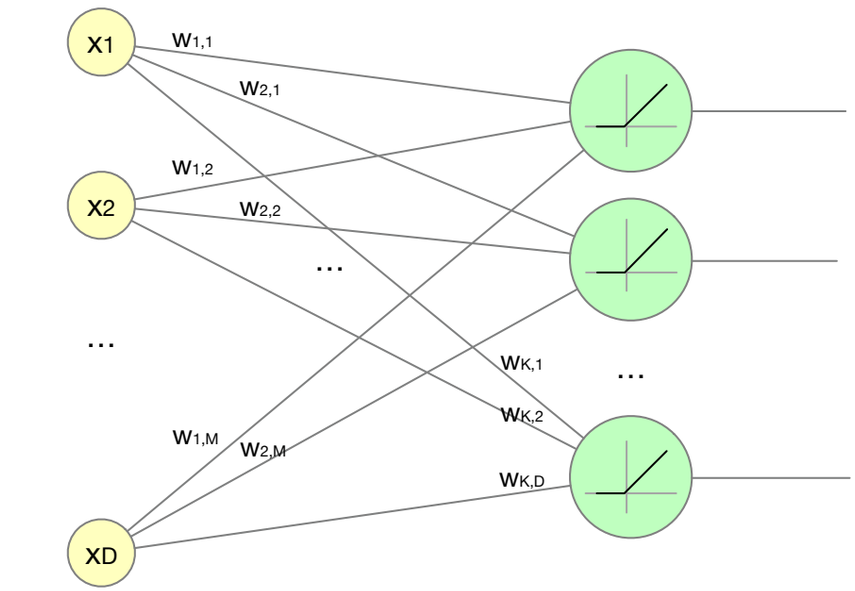
Then run the graph many times

```
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
```

This is the “static” graph strategy of TensorFlow. The graph structure is defined and then sits on the gpu. Quite efficient for graph where the structure is not dynamic, e.g. CNN. Optimisations can also be applied on the graph if it is static.

TensorFlow - a more complex example

Assuming a classification problem with training set in x_train and target variables 1-hot in y_train.



Training with MSE loss

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.placeholder(tf.float32, shape=(D, n_classes))

y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1 = tf.gradients(loss, [w1])

with tf.Session() as sess:
    values = {
        x: x_train,
        y: y_train,
        w1: np.random.randn(D, n_classes)
    }
    out = sess.run([loss, grad_w1],
                   feed_dict=values)
    loss_val, grad_w1_val = out
```

TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

First define the computational graph.
No computation, just building the graph.

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.placeholder(tf.float32, shape=(D, n_classes))

y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1 = tf.gradients(loss, [w1])
```

Then run the graph.

```
with tf.Session() as sess:
    values = {
        x: x_train,
        y: y_train,
        w1: np.random.randn(D, n_classes)
    }
    out = sess.run([loss, grad_w1],
                   feed_dict=values)
    loss_val, grad_w1_val = out
```

TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

Create placeholders for input x, output y, weights

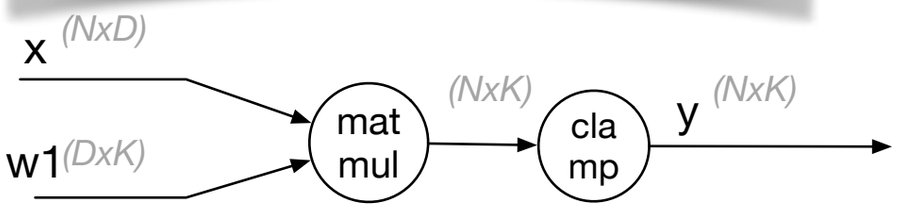
Placeholders are fed with data external to the graph when running the graph.

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.placeholder(tf.float32, shape=(D, n_classes))

y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1 = tf.gradients(loss, [w1])

with tf.Session() as sess:
    values = {
        x: x_train,
        y: y_train,
        w1: np.random.randn(D, n_classes)
    }
    out = sess.run([loss, grad_w1],
                   feed_dict=values)
    loss_val, grad_w1_val = out
```

TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

Computational graph creation: forward pass



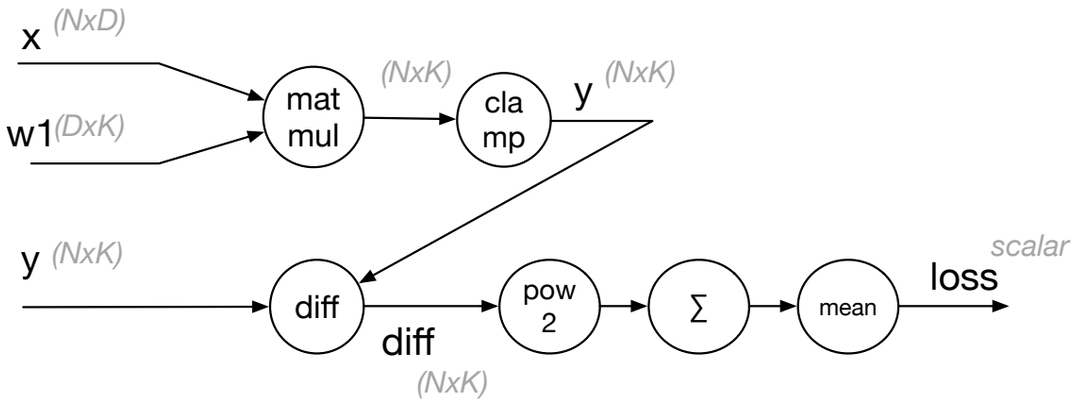

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.placeholder(tf.float32, shape=(D, n_classes))

y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1 = tf.gradients(loss, [w1])

with tf.Session() as sess:
    values = {
        x: x_train,
        y: y_train,
        w1: np.random.randn(D, n_classes)
    }
    out = sess.run([loss, grad_w1],
                   feed_dict=values)
    loss_val, grad_w1_val = out
```

TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.placeholder(tf.float32, shape=(D, n_classes))

y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1 = tf.gradients(loss, [w1])

with tf.Session() as sess:
    values = {
        x: x_train,
        y: y_train,
        w1: np.random.randn(D, n_classes)
    }
    out = sess.run([loss, grad_w1],
                   feed_dict=values)
    loss_val, grad_w1_val = out
```

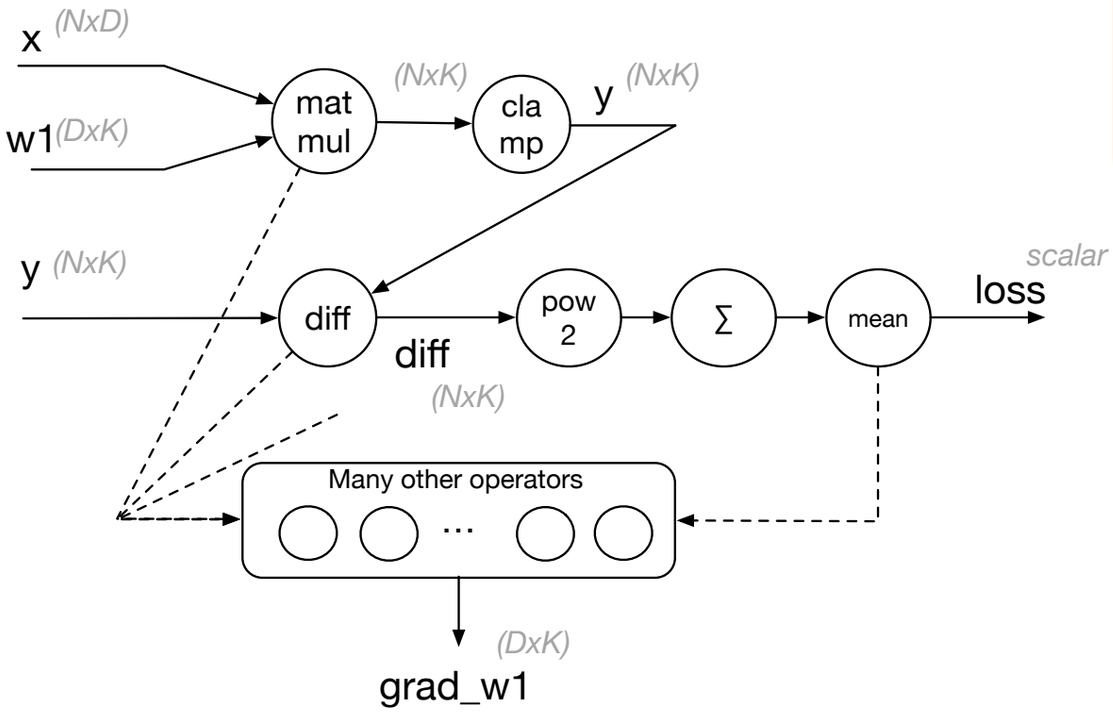
Computational graph creation: forward pass and loss



TensorFlow - a more complex example

Below: 1 forward pass on training set, loss computation and gradient of loss w.r.t weights w1

Computational graph creation: forward pass, loss and gradients




```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.placeholder(tf.float32, shape=(D, n_classes))

y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1 = tf.gradients(loss, [w1])

with tf.Session() as sess:
    values = {
        x: x_train,
        y: y_train,
        w1: np.random.randn(D, n_classes)
    }
    out = sess.run([loss, grad_w1],
                   feed_dict=values)
    loss_val, grad_w1_val = out
```

TensorFlow - a more complex example

Below: I forward pass on training set, loss computation and gradient of loss w.r.t weights w1

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.placeholder(tf.float32, shape=(D, n_classes))

y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1 = tf.gradients(loss, [w1])

with tf.Session() as sess:
    values = {
        x: x_train,
        y: y_train,
        w1: np.random.randn(D, n_classes)
    }
    out = sess.run([loss, grad_w1],
                   feed_dict=values)
    loss_val, grad_w1_val = out
```

Now we enter a session to run the graph.

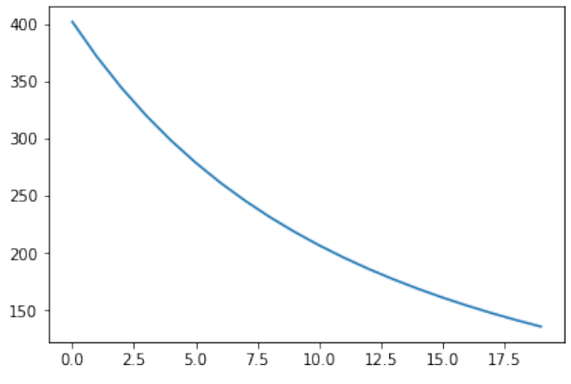
Feed the placeholders with x, y and w1

Run graph and get outputs: loss and grad_w1

TensorFlow - a more complex example

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.placeholder(tf.float32, shape=(D, n_classes))

y_pred = tf.maximum(tf.matmul(x, w1), 0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1 = tf.gradients(loss, [w1])[0] # returns a list of
                                      # sums of gradients

with tf.Session() as sess:
    values = {
        x: x_train,
        y: y_train,
        w1: np.random.randn(D, n_classes)
    }
    alpha = 1e-3
    J = []
    for epoch in range(20):
        out = sess.run([loss, grad_w1], feed_dict=values)
        loss_val, grad_w1_val = out
        values[w1] -= alpha * grad_w1_val
        J.append(loss_val)
        print("epoch", epoch, loss_val)
    pl.plot(J)
```

Now train for 20 epochs upgrading the weights.



Problem: weights are copied between cpu and gpu at each step.

TensorFlow - a more complex example

Change w1 from **placeholder** (fed on each call) to **Variable** (persist in the graph between calls).

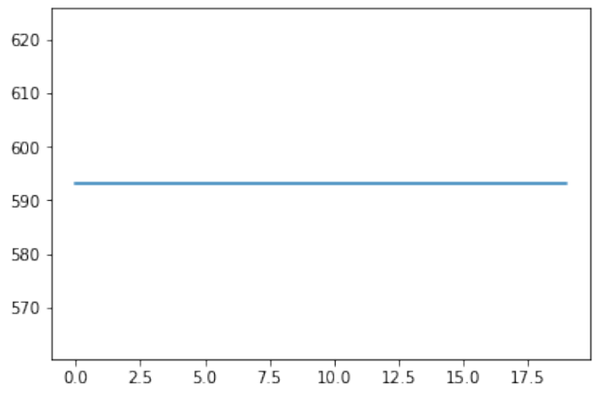
Add **assign** operation to update w1 as part of the graph.

Run once the graph to initialise the variable w1

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.Variable(tf.random_normal((D, n_classes)))

y_pred = tf.maximum(tf.matmul(x, w1), 0.0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1 = tf.gradients(loss, [w1])[0] # returns a list of
                                      # sums of gradients

alpha = 1e-3
new_w1 = w1.assign(w1 - alpha * grad_w1)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = { x: x_train, y: y_train}
    J = []
    for epoch in range(20):
        loss_val = sess.run([loss], feed_dict=values)
        J.append(loss_val)
        print("epoch", epoch, loss_val)
    pl.plot(J)
```

TensorFlow - a more complex example

Problem: the loss is not going down…

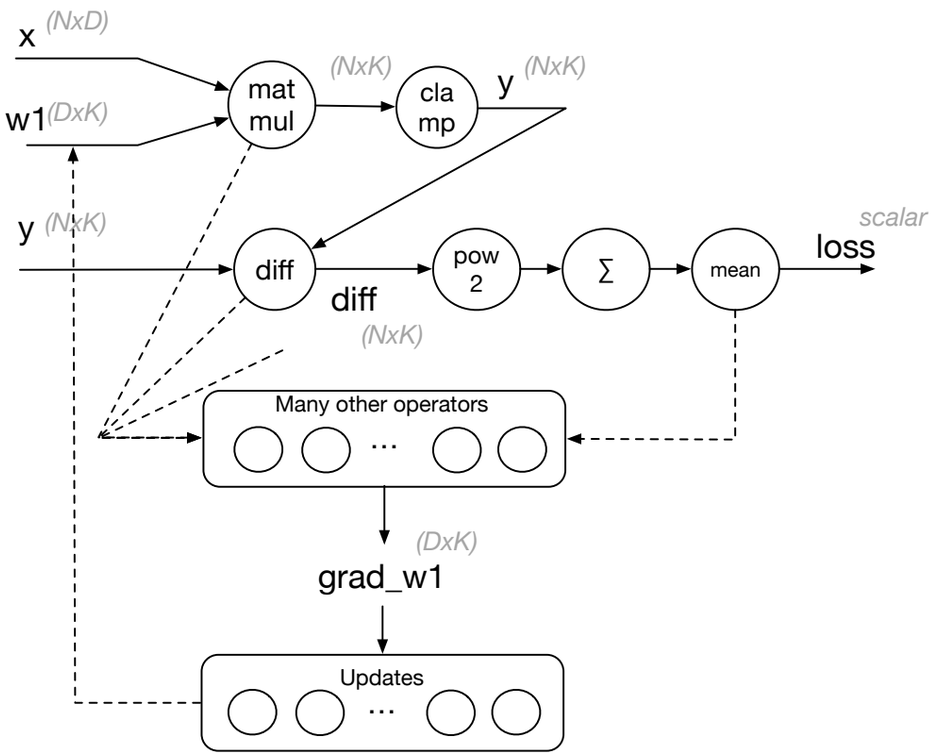


assign calls are not executed

The reason is: when we run the graph, we just ask for the loss, and TF "smartly" optimises by computing only the part of the graph for the loss.

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.Variable(tf.random_normal((D, n_classes)))

y_pred = tf.maximum(tf.matmul(x, w1), 0.0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1 = tf.gradients(loss, [w1])[0] # returns a list of
                                      # sums of gradients

alpha = 1e-3
new_w1 = w1.assign(w1 - alpha * grad_w1)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = { x: x_train, y: y_train}
    J = []
    for epoch in range(20):
        loss_val = sess.run([loss], feed_dict=values)
        J.append(loss_val)
        print("epoch", epoch, loss_val)
    pl.plot(J)
```

TensorFlow - a more complex example



Add dummy graph node that depends on the updates

Ask for updates values in the output of the run.

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))
w1 = tf.Variable(tf.random_normal((D, n_classes)))

y_pred = tf.maximum(tf.matmul(x, w1), 0.0) # ReLU on logit
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1 = tf.gradients(loss, [w1])[0] # returns a list of
                                      # sums of gradients

alpha = 1e-3
new_w1 = w1.assign(w1 - alpha * grad_w1)
updates = tf.group(new_w1)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = { x: x_train, y: y_train}
    J = []
    for epoch in range(20):
        loss_val = sess.run([loss, updates], feed_dict=values)
        J.append(loss_val)
        print("epoch", epoch, loss_val)
    pl.plot(J)
```

TensorFlow - a more complex example

Use pre-defined layers, it automatically init weights (in this case with He init) for us.

Use pre-defined loss functions

Use pre-defined optimizers

```
N = x_train.shape[0]  # number of samples
D = x_train.shape[1]  # dimension of input sample
n_classes = 10        # output dim

x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, n_classes))

init = tf.variance_scaling_initializer(2.0) # He init
y_pred = tf.layers.dense(inputs=x, units=n_classes,
                         activation=tf.nn.relu,
                         kernel_initializer=init)

loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-2)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = { x: x_train, y: y_train}
    J = []
    for epoch in range(20):
        loss_val = sess.run([loss, updates],
                            feed_dict=values)
        J.append(loss_val)
        print("epoch", epoch, loss_val[0])
    pl.plot(J)
```

Wrap-up

- **Significant speedups** can be gained in deep learning moving from **cpu to gpu to tpu**
  - Beware of benchmarks that are sometimes biased
- **Computational graph** implementations are important in deep learning:
  - Intuitive interpretation of gradient backpropagation
  - Easy to define new nodes using the forward/backward pattern
  - Node composition or factorisation: any complex learning architecture can be composed from atomic nodes.
  - No need to compute manually complex global gradient.
  - The loss functions can actually be seen as extra nodes in the graph (update rules too).
- **Deep learning frameworks**
  - It is a zoo! Fast moving landscape.
  - Two big players of 2018 are TensorFlow and Pytorch