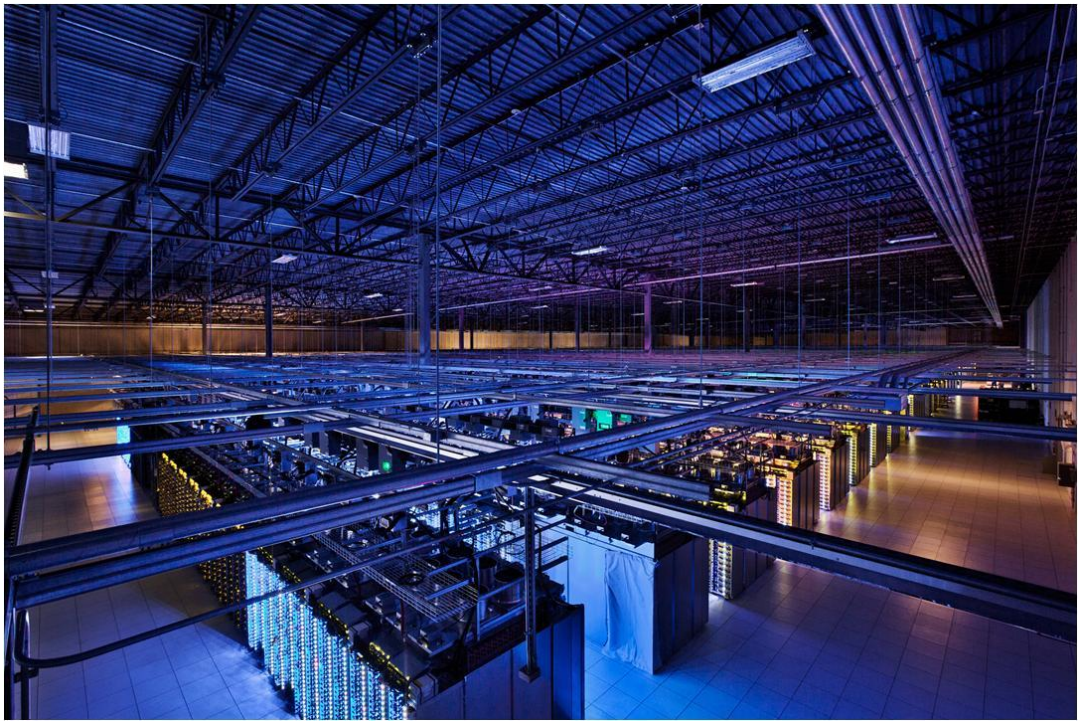
CWI
Centrum Wiskunde & Informatica

Big Data for Data Science

Scalable Machine Learning



event.cwi.nl/lsde

A SHORT INTRODUCTION TO NEURAL NETWORKS

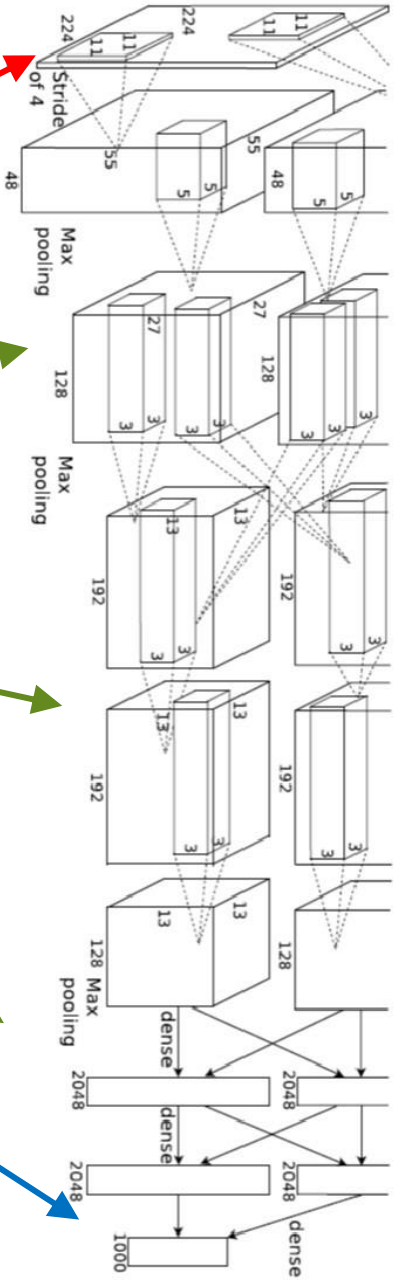
credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Example: Image Recognition

Input Image

Weights

Loss

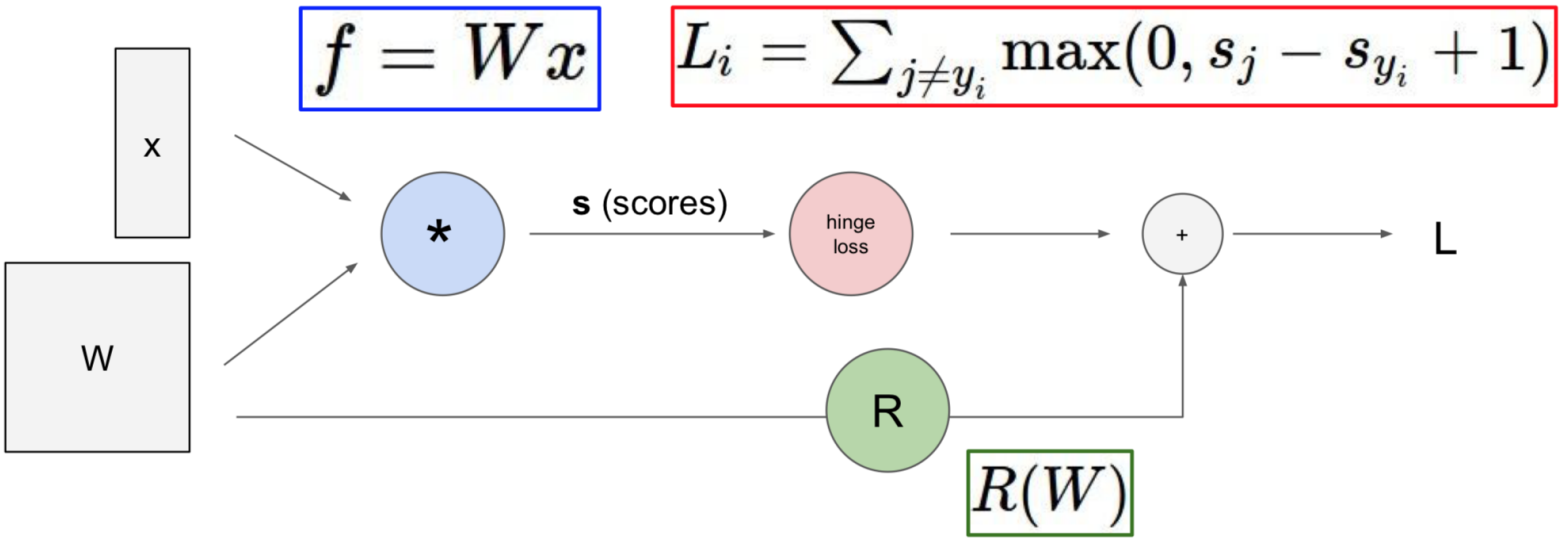


AlexNet
‘convolutional’ neural network

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Neural Nets - Basics

- **Score** function (linear, matrix)
- **Activation** function (normalize [0-1])
- **Regularization** function (penalize complex W)

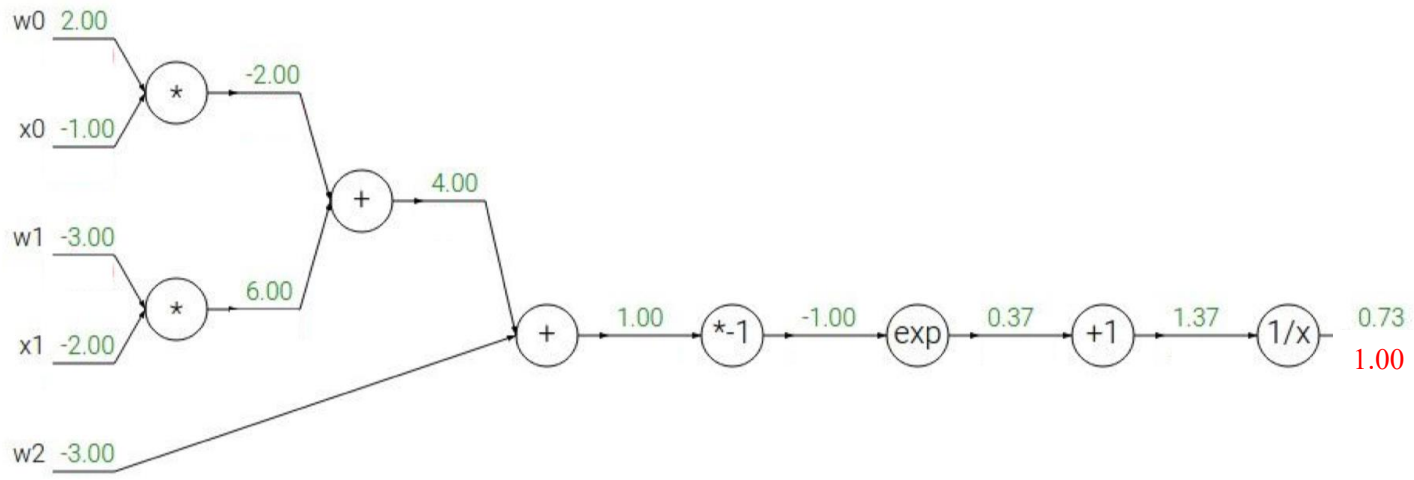


credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Neural Nets are Computational Graphs

- **Score**, **Activation** and **Regularization** together with a **Loss** function

example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



- For backpropagation, we need a formula for the "gradient", i.e. the derivative of each computational function:

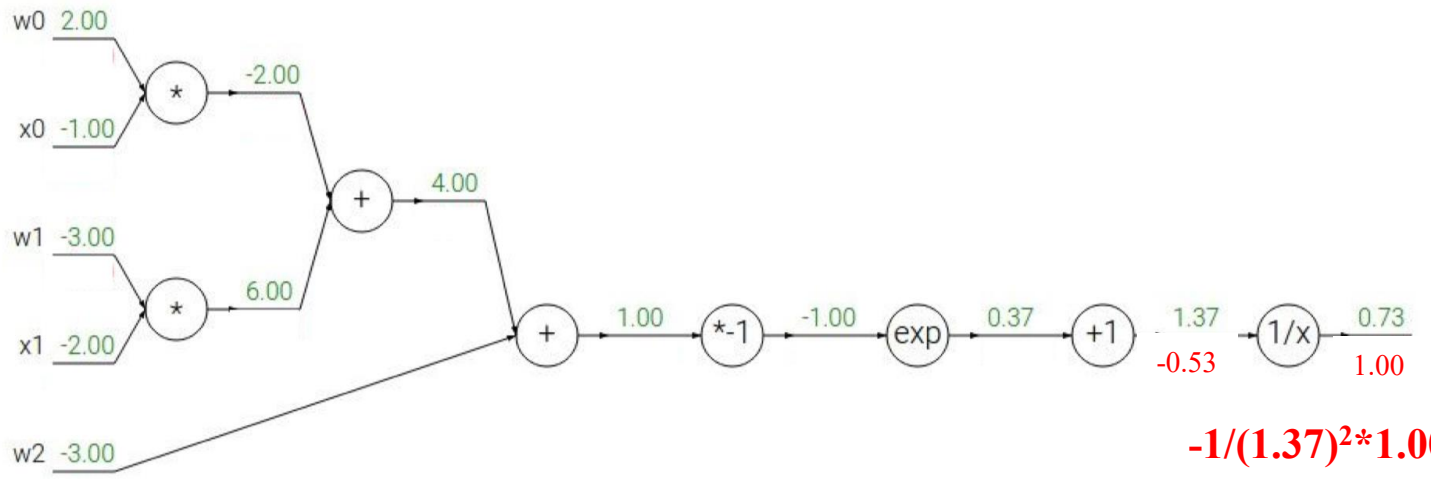
$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Training the model: backpropagation

- backpropagate loss to the weights to be adjusted, proportional to a learning rate

example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$-1/(1.37)^2 * 1.00 = -0.53$

- For backpropagation, we need a formula for the "gradient", i.e. the derivative of each computational function:

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

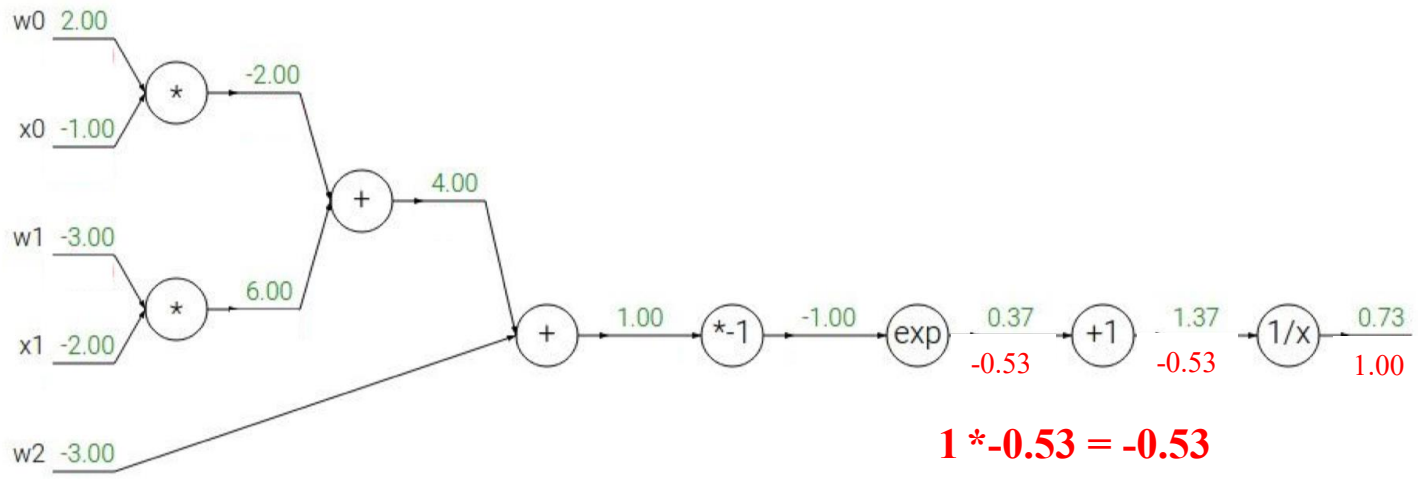
$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung, Song Han)

Training the model: backpropagation

- backpropagate loss to the weights to be adjusted, proportional to a learning

example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



1 *-0.53 = -0.53

- For backpropagation, we need a formula for the "gradient", i.e. the derivative of each computational function:

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

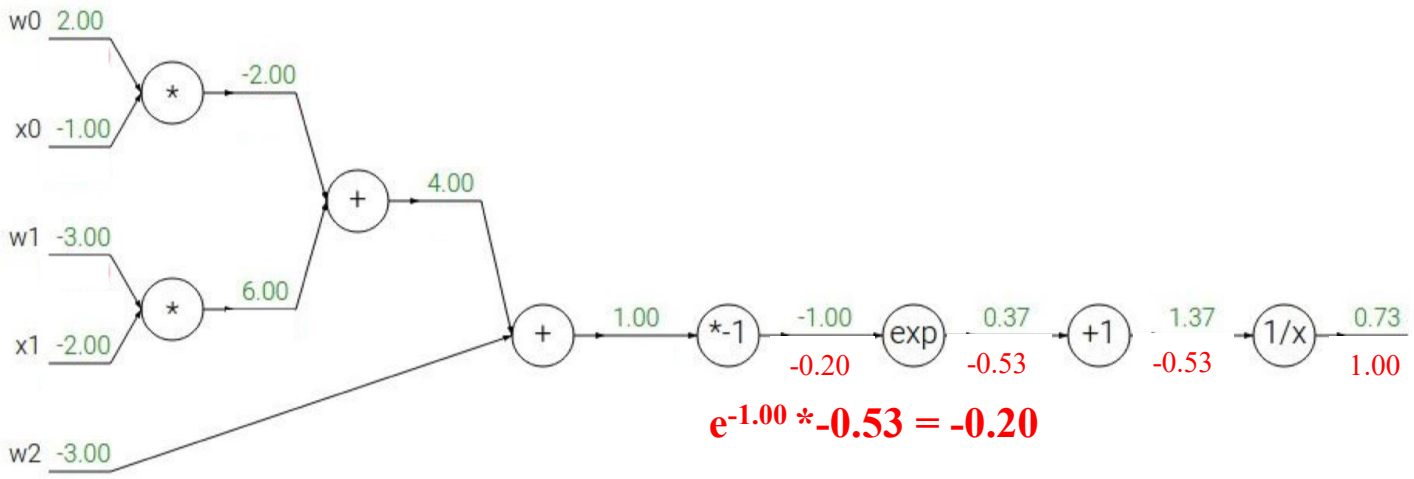
$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung, Song Han)

Training the model: backpropagation

- backpropagate loss to the weights to be adjusted, proportional to a learning

example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$e^{-1.00} * -0.53 = -0.20$

- For backpropagation, we need a formula for the "gradient", i.e. the derivative of each computational function:

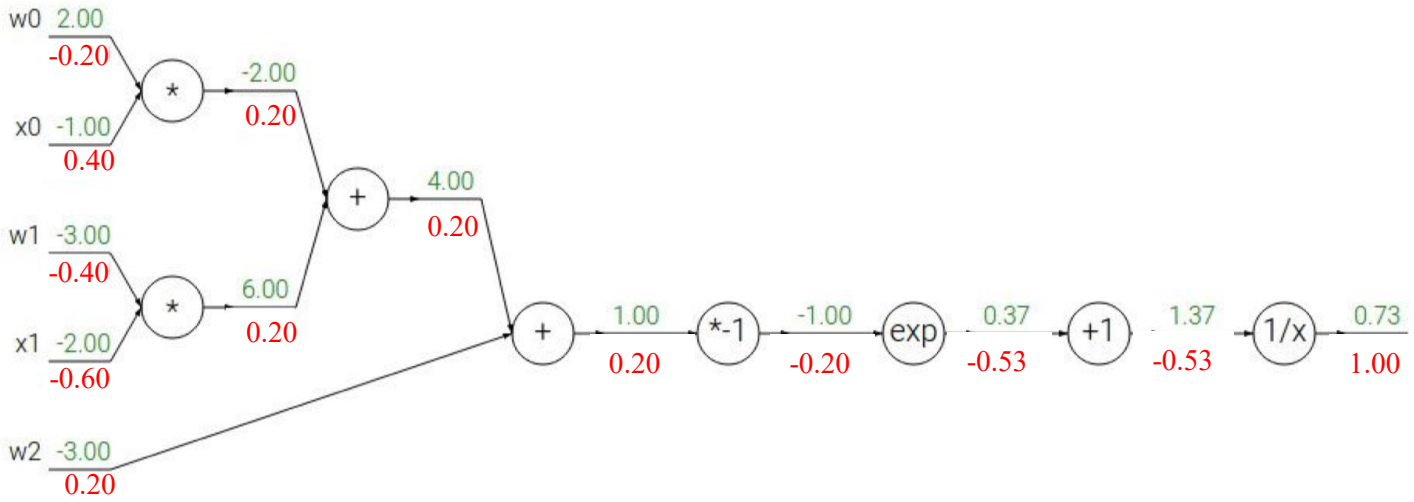
$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Training the model: backpropagation

- backpropagate loss to the weights to be adjusted, proportional to a learning

example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



- For backpropagation, we need a formula for the “gradient”, i.e. the derivative of each computational function:

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x \qquad f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

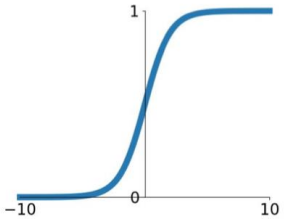
$$f_a(x) = ax \rightarrow \frac{df}{dx} = a \qquad f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Activation Functions

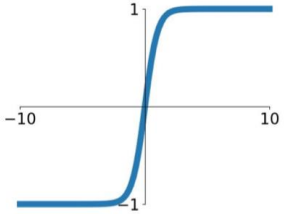
Sigmoid

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



tanh

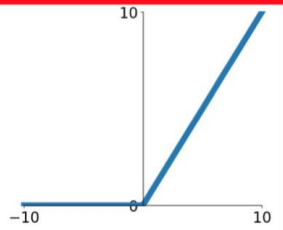
$$\tanh(x)$$



ReLU

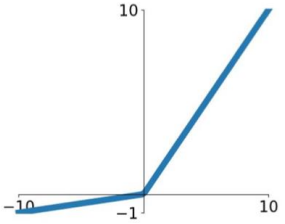
$$\max(0, x)$$

Good default choice



Leaky ReLU

$$\max(0.1x, x)$$

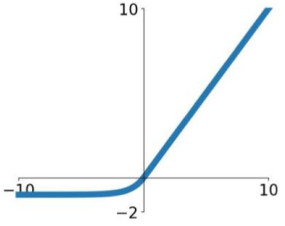


Maxout

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

ELU

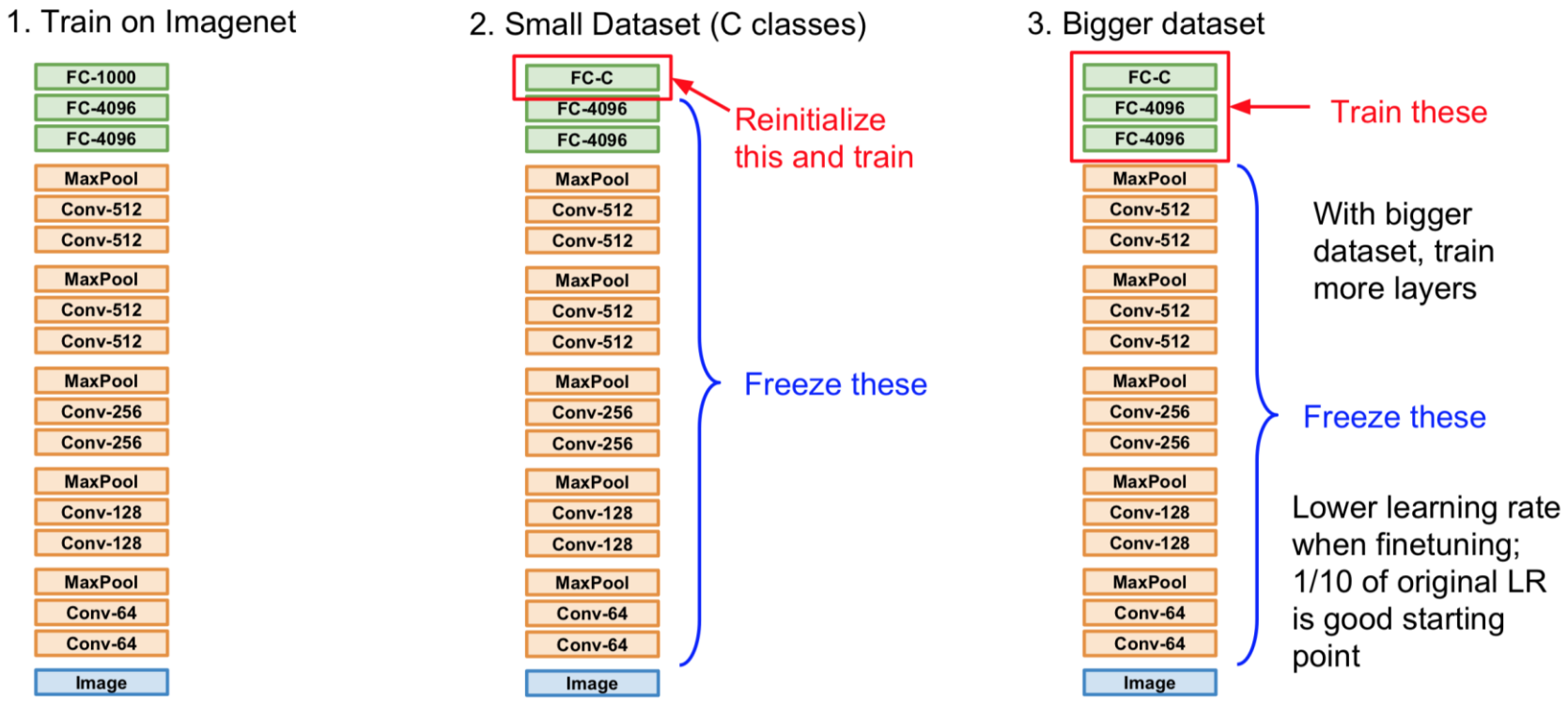
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

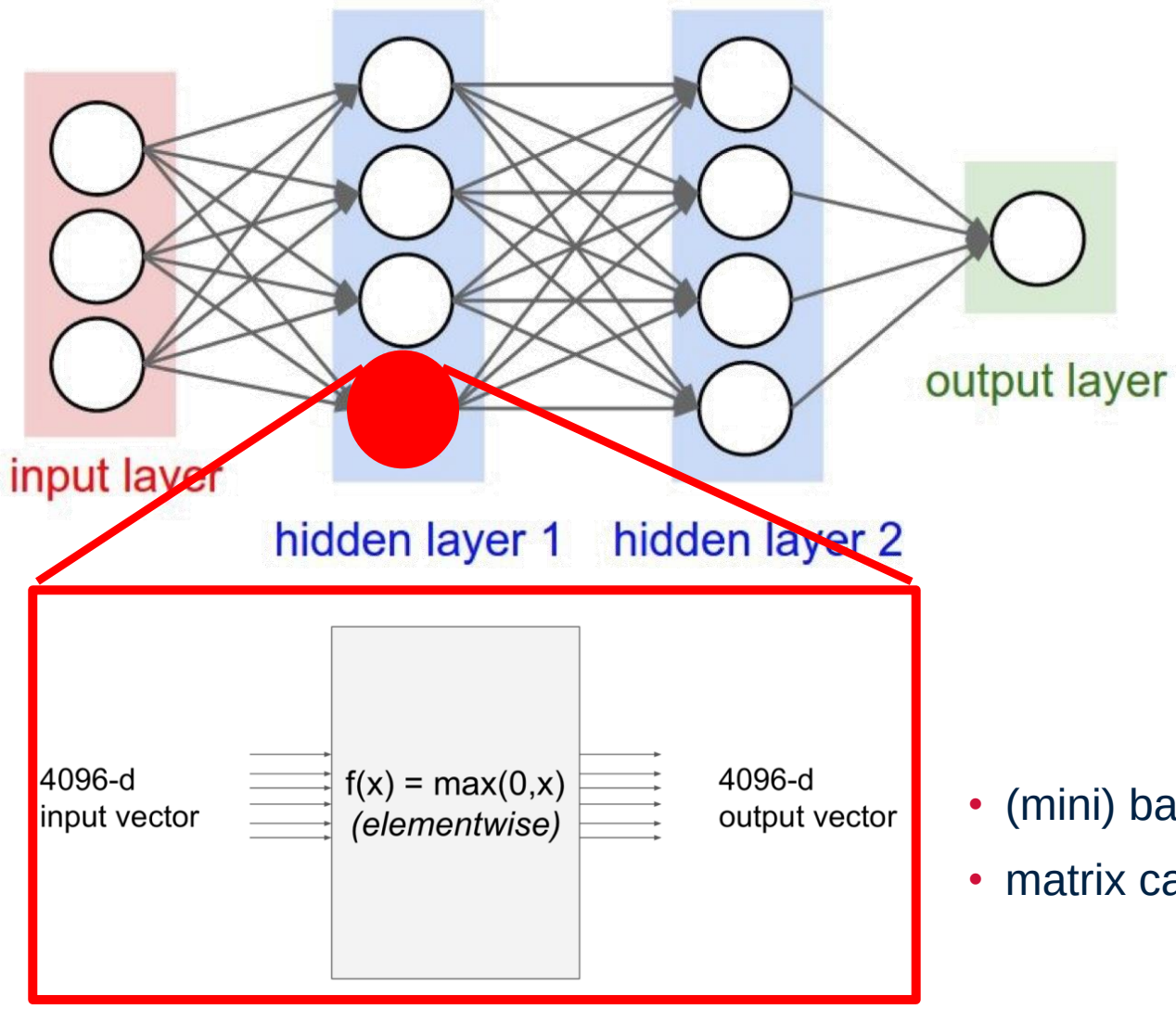
Get going quickly: Transfer Learning

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014



credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Neural Network Architecture



- (mini) batch-wise training
- matrix calculations galore

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

CWI
Centrum Wiskunde & Informatica

DEEP LEARNING SOFTWARE

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Deep Learning Frameworks

Caffe (UC Berkeley)	➔ Caffe2 (Facebook)	Paddle (Baidu)
Torch (NYU/Facebook)	➔. PyTorch (Facebook)	CNTK (Microsoft)
Theano (Univ. Montreal)	➔ TensorFlow (Google)	MXNET (Amazon)

- Easily build big computational graphs
- Easily compute gradients in these graphs
- Run it at high speed (e.g. GPU)

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Deep Learning Frameworks

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
```

..have to compute gradients by hand..

No GPU support

TensorFlow

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

N, D = 3, 4

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
```

..gradient computations are generated automagically from the forward phase (z=x*y; b=a+x; c= sum(b)) + GPU support

PyTorch

```
import torch
from torch.autograd import Variable

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)
```

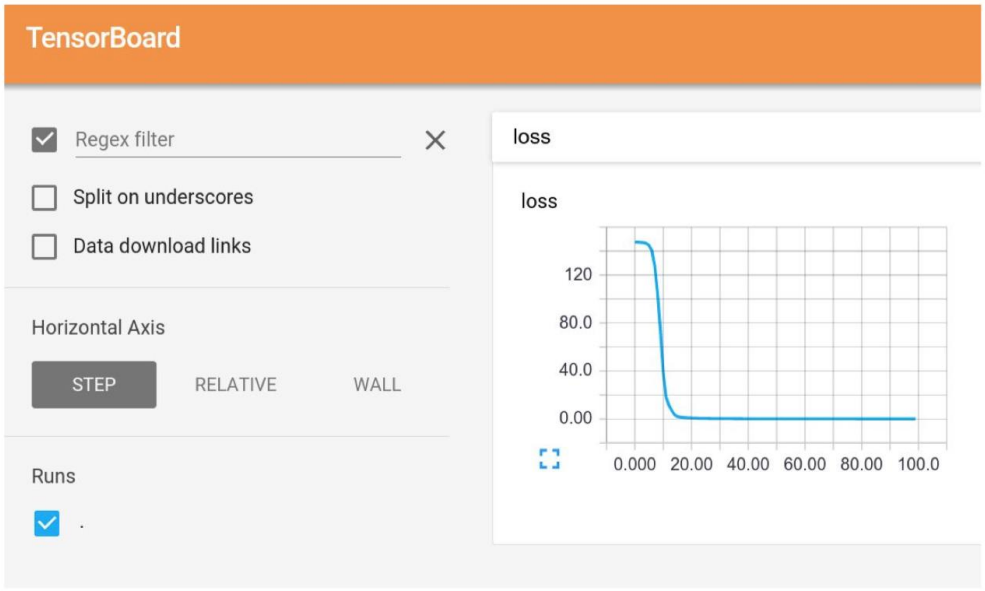
..similar to TensorFlow

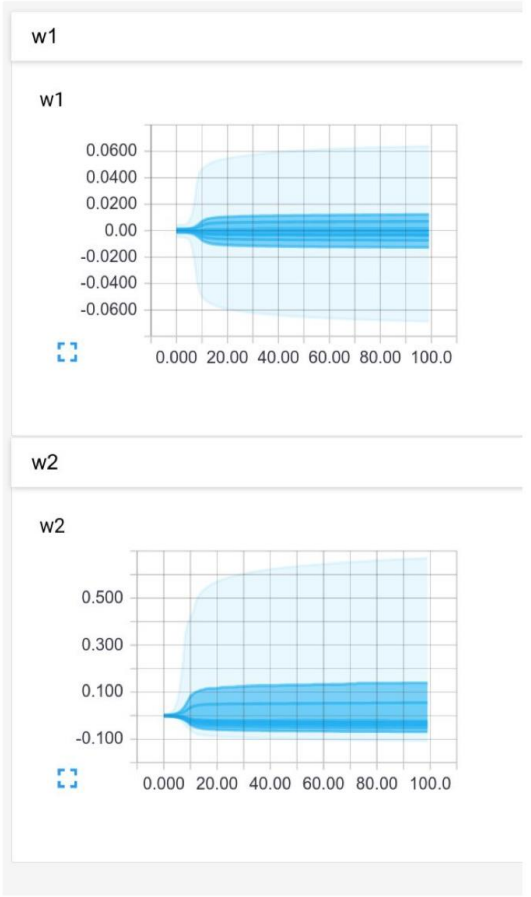
Not a “new language” but embedded in Python (control flow).

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

TensorFlow: TensorBoard GUI

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





credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Higher Levels of Abstraction

Keras: High-Level Wrapper

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

(Also supports Theano backend)

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))

optimizer = SGD(lr=1e0)
model.compile(loss='mean_squared_error',
              optimizer=optimizer)

x = np.random.randn(N, D)
y = np.random.randn(N, D)
history = model.fit(x, y, nb_epoch=50,
                    batch_size=N, verbose=0)

                    batch_size=N, verbose=0)
```

formulas “by name”

=stochastic gradient descent

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

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)**

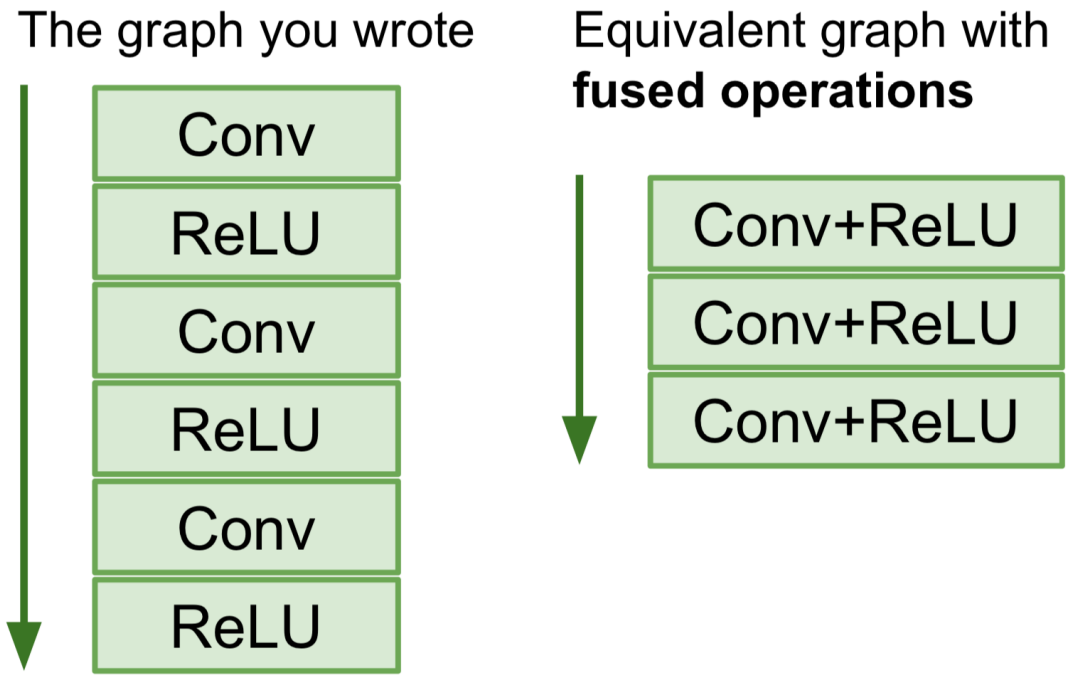
```
import torch
from torch.autograd import Variable

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

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

New graph each iteration

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

Static vs Dynamic: optimization

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



credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung, Song Han)

Static vs Dynamic: serialization

serialization = create a runnable program from the trained network

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

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung, Song Han)

Static vs Dynamic: conditionals, loops

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

PyTorch: Normal Python

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

x = Variable(torch.randn(N, D))
w1 = Variable(torch.randn(D, H))
w2 = Variable(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)
```

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

What to Use?

- **TensorFlow** is a safe bet for most projects. Not perfect but has huge community, wide usage. Maybe pair with high-level wrapper (Keras, Sonnet, etc)
- **PyTorch** is best for research. However still new, there can be rough patches.
- Use **TensorFlow** for one graph over many machines
  Consider **Caffe**, **Caffe2**, or **TensorFlow** for production deployment
- Consider **TensorFlow** or **Caffe2** for mobile

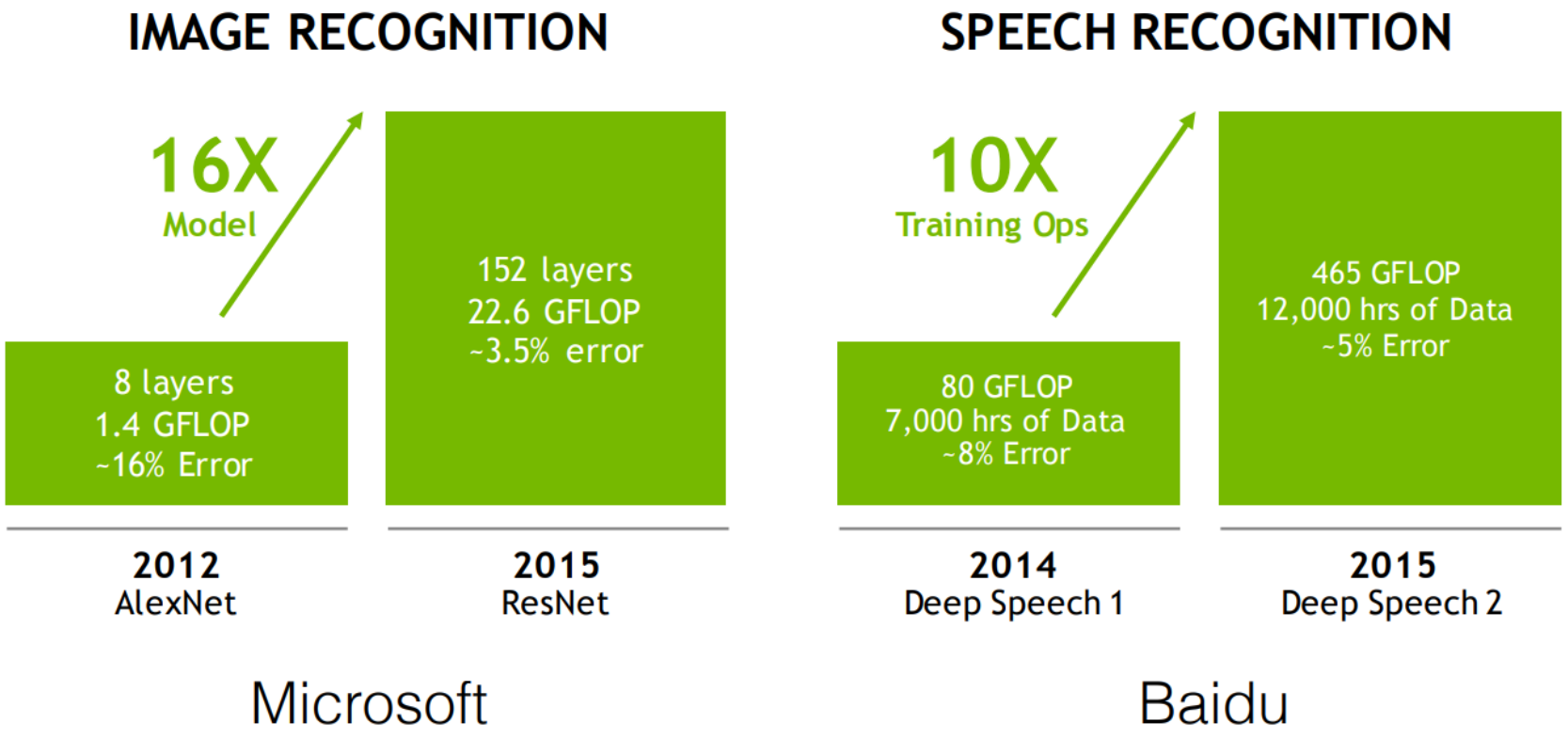
credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

CWI
Centrum Wiskunde & Informatica

DEEP LEARNING PERFORMANCE OPTIMIZATIONS

credits:
cs231n.stanford.edu, Song Han

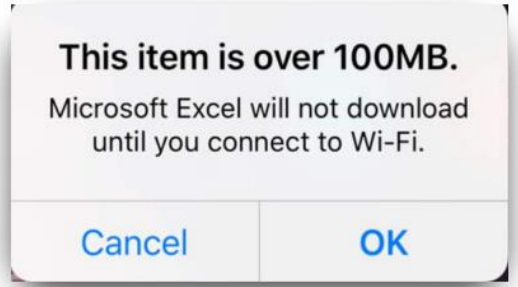
ML models are getting larger

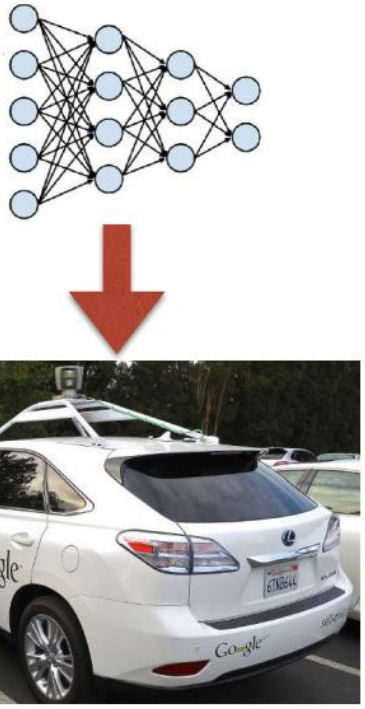


credits:
cs231n.stanford.edu, Song Han

First Challenge: Model Size

Hard to distribute large models through over-the-air update





credits:
cs231n.stanford.edu, Song Han

Second Challenge: Energy Efficiency

AlphaGo: 1920 CPUs and 280 GPUs,
$3000 electric bill per game

on mobile: drains battery
on data-center: increases TCO

credits:
cs231n.stanford.edu, Song Han

Third Challenge: Training Speed

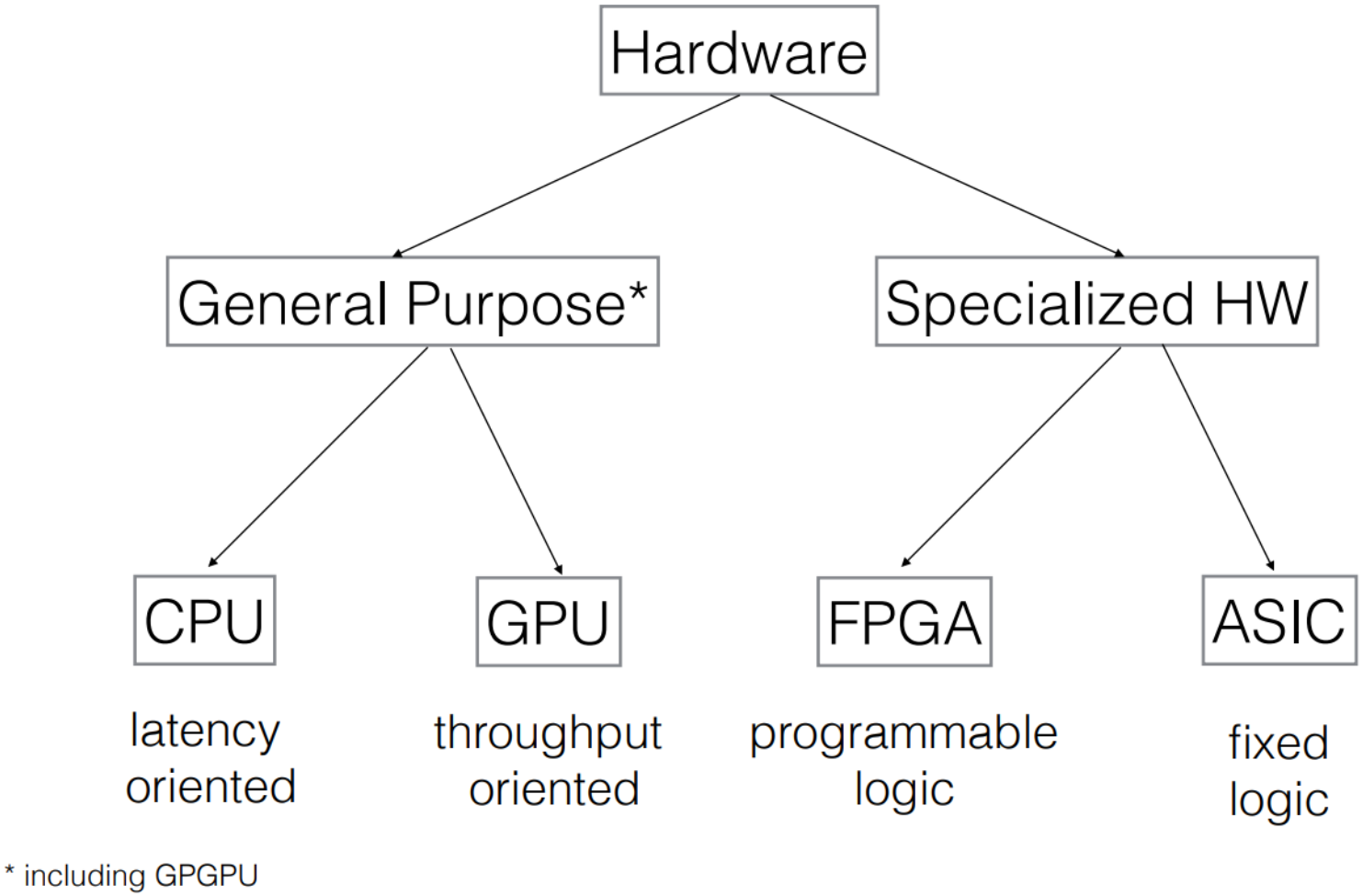
	Error rate	Training time
ResNet18:	10.76%	2.5 days
ResNet50:	7.02%	5 days
ResNet101:	6.21%	1 week
ResNet152:	6.16%	1.5 weeks

Such long training time limits ML researcher's productivity

Training time benchmarked with fb.resnet.torch using four M40 GPUs

credits:
cs231n.stanford.edu, Song Han

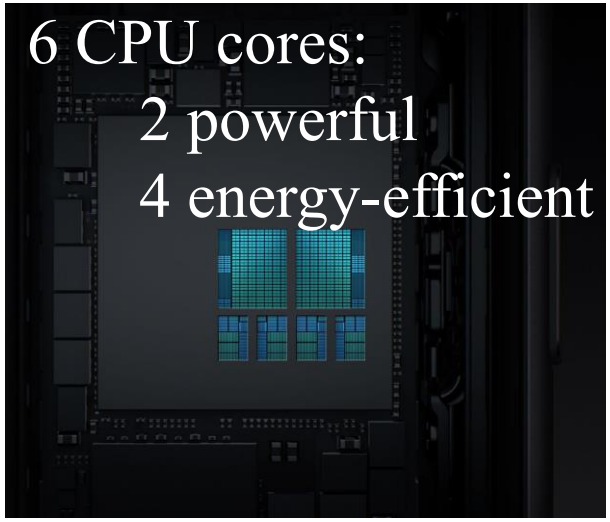
Hardware Basics

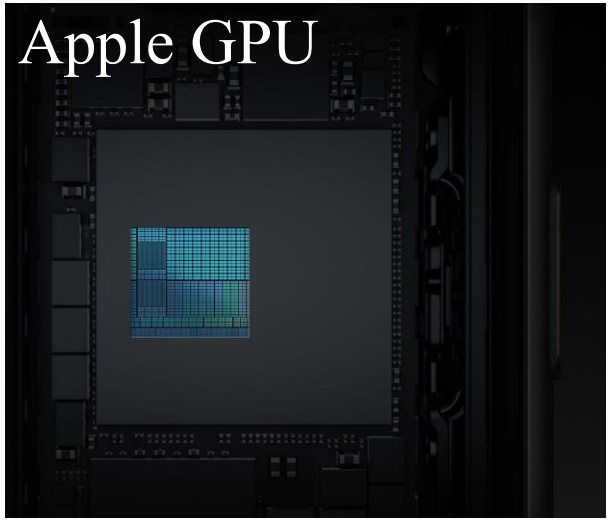


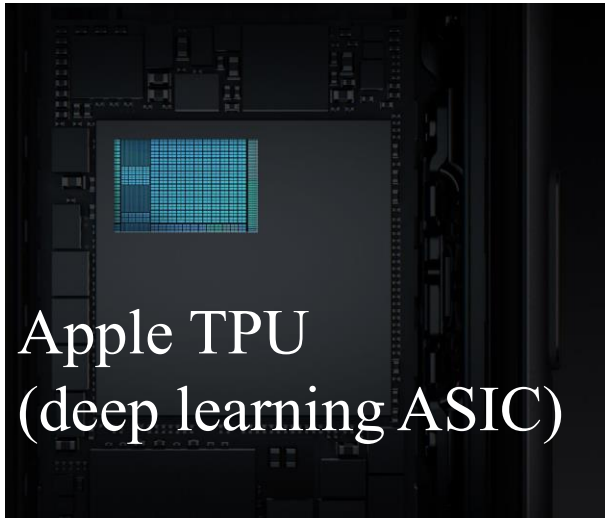
credits:
cs231n.stanford.edu, Song Han

Special hardware? It's in your pocket..

- iPhone 8 with A11 chip



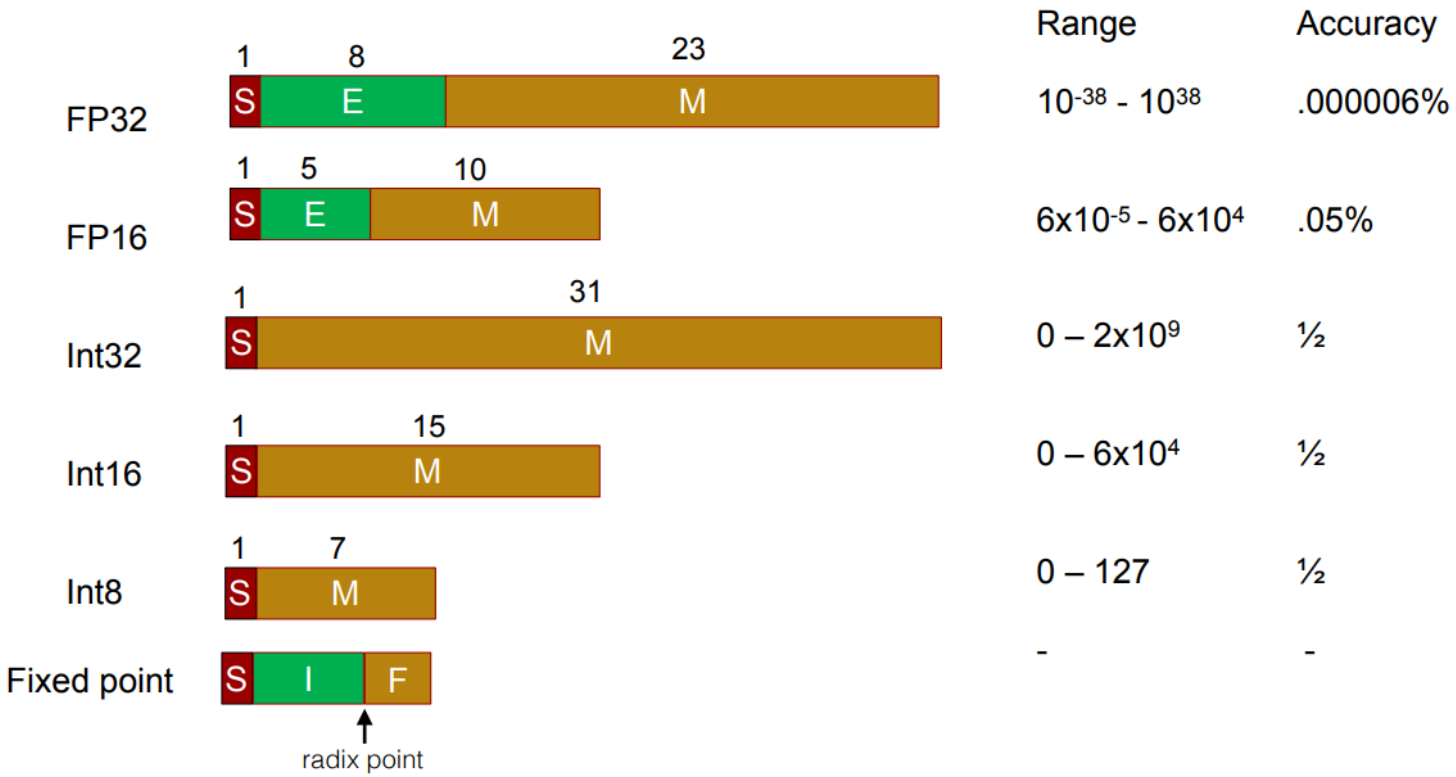




only on-chip FPGA missing (will come in time..)

Hardware Basics: Number Representation

$$(-1)^S \times (1.M) \times 2^E$$

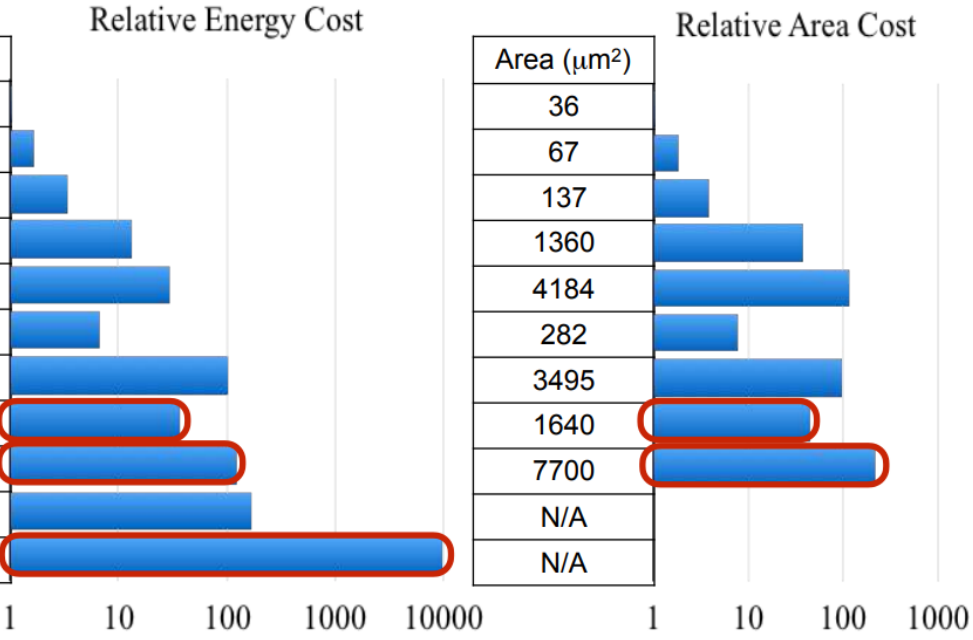


Dally, High Performance Hardware for Machine Learning, NIPS'2015

credits:
cs231n.stanford.edu, Song Han

Hardware Basics: Number Representation

Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A



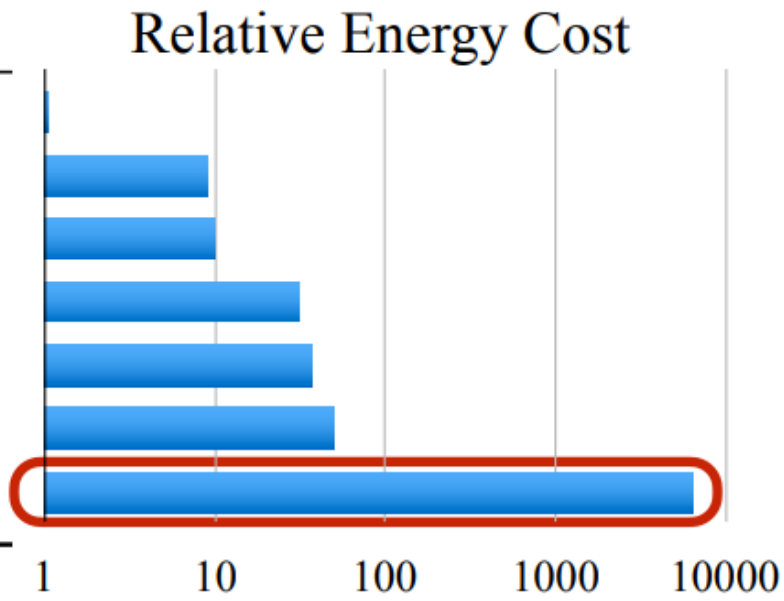
Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

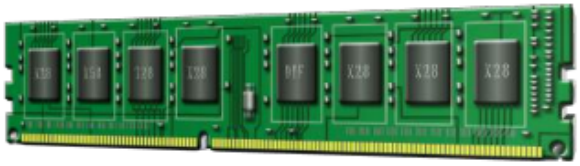
credits:
cs231n.stanford.edu, Song Han

Hardware Basics: Memory = Energy

larger model ➔ more memory references ➔ more energy consumed

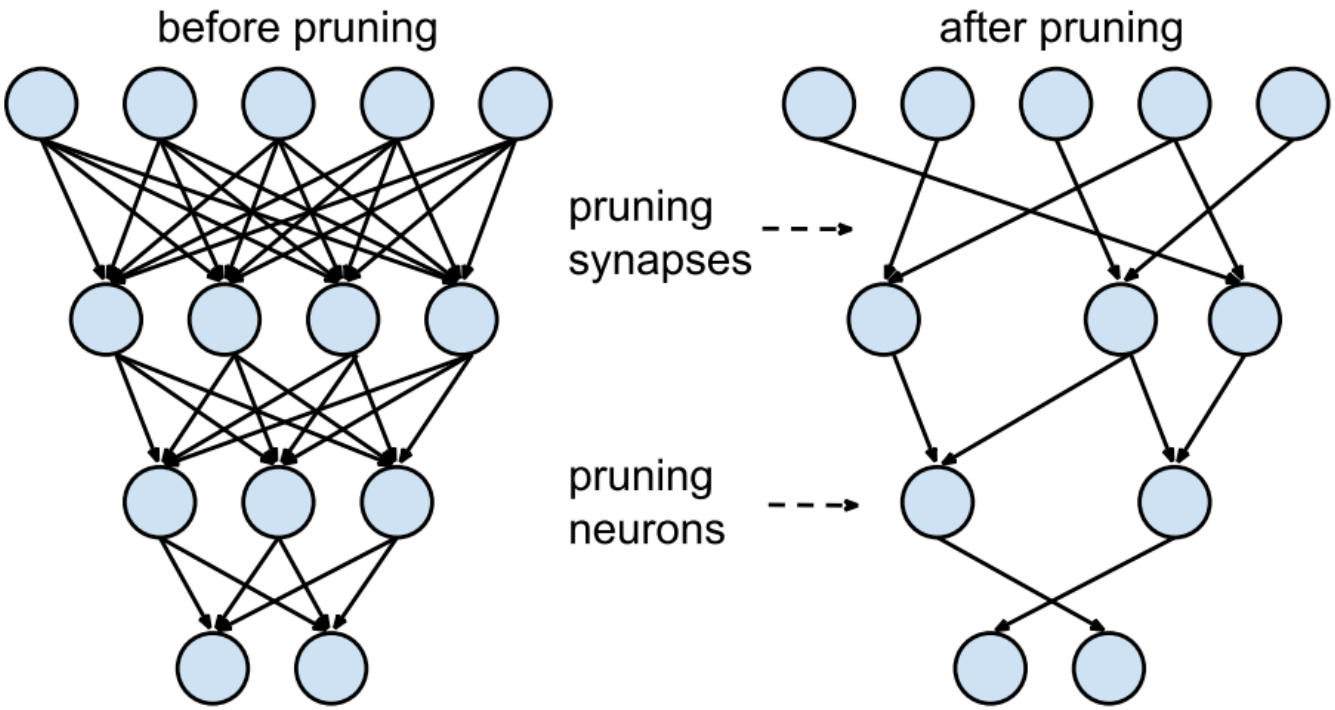
Operation	Energy [pJ]
32 bit int ADD	0.1
32 bit float ADD	0.9
32 bit Register File	1
32 bit int MULT	3.1
32 bit float MULT	3.7
32 bit SRAM Cache	5
32 bit DRAM Memory	**640**



1  = 1000 

credits:
cs231n.stanford.edu, Song Han

Pruning Neural Networks

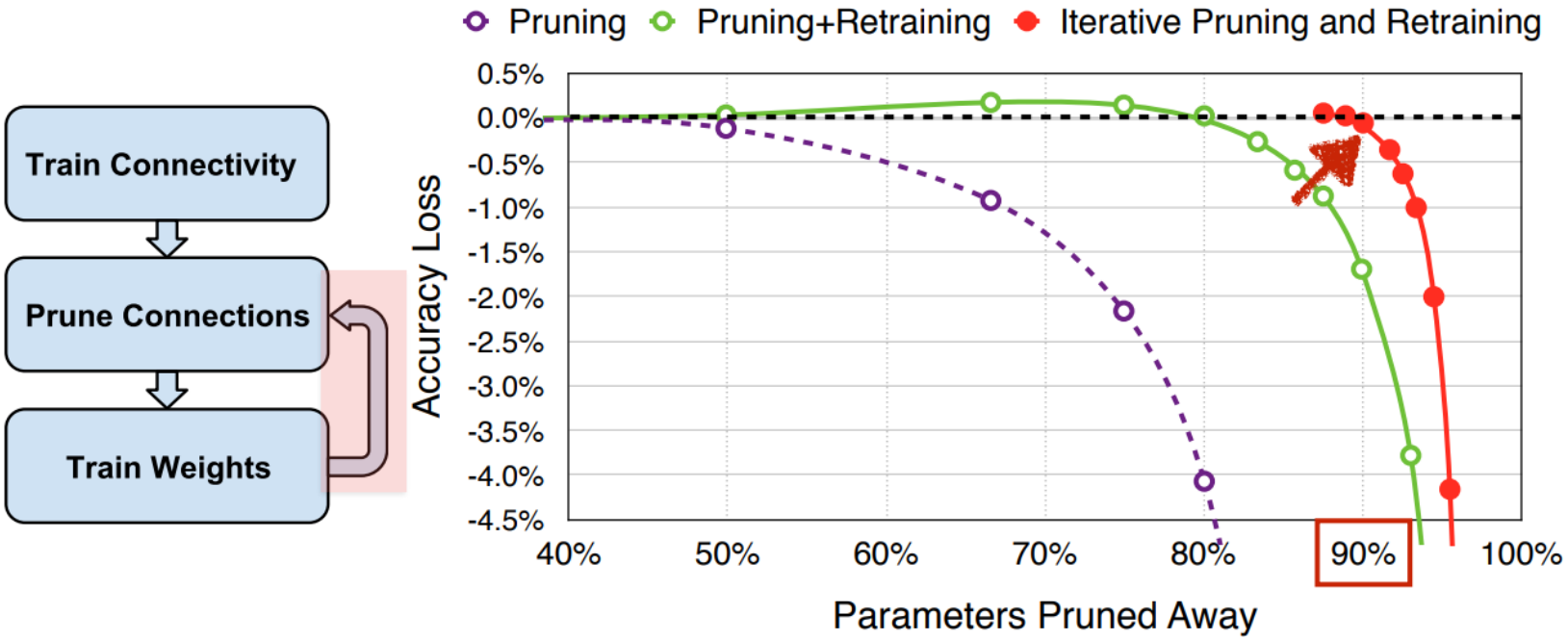


[Lecun et al. NIPS'89]

[Han et al. NIPS'15]

credits:
cs231n.stanford.edu, Song Han

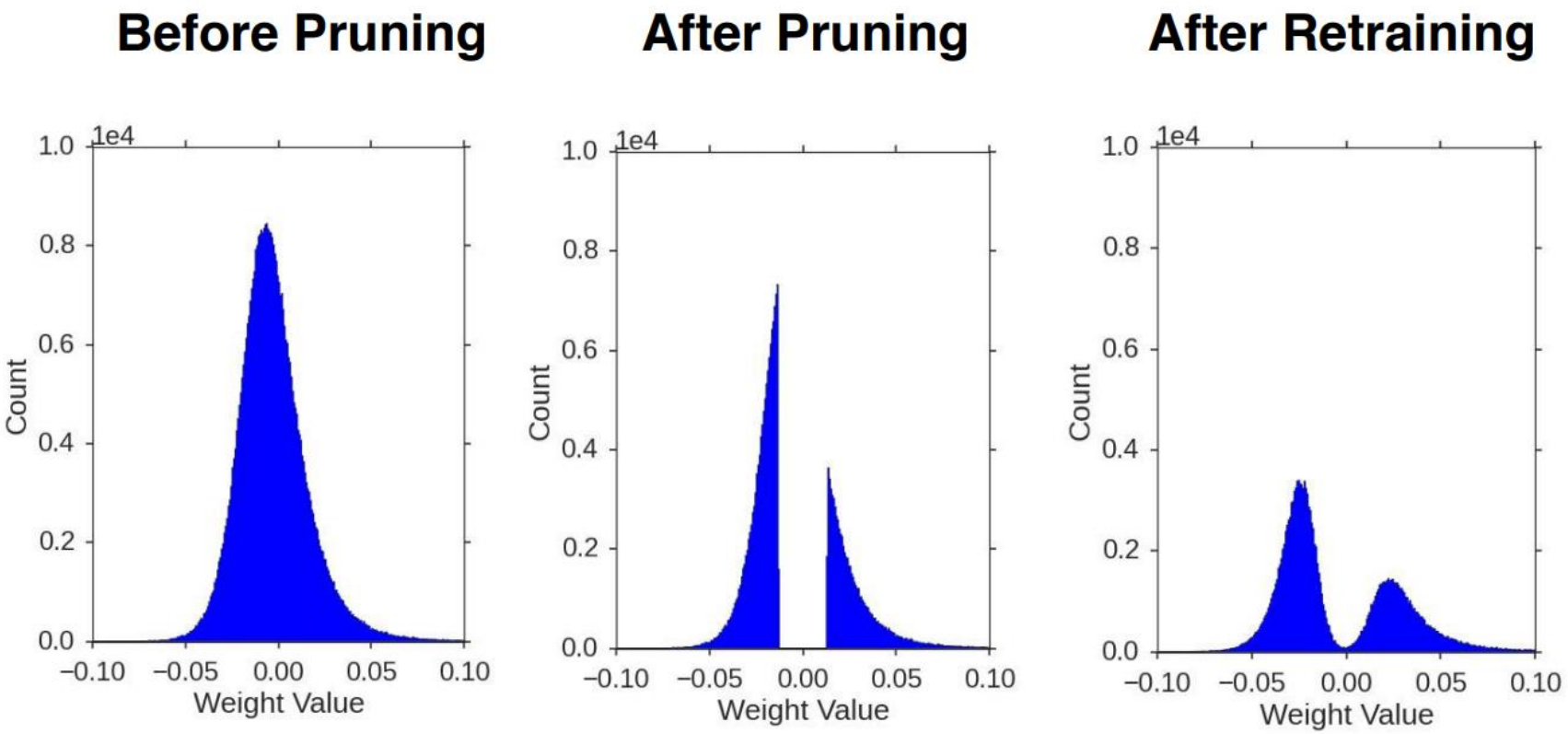
Pruning Neural Networks



- Learning both Weights and Connections for Efficient Neural Networks, Han, Pool, Tran Dally, NIPS2015

credits:
cs231n.stanford.edu, Song Han

Pruning Changes the Weight Distribution

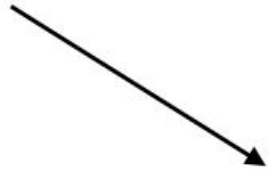


Conv5 layer of Alexnet. Representative for other network layers as well.

credits:
cs231n.stanford.edu, Song Han

Pruning Happens in the Human Brain

1000 Trillion Synapses



500 Trillion Synapses

This image is in the public domain

This image is in the public domain

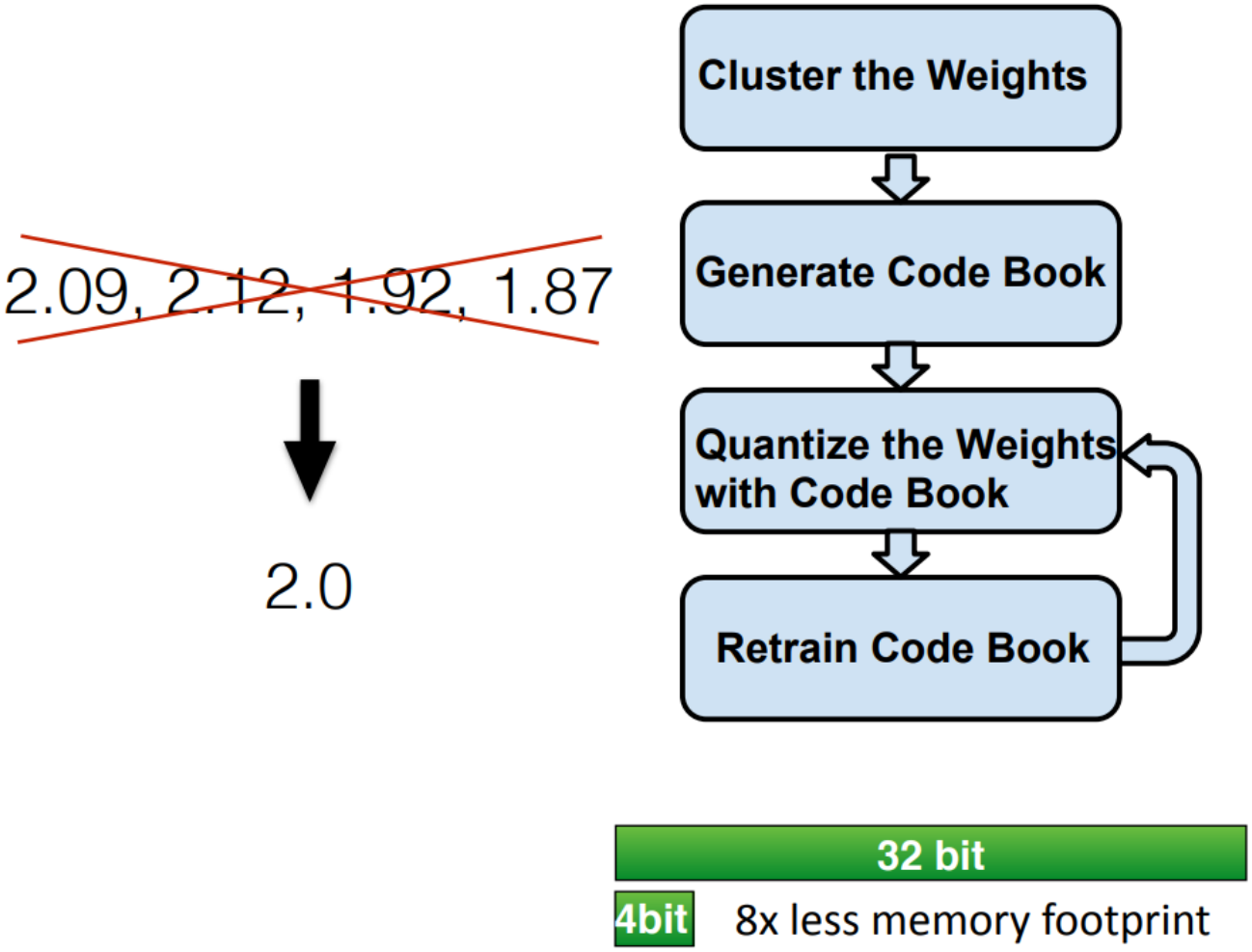
1 year old

Adolescent

Christopher A Walsh. Peter Huttenlocher (1931-2013). Nature, 502(7470):172–172, 2013.

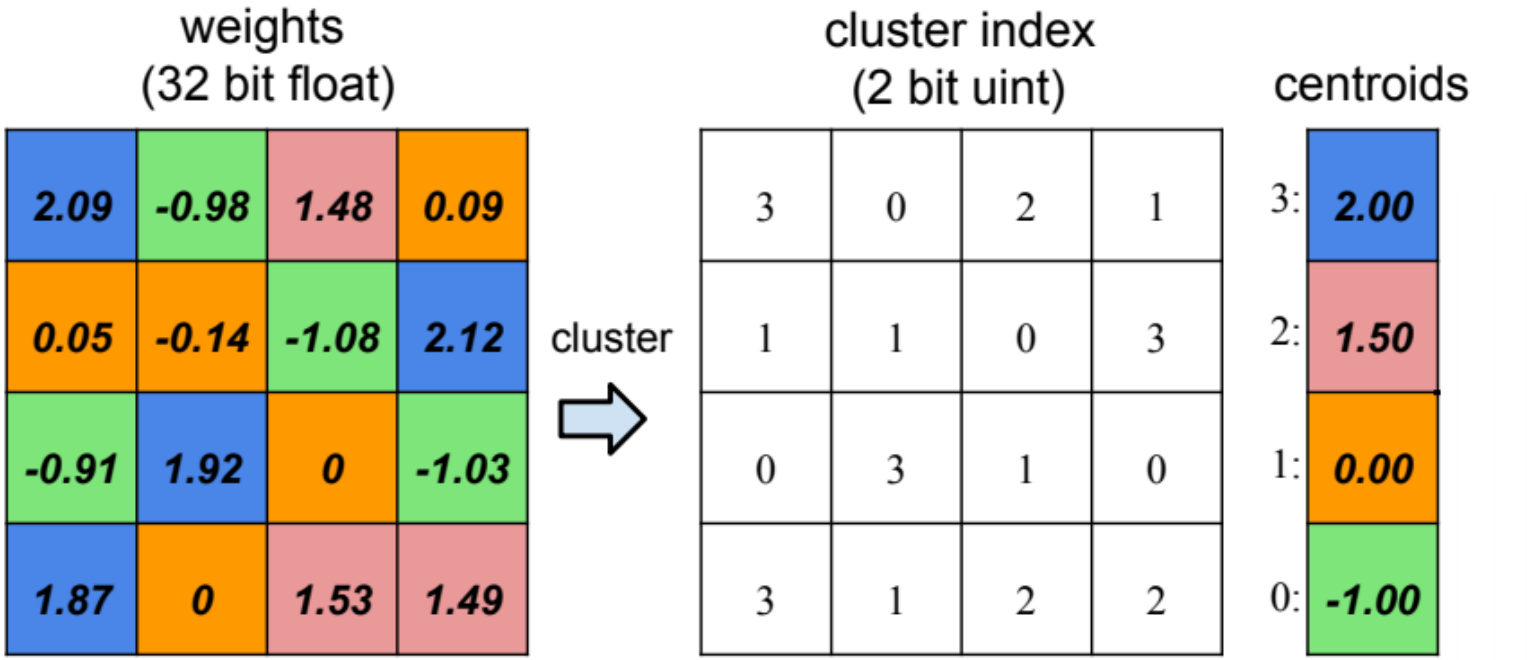
credits:
cs231n.stanford.edu, Song Han

Trained Quantization



credits:
cs231n.stanford.edu, Song Han

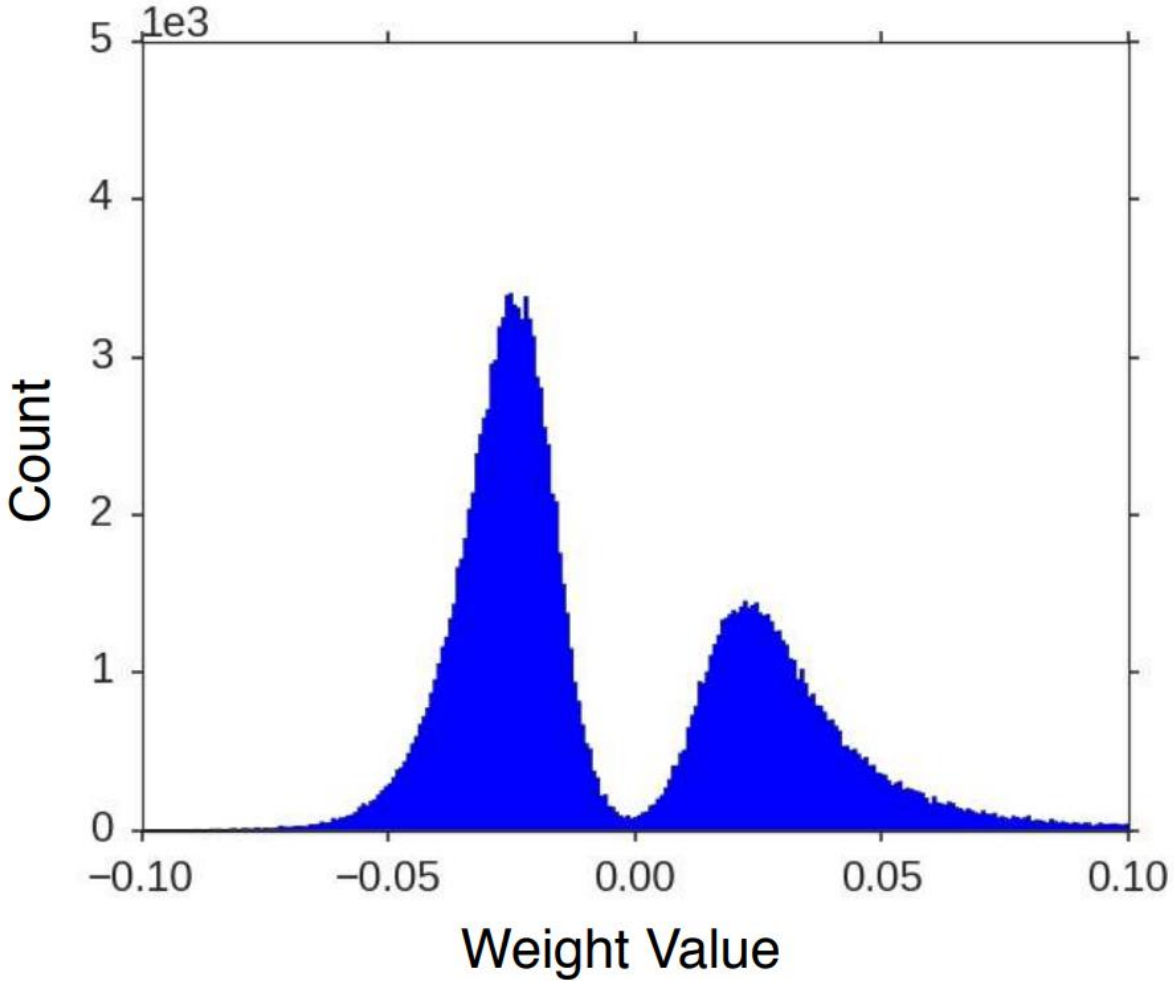
Trained Quantization



credits:
cs231n.stanford.edu, Song Han

Trained Quantization: Before

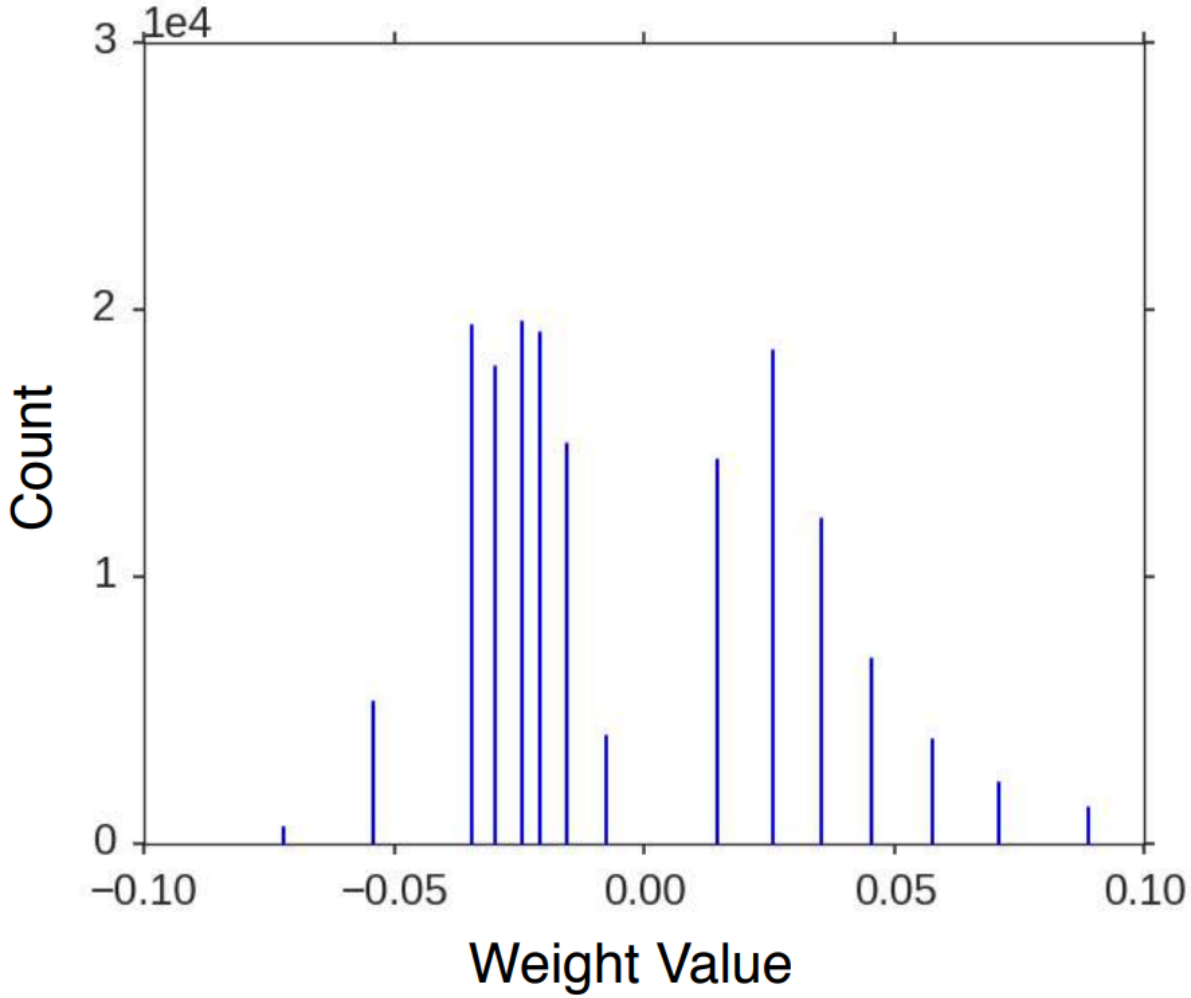
- Continuous weight distribution



credits:
cs231n.stanford.edu, Song Han

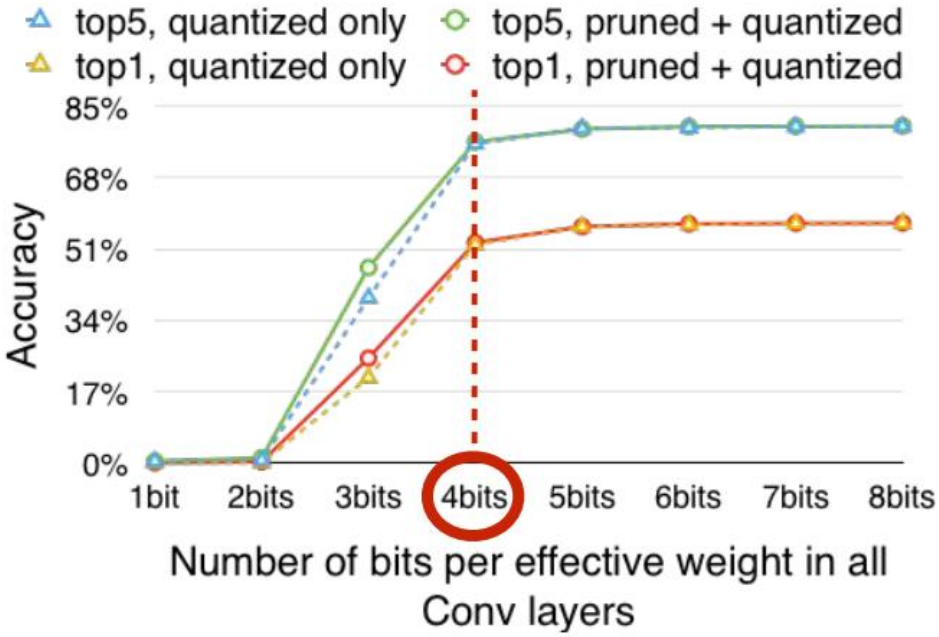
Trained Quantization: After

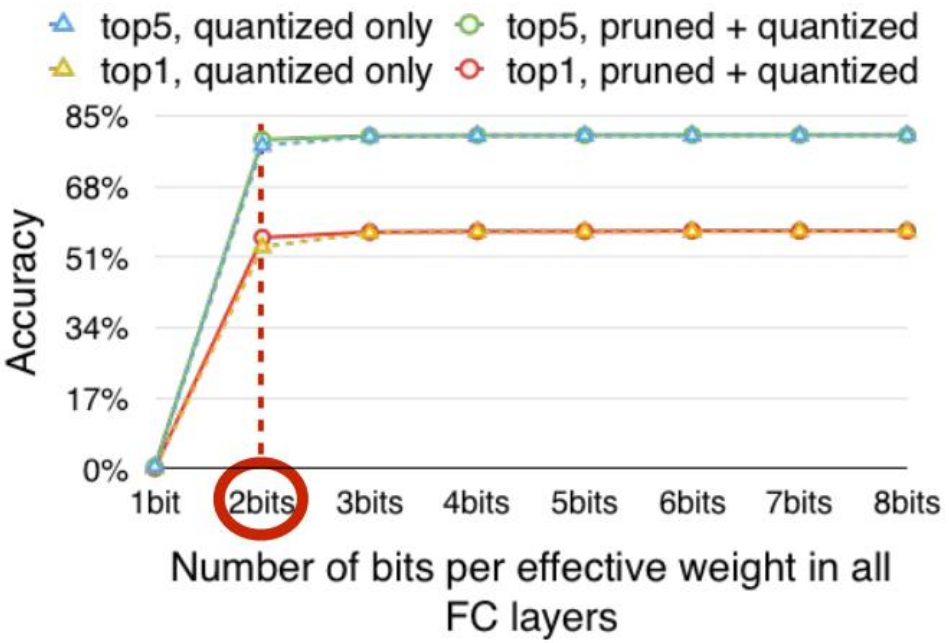
- Discrete weight distribution



credits:
cs231n.stanford.edu, Song Han

Trained Quantization: How Many Bits?

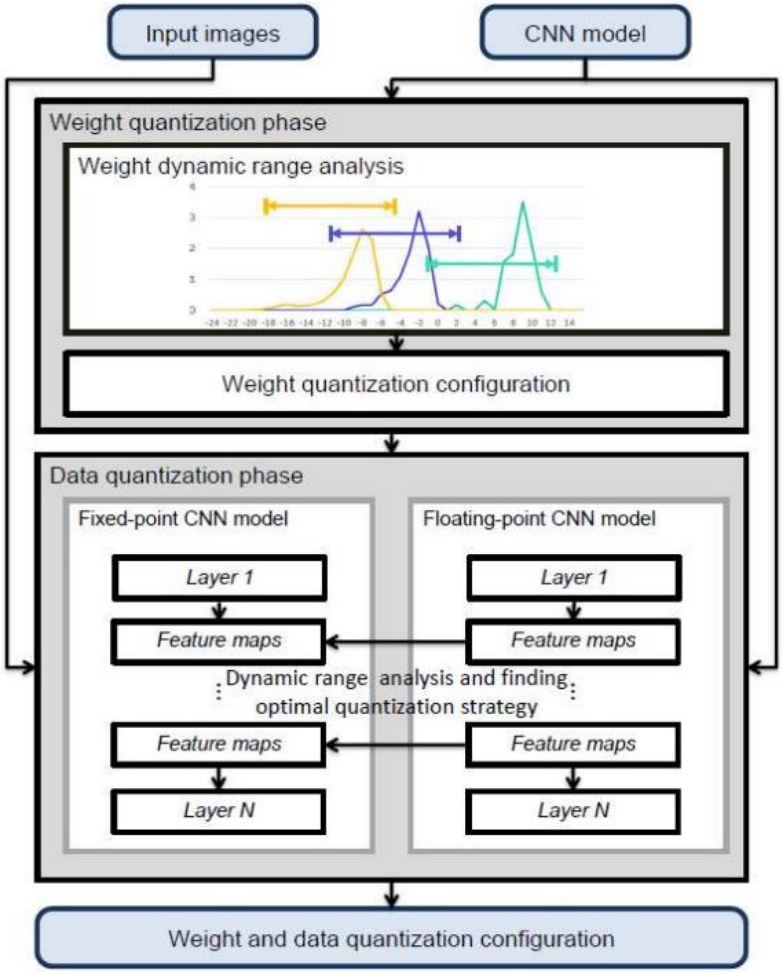




- Deep Compression: compressing deep neural networks with pruning, trained quantization and Huffman coding, Han, Moa, Dally, ICLR2016

credits:
cs231n.stanford.edu, Song Han

Quantization to Fixed Point Decimals (=Ints)

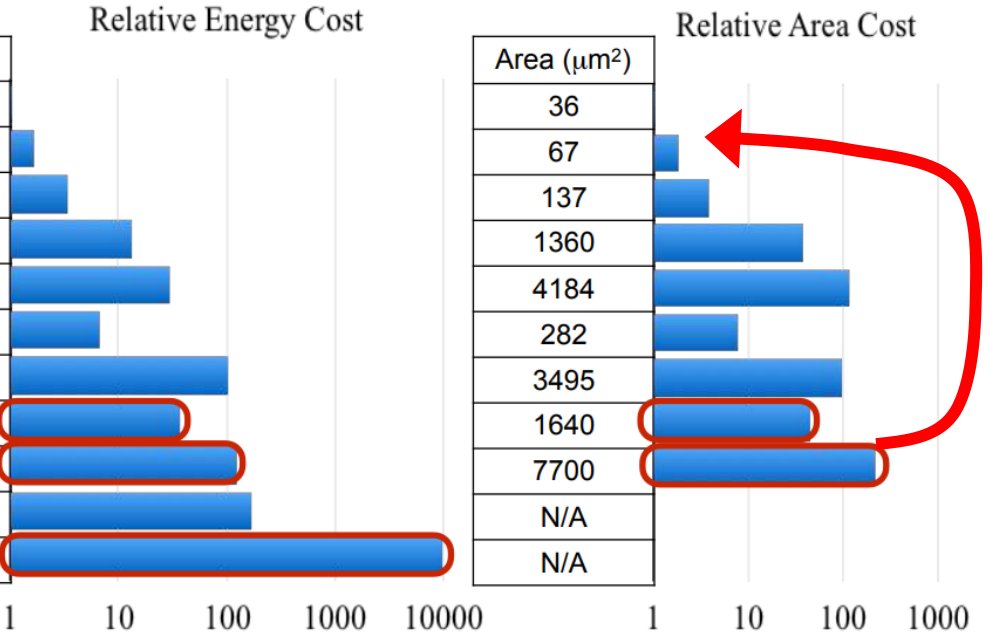


- Train with float
- Quantizing the weight and activation:
  - Gather the statistics for weight and activation
  - Choose proper radix point position
- Fine-tune in float format
- Convert to fixed-point format

credits:
cs231n.stanford.edu, Song Han

Hardware Basics: Number Representation

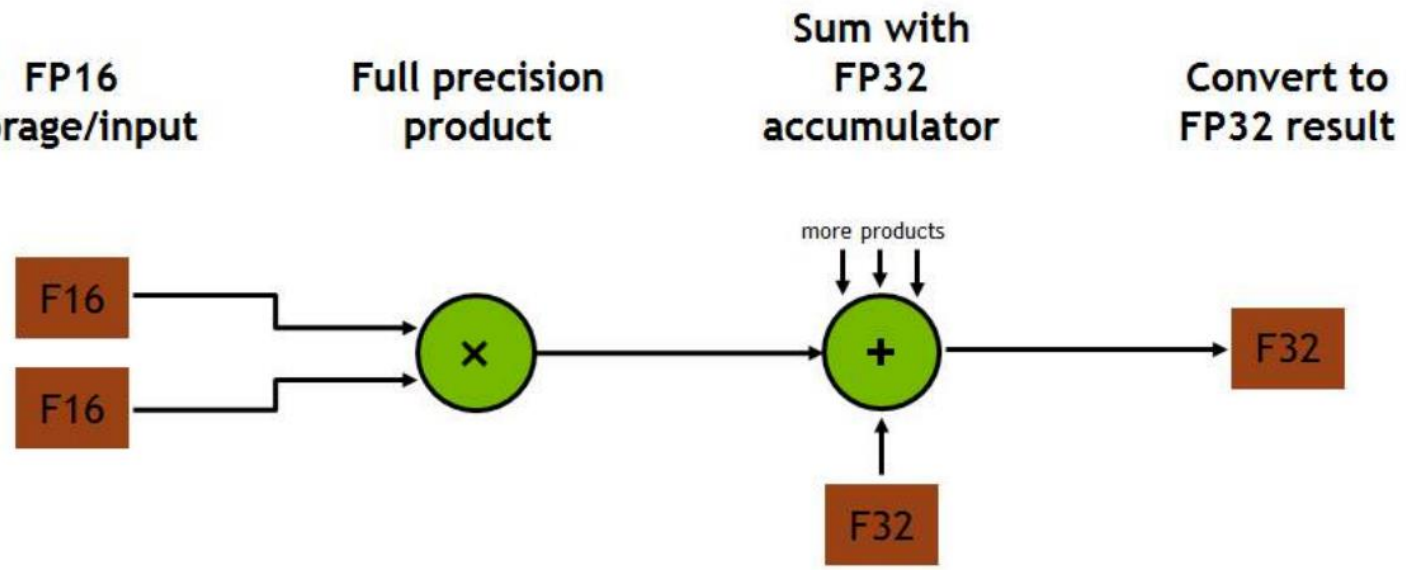
Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A



Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

credits:
cs231n.stanford.edu, Song Han

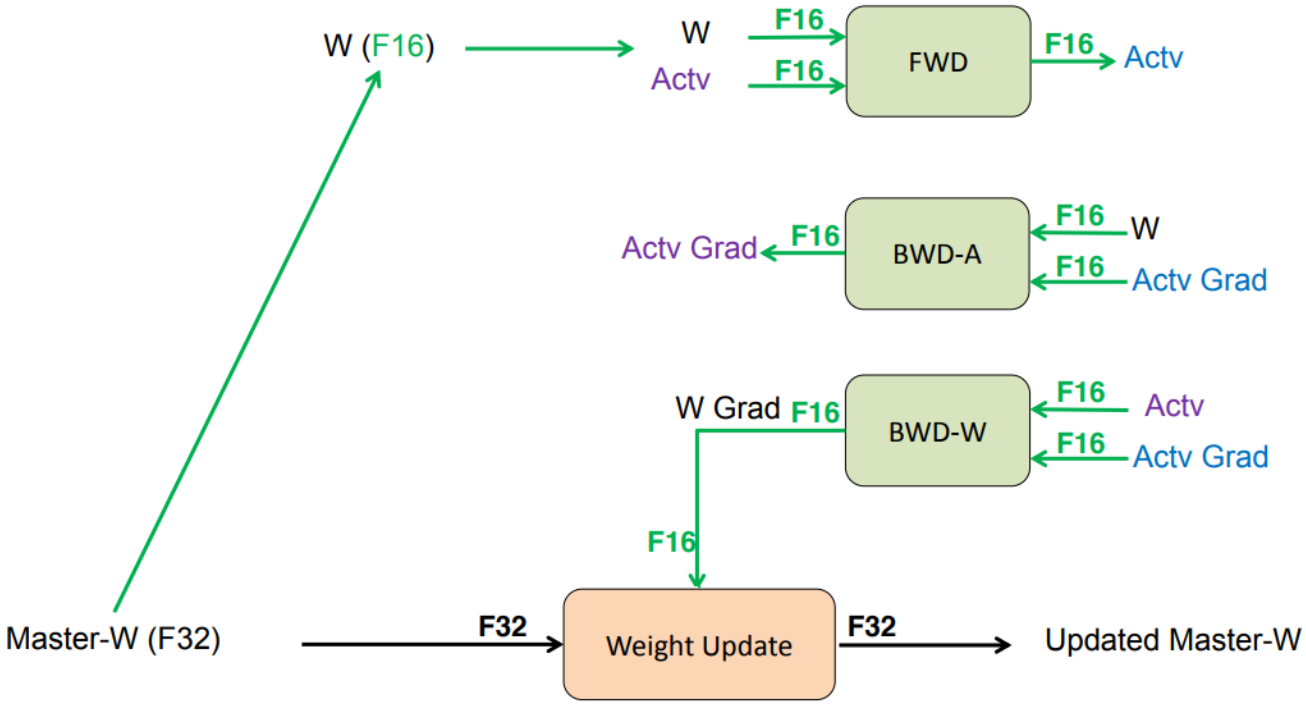
Mixed Precision Training



https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/

credits:
cs231n.stanford.edu, Song Han

Mixed Precision Training



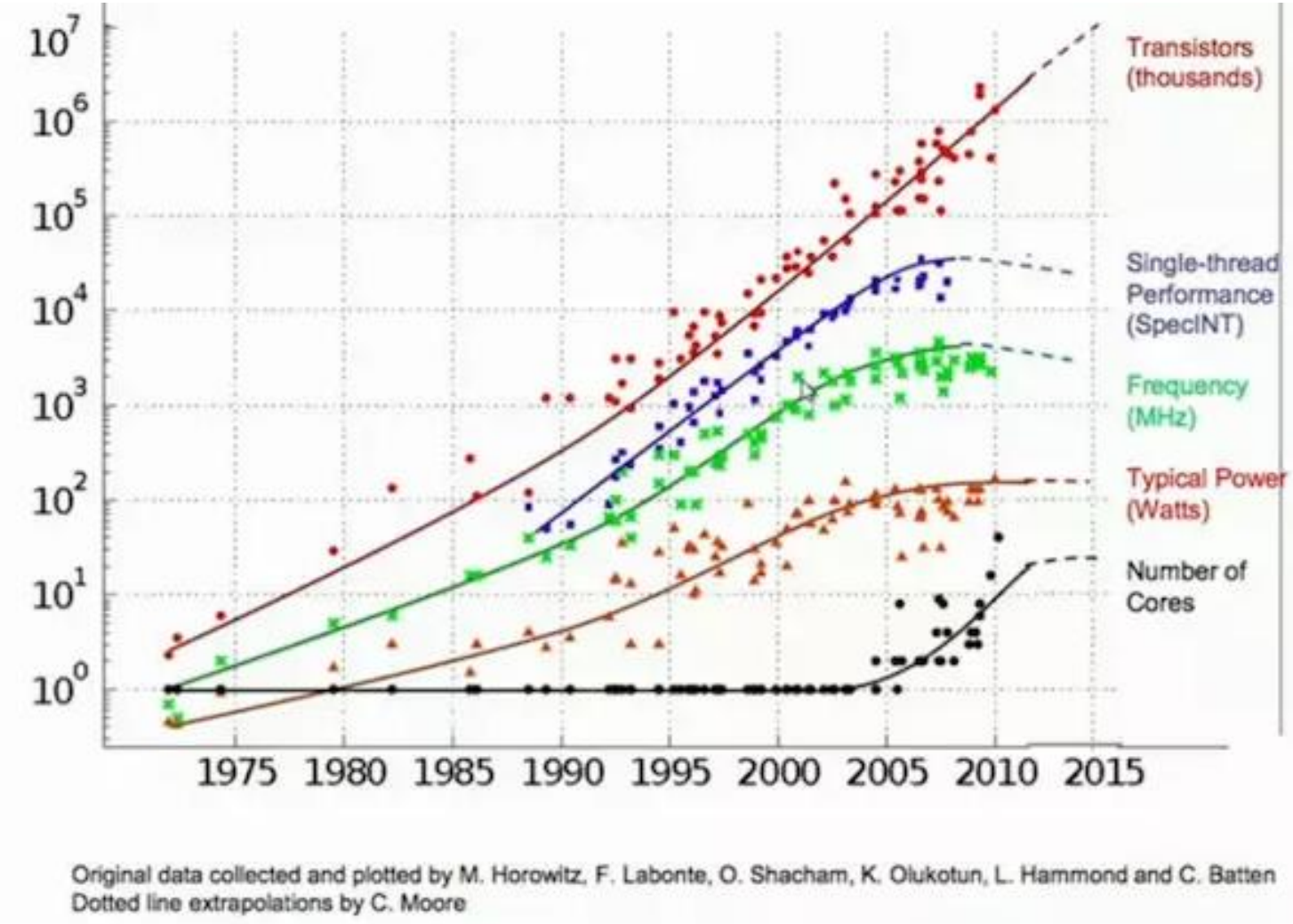
Boris Ginsburg, Sergei Nikolaev, Paulius Micikevicius, “Training with mixed precision”, NVIDIA GTC 2017

credits:
cs231n.stanford.edu, Song Han

DEEP LEARNING HARDWARE

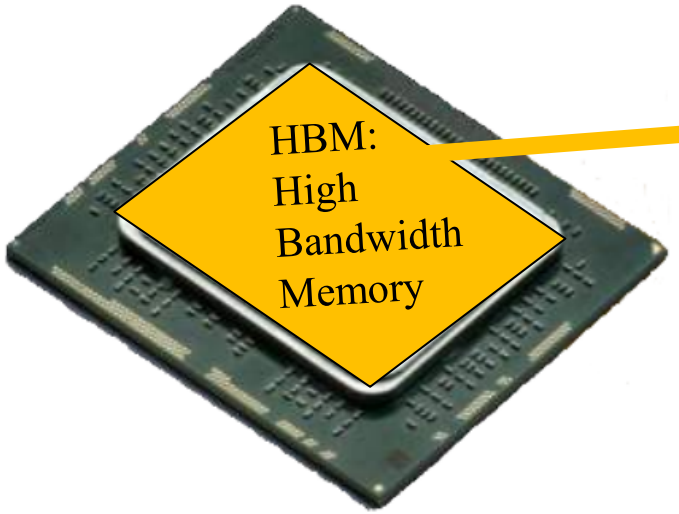
credits:
cs231n.stanford.edu, Song Han

The end of CPU scaling



CPUs for Training - SIMD to the rescue?

Intel Knights Landing (2016)



- 7 TFLOPS FP32
- 16GB MCDRAM– 400 GB/s
- 245W TDP
- 29 GFLOPS/W (FP32)
- 14nm process

Knights Mill: next gen Xeon Phi "optimized for deep learning"

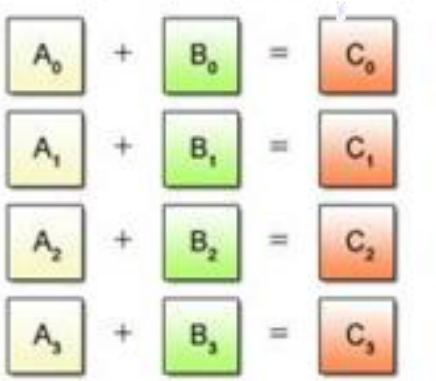
Intel announced the addition of new vector instructions for deep learning (AVX512-4VNNIW and AVX512-4FMAPS), October 2016

Slide Source: Sze et al Survey of DNN Hardware, MICRO'16 Tutorial.
Image Source: Intel, Data Source: Next Platform

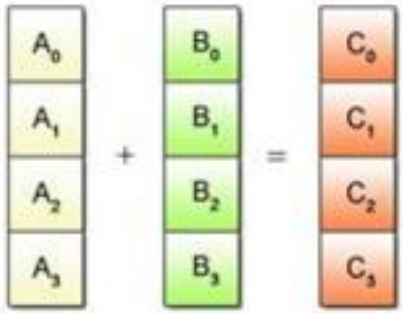
credits:
cs231n.stanford.edu, Song Han

CPUs for Training - SIMD to the rescue?

4 scalar instructions



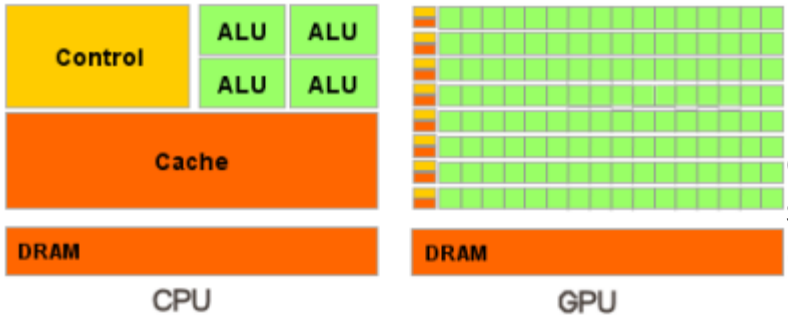
1 SIMD instruction



"ALU": arithmetic logic unit (implements +, *, - etc. instructions)

CPU vs GPU

CPU: A lot of chip surface for cache memory and control



GPU: almost all chip surface for ALUs (compute power)

	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	$399

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

GPU cards have their own memory chips: smaller but nearby and faster than system memory

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

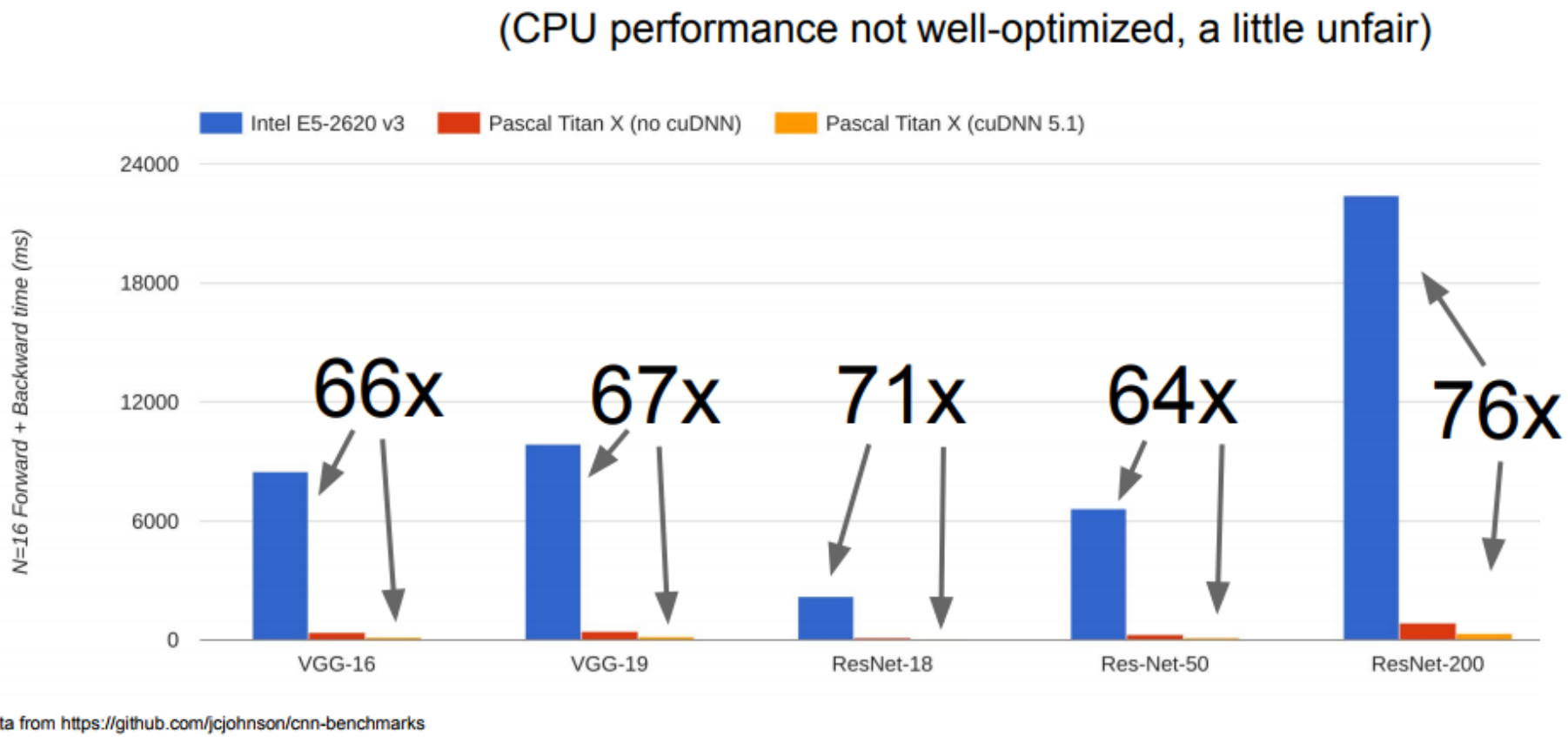
Programming GPUs

- CUDA (NVIDIA only)
  - Write C-like code that runs directly on the GPU
  - Higher-level APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
  - Similar to CUDA, but runs on anything
  - Usually slower :(

All major deep learning libraries (TensorFlow, PyTorch, MXNET, etc) support training and model evaluation on GPUs.

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

CPU vs GPU: performance



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

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

CPU - GPU: communication

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

credits:
cs231n.stanford.edu; Fei-Fei Li, Justin Johnson, Serena Yeung

GPUs for Training

Nvidia PASCAL GP100 (2016)

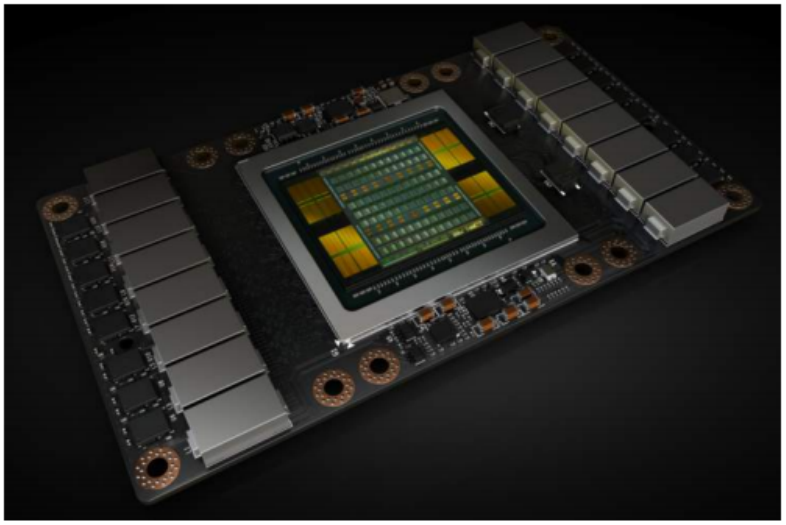
- 10/20 TFLOPS FP32/FP16
- 16GB HBM – 750 GB/s
- 300W TDP
- 67 GFLOPS/W (FP16)
- 16nm process
- 160GB/s NV Link

Slide Source: Sze et al Survey of DNN Hardware, MICRO'16 Tutorial.
Data Source: NVIDIA

credits:
cs231n.stanford.edu, Song Han

GPUs for Training

Nvidia Volta GV100 (2017)

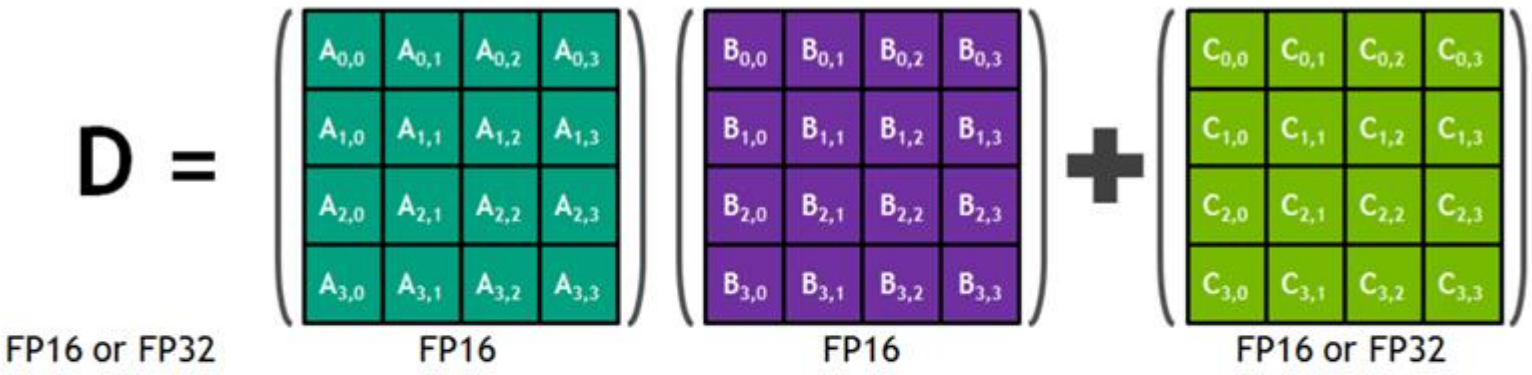


- 15 FP32 TFLOPS
- 120 Tensor TFLOPS
- 16GB HBM2 @ 900GB/s
- 300W TDP
- 12nm process
- 21B Transistors
- die size: 815 mm2
- 300GB/s NVLink

Data Source: NVIDIA

credits:
cs231n.stanford.edu, Song Han

New in Volta: Tensor Core



a new instruction that performs 4x4x4 FMA mixed-precision operations per clock
12X increase in throughput for the Volta V100 compared to the Pascal P100

https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/

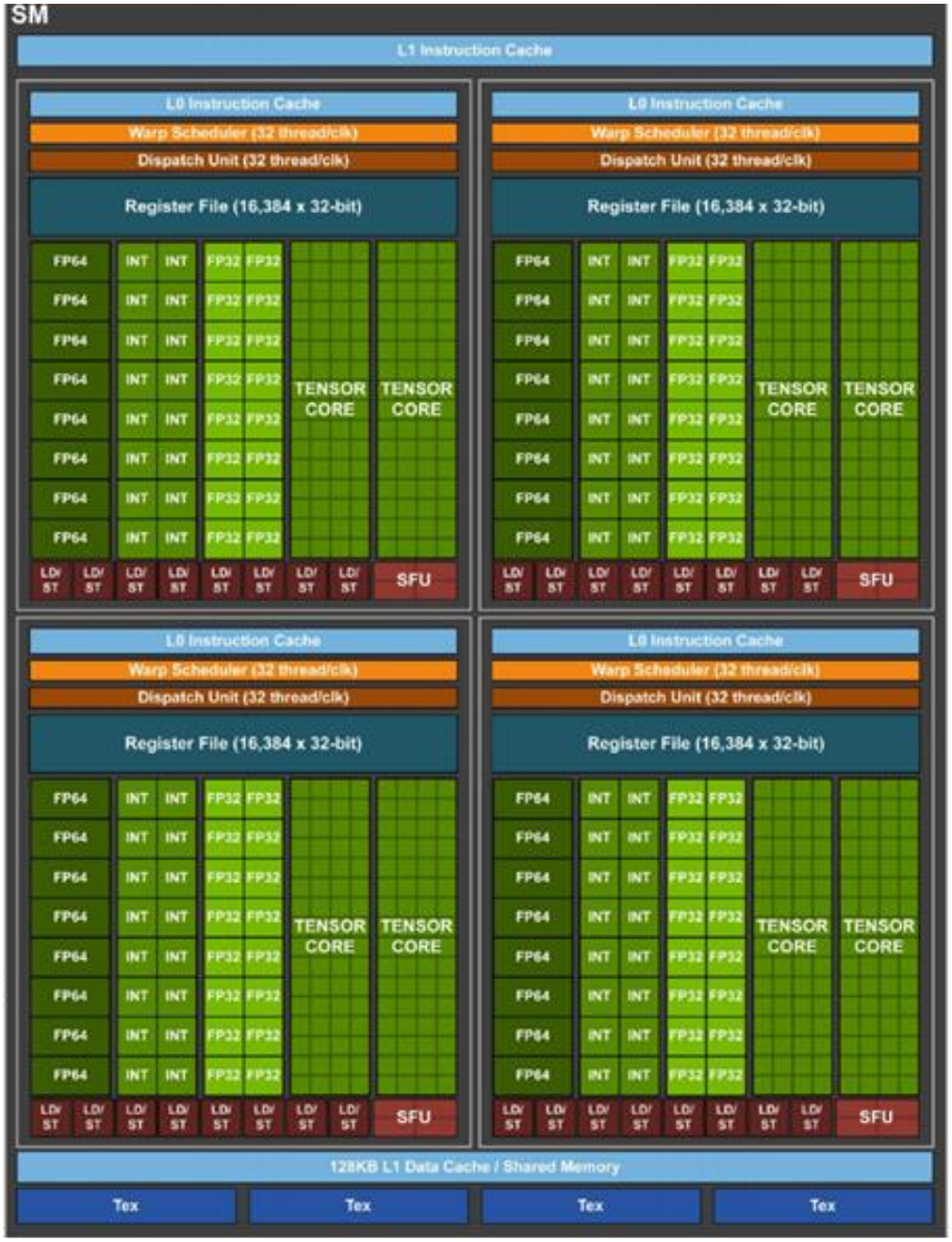
credits:
cs231n.stanford.edu, Song Han

Volta Chip Area

The GV100 SM is partitioned into four processing blocks, each with:

- 8 FP64 Cores
- 16 FP32 Cores
- 16 INT32 Cores
- two of the new mixed-precision Tensor Cores for deep learning
- a new L0 instruction cache
- one warp scheduler
- one dispatch unit
- a 64 KB Register File.

https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/



credits:
cs231n.stanford.edu, Song Han

GPU evolution

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1455 MHz
Peak FP32 TFLOP/s*	5.04	6.8	10.6	15
Peak Tensor Core TFLOP/s*	-	-	-	120
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

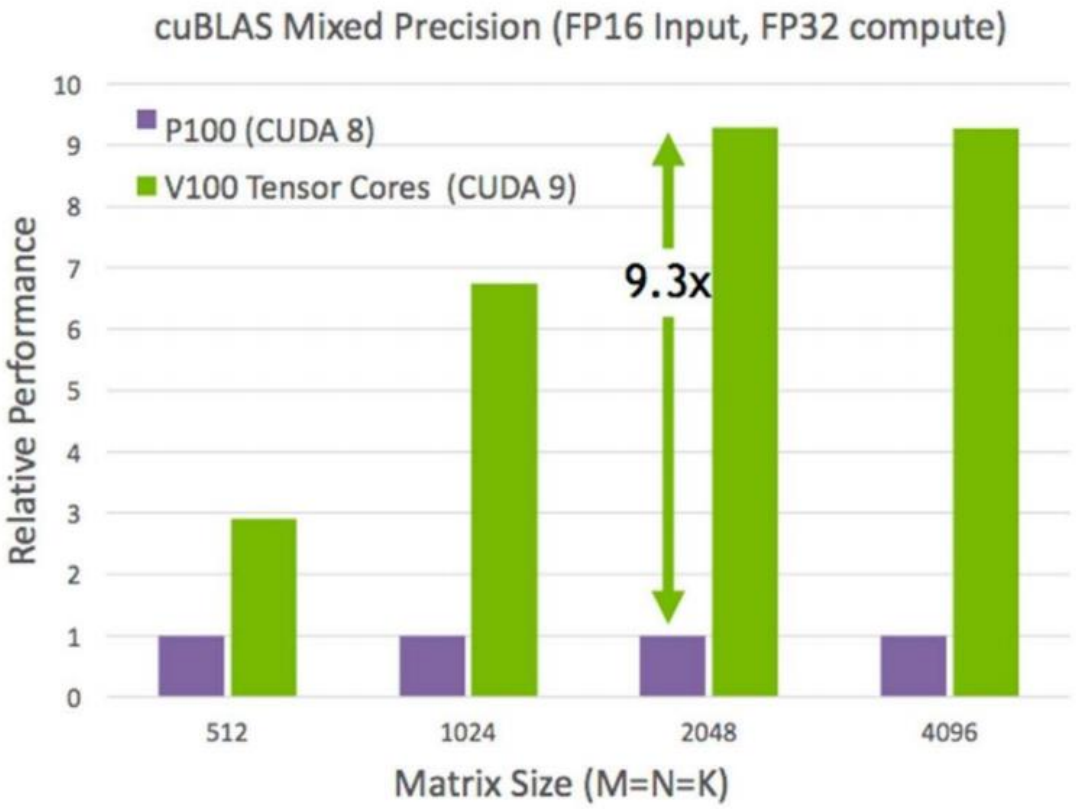
big jump in ML speed

fast memory, sits on top of the GPU chip

https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/

credits:
cs231n.stanford.edu, Song Han

Pascal vs Volta

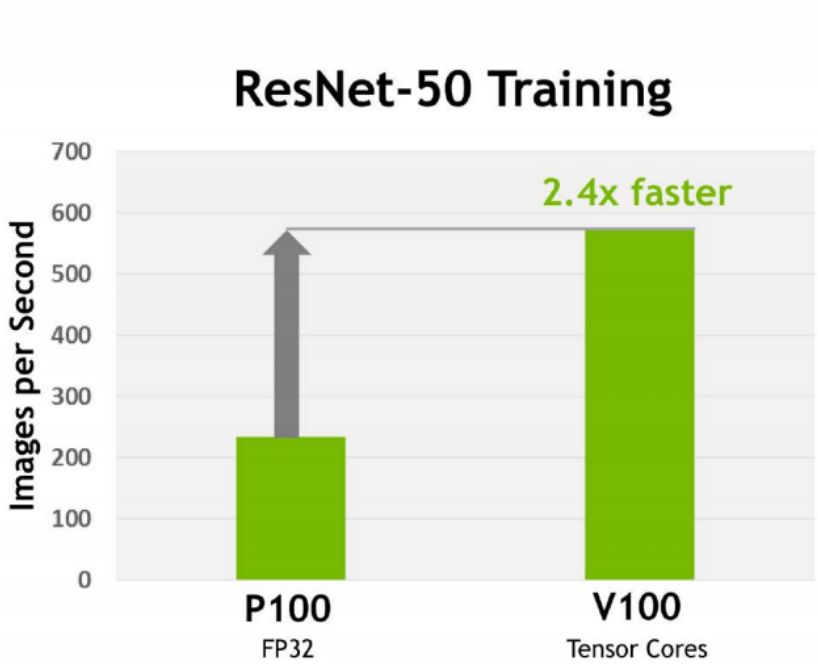


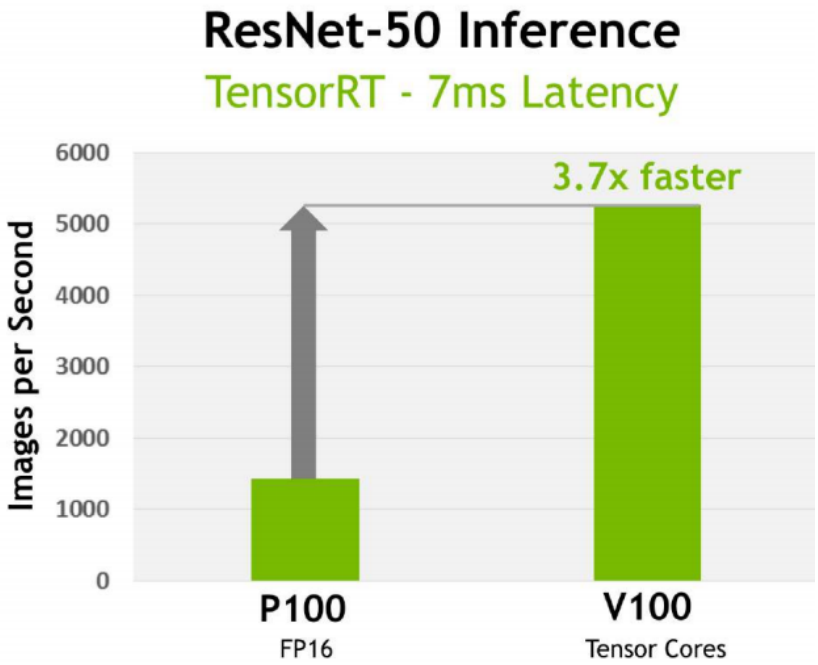
Tesla V100 Tensor Cores and CUDA 9 deliver up to 9x higher performance for GEMM operations.

https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/

credits:
cs231n.stanford.edu, Song Han

Pascal vs Volta



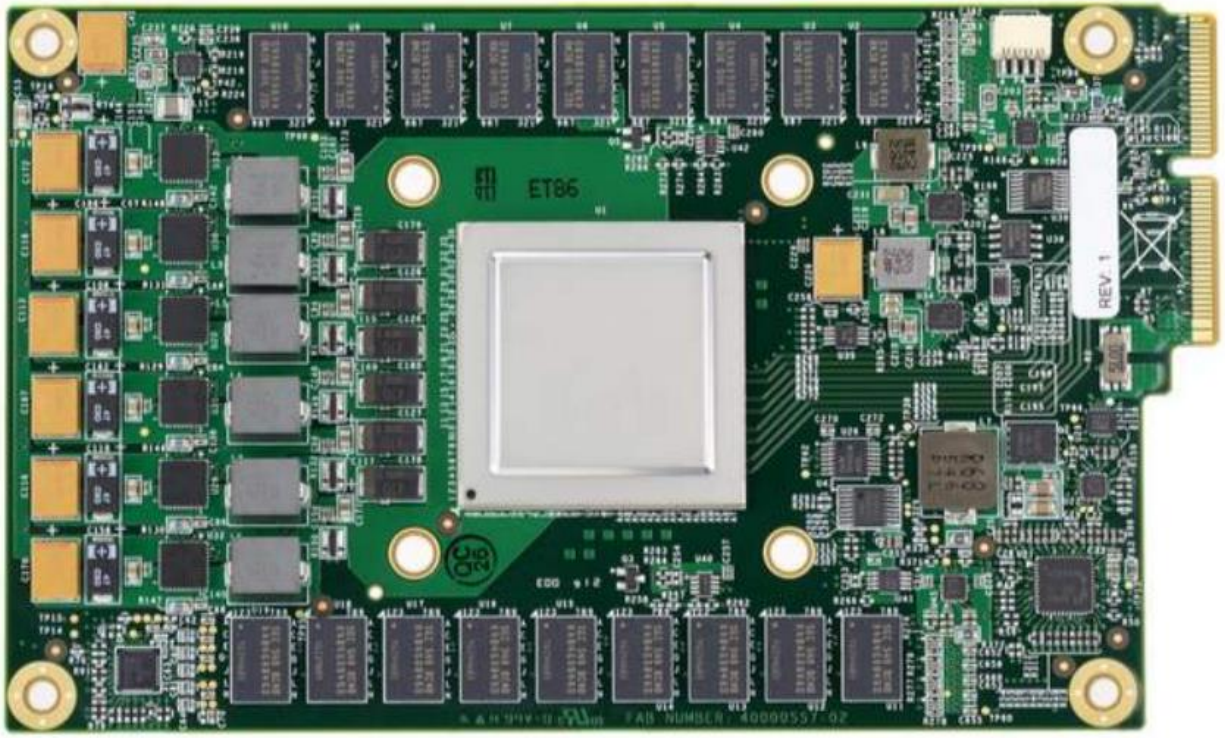


Left: Tesla V100 trains the ResNet-50 deep neural network 2.4x faster than Tesla P100.
Right: Given a target latency per image of 7ms, Tesla V100 is able to perform inference using the ResNet-50 deep neural network 3.7x faster than Tesla P100.

https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/

credits:
cs231n.stanford.edu, Song Han

TensorFlow Processing Unit (TPU) 2015



TPU Card to replace a disk

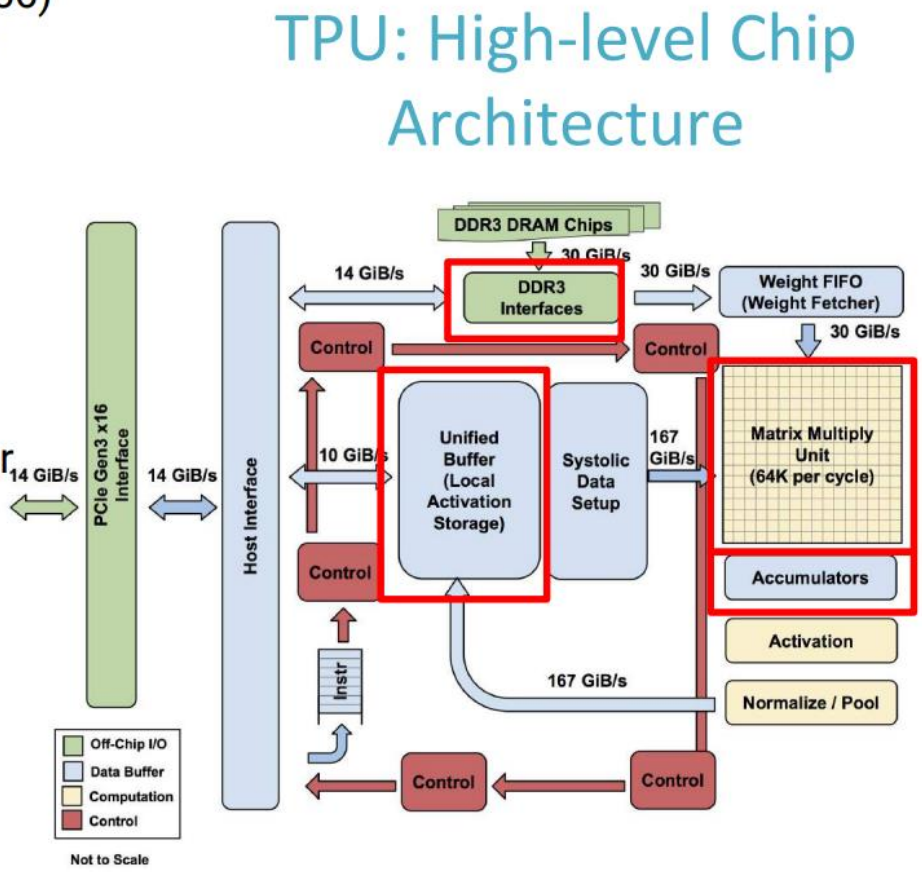
Up to 4 cards / server

David Patterson and the Google TPU Team, In-Data Center Performance Analysis of a Tensor Processing Unit

credits:
cs231n.stanford.edu, Song Han

TPU Architecture

- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
- 700 MHz clock rate
- Peak: 92T operations/second
  - 65,536 * 2 * 700M
- >25X as many MACs vs GPU
- >100X as many MACs vs CPU
- 4 MiB of on-chip Accumulator memory
- 24 MiB of on-chip Unified Buffer (activation memory)
- 3.5X as much on-chip memory vs GPU
- Two 2133MHz DDR3 DRAM channels
- 8 GiB of off-chip weight DRAM memory



David Patterson and the Google TPU Team, In-Data Center Performance Analysis of a Tensor Processing Unit

credits:
cs231n.stanford.edu, Song Han

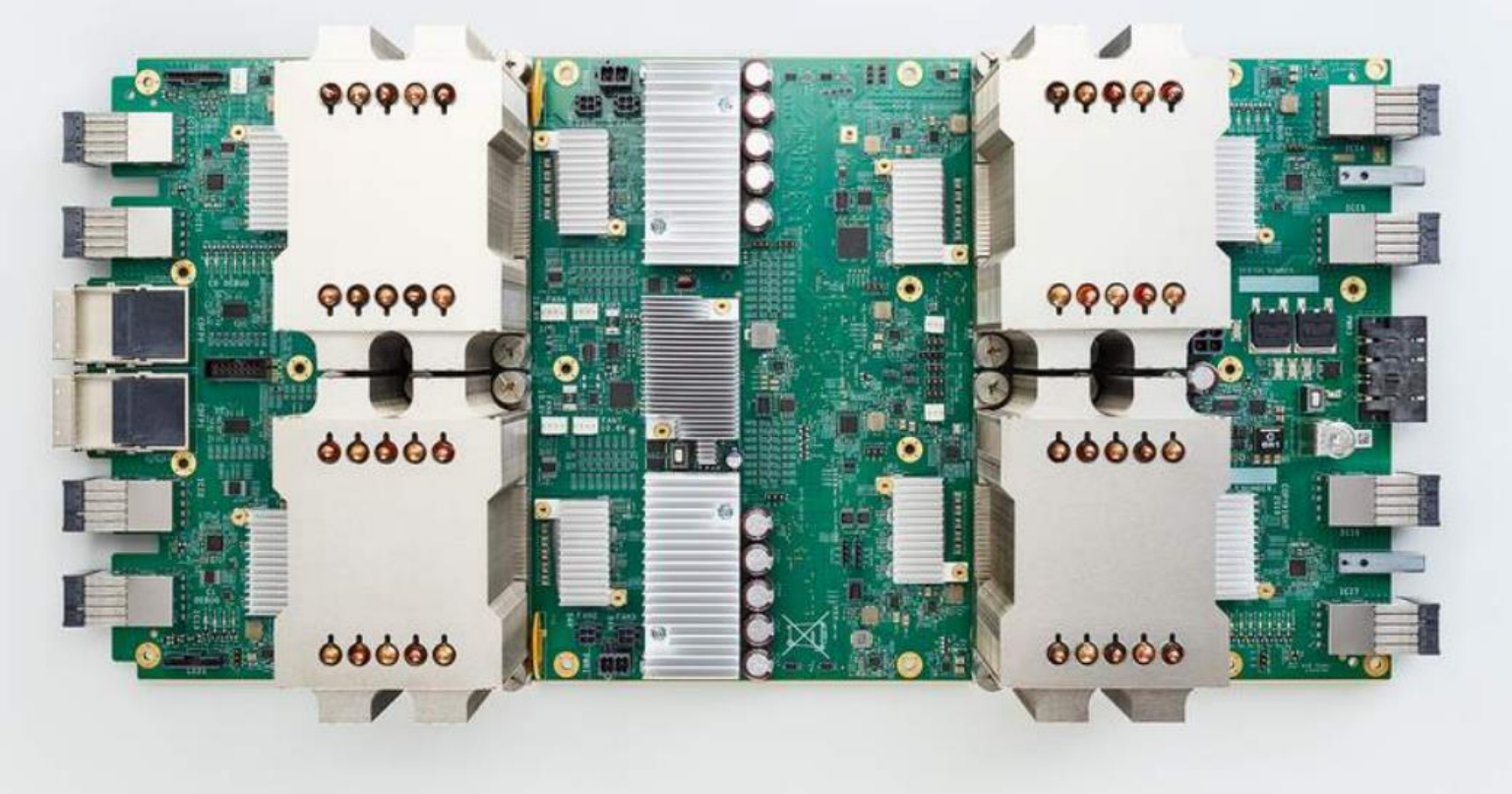
GPU vs TPU

	K80 2012	TPU 2015	P40 2016
Inferences/Sec <10ms latency	$^1/_{13}$X	1X	2X
Training TOPS	6 FP32	NA	12 FP32
Inference TOPS	6 FP32	90 INT8	48 INT8
On-chip Memory	16 MB	24 MB	11 MB
Power	300W	75W	250W
Bandwidth	320 GB/S	34 GB/S	350 GB/S

https://blogs.nvidia.com/blog/2017/04/10/ai-drives-rise-accelerated-computing-datacenter/

credits:
cs231n.stanford.edu, Song Han

Google Cloud TPU (v2 2017)



- Cloud TPU delivers up to 180 teraflops to train and run machine learning models.

— Google Blog

credits:
cs231n.stanford.edu, Song Han

Google TPU pods

- A “TPU pod” built with 64 second-generation TPUs delivers up to 11.5 petaflops of machine learning acceleration.
- “One of our new large-scale translation models used to take a full day to train on 32 of the best commercially-available GPUs—now it trains to the same accuracy in an afternoon using just **one eighth** of a TPU pod.”

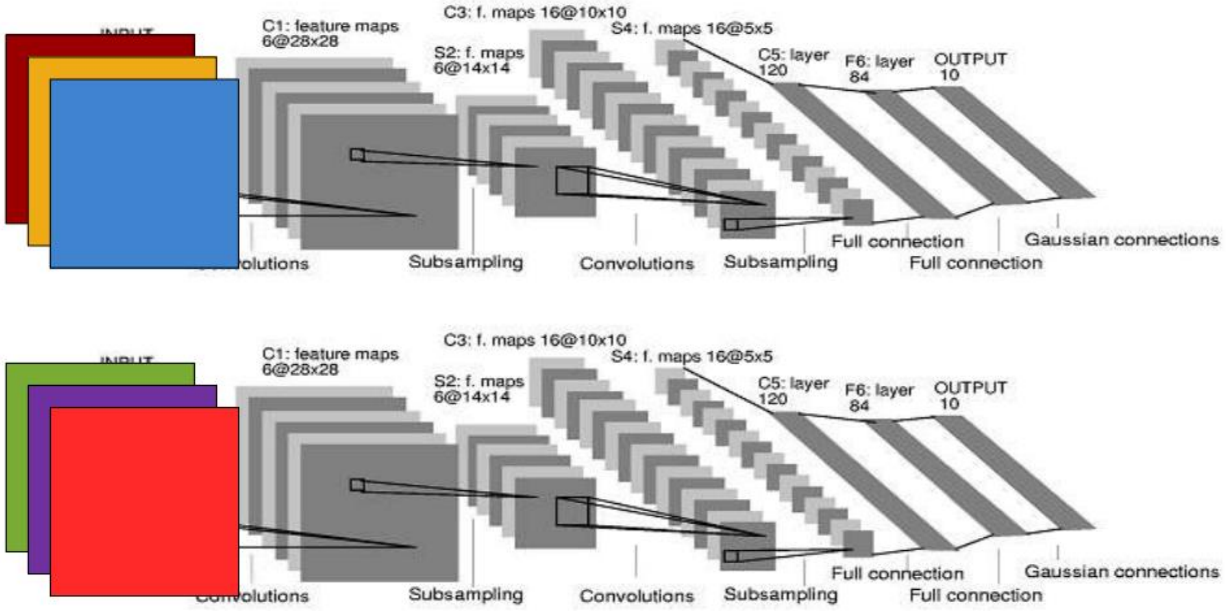
— Google Blog

credits:
cs231n.stanford.edu, Song Han

DEEP LEARNING
PARALLEL TRAINING

credits:
cs231n.stanford.edu, Song Han

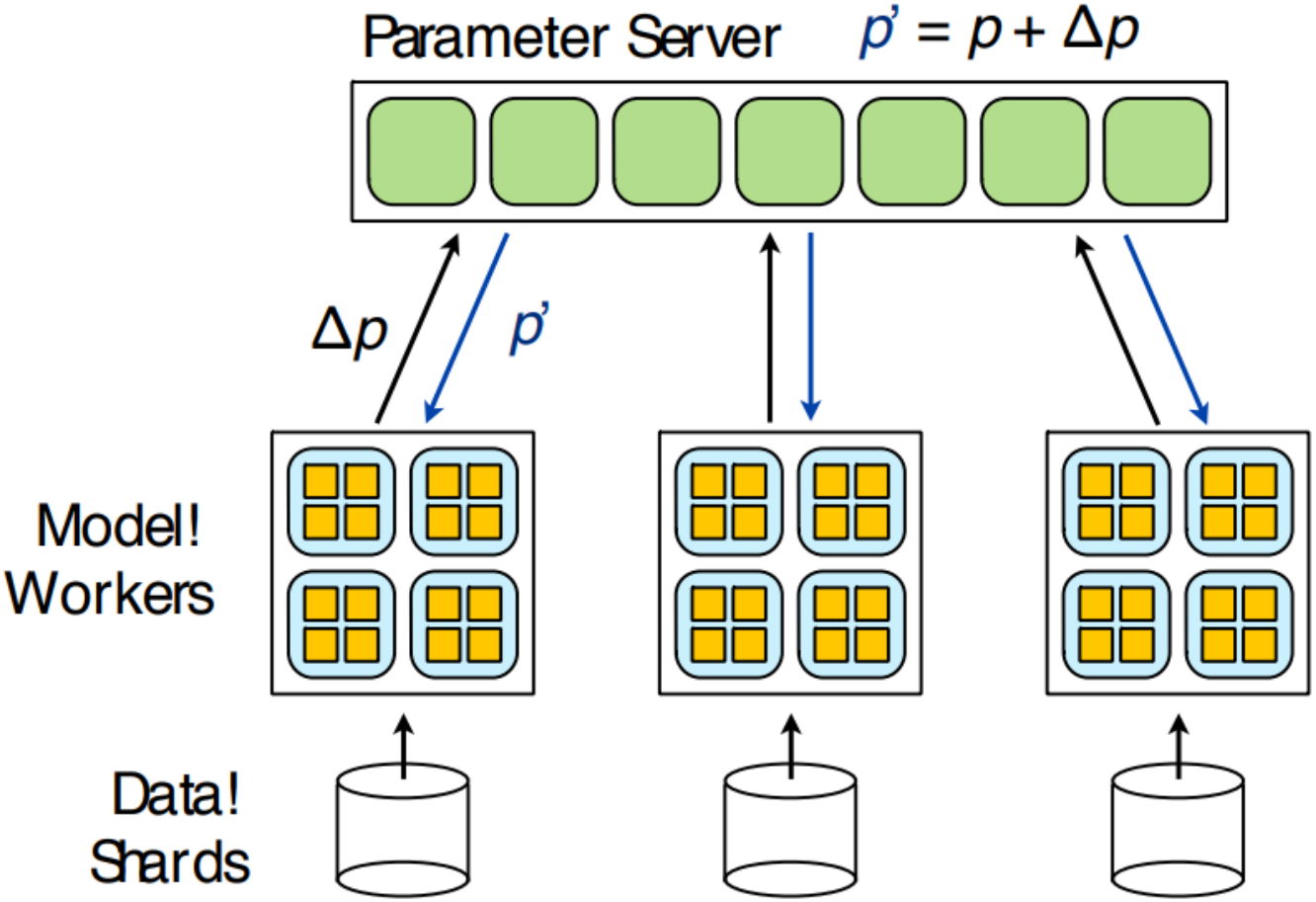
Data Parallel: run multiple inputs in parallel



- Doesn't affect latency for one input
- Requires P-fold larger batch size (i.e. limited scaling only)
- For training requires coordinated weight update

credits:
cs231n.stanford.edu, Song Han

The Need to Exchange Weight-deltas

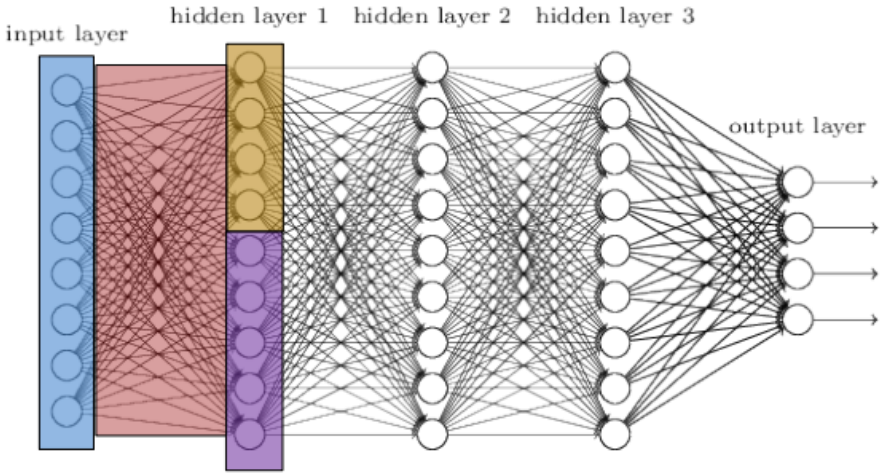


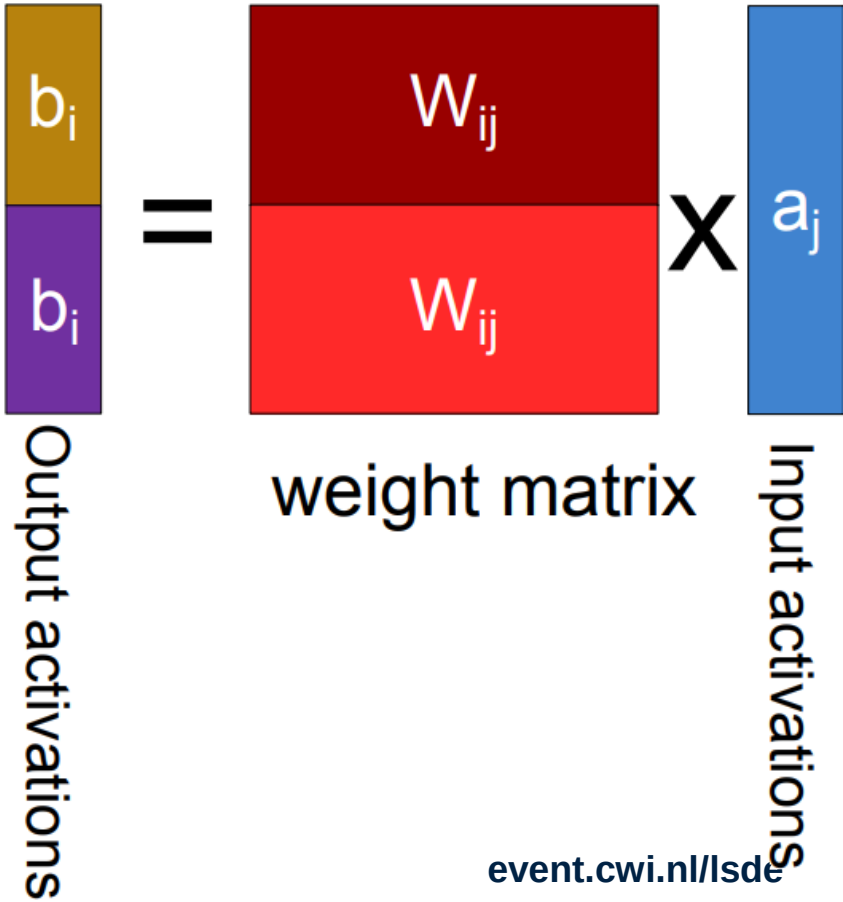
Large Scale Distributed Deep Networks, Jeff Dean et al., 2013

credits:
cs231n.stanford.edu, Song Han

Fully connected layers

- Parallelize by partitioning the weight matrix



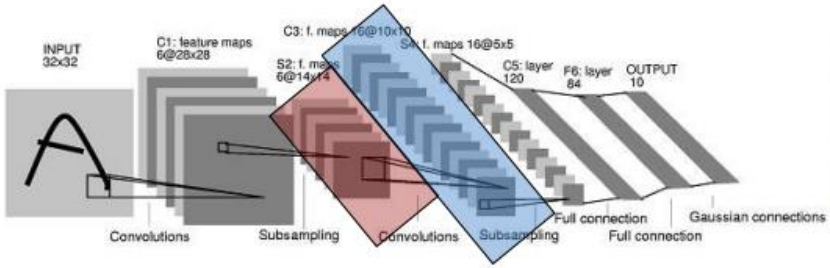


requires communicating the activations

credits:
cs231n.stanford.edu, Song Han

Convolution layers: easier to parallelize

- by output region (needs some communication around convolution borders)



Kernels
Multiple 3D
K_{uvkj}

x

A_{xyk}

B_{xyj} B_{xyj}
B_{xyj} B_{xyj}

6D Loop
Forall output map j
For each input map k
For each pixel x,y
For each kernel element u,v
B_{xyj} += $A_{(x-u)(y-v)k}$ x K_{uvkj}

Input maps
A_{xyk}

Output maps
B_{xyj}

credits:
cs231n.stanford.edu, Song Han

Multi-GPU training

- Servers with up to 8 GPUs
- Direct GPU-GPU communication
  - "NVLink" (2x150GB/s on Volta)

    (compare to 2x10Gb/s~=2GB/s Ethernet networks..)

P3

P3 instances are the latest generation of general purpose GPU instances.

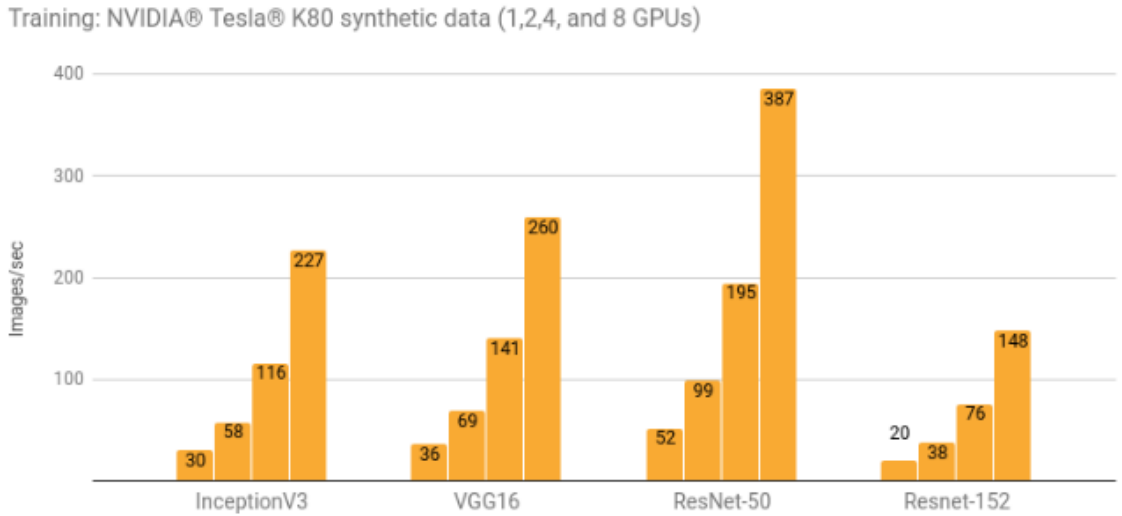
Features:

- Up to 8 NVIDIA Tesla V100 GPUs, each pairing 5,120 CUDA Cores and 640 Tensor Cores
- High frequency Intel Xeon E5-2686 v4 (Broadwell) processors
- Supports NVLink for peer-to-peer GPU communication
- Provide Enhanced Networking using Elastic Network Adapter with up to 25 Gbps of aggregate network bandwidth within a Placement Group

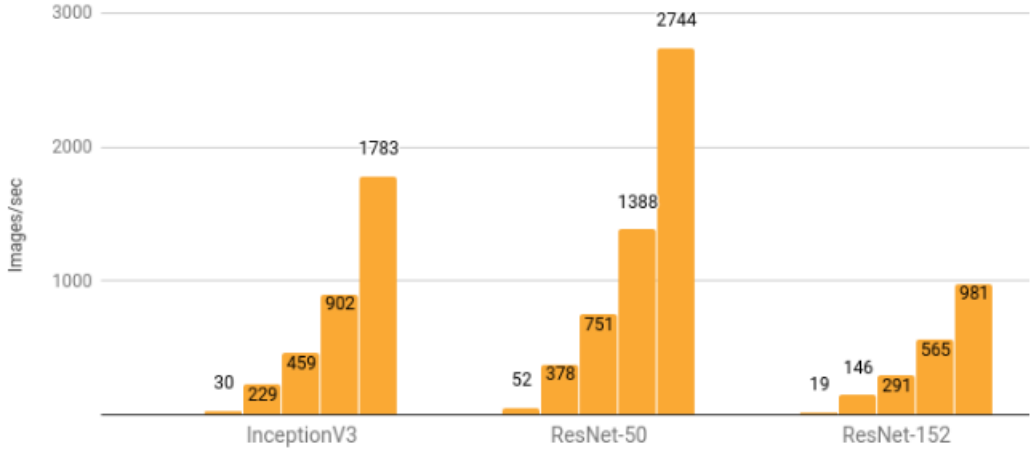
Model	GPUs	vCPU	Mem (GiB)	GPU Mem (GiB)	GPU P2P
p3.2xlarge	1	8	61	16	-
p3.8xlarge	4	32	244	64	NVLink
p3.16xlarge	8	64	488	128	NVLink

Paralellism in Tensorflow

- Multi-GPU (1 machine) training with normal TensorFlow



- Distributed TensorFlow: results for 1-8 machines (8-64 GPUs)

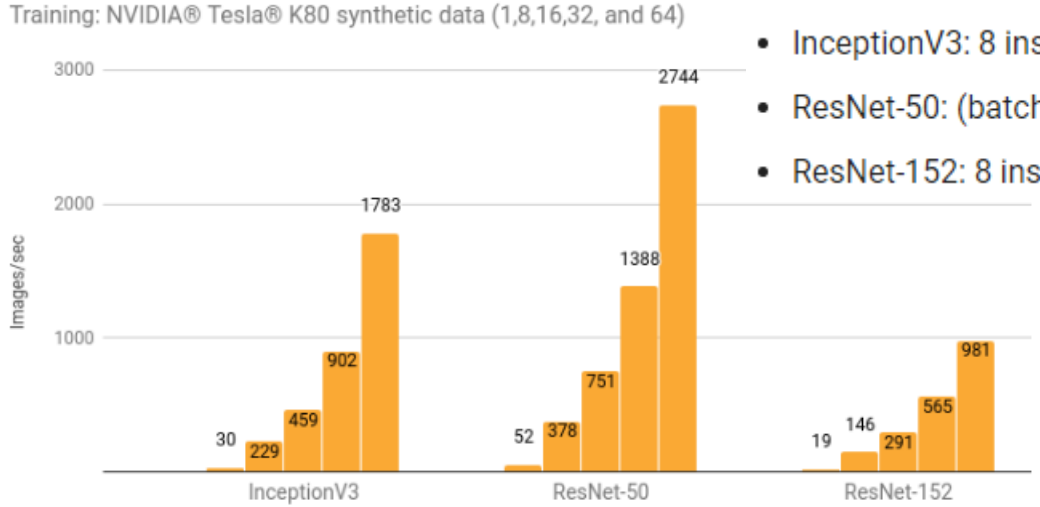


Paralellism in Tensorflow

Options	InceptionV3	ResNet-50	ResNet-152	AlexNet	VGG16
Batch size per GPU	64	64	32	512	32
Optimizer	sgd	sgd	sgd	sgd	sgd

Model	variable_update	local_parameter_device
InceptionV3	parameter_server	cpu
ResNet-50	replicated (without NCCL)	gpu
ResNet-152	replicated (without NCCL)	gpu
AlexNet	parameter_server	gpu
VGG16	parameter_server	gpu

- Distributed TensorFlow: results for 1-8 machines (8-64 GPUs)



- InceptionV3: 8 instances / 6 parameter servers
- ResNet-50: (batch size 32) 8 instances / 4 parameter servers
- ResNet-152: 8 instances / 4 parameter servers

Recap Parallelism

- Lots of parallelism in DNNs
  - 16M independent multiplies in one FC layer
  - Limited by overhead to exploit a fraction of this
- Hyper-parameter search parallelism (not discussed so far)
  - Train multiple networks in parallel with different parameters
- Data parallel
  - Run multiple training examples in parallel
  - Limited by batch size
- Model parallel
  - Split model over multiple processors
  - By layer
  - Conv layers by map region
  - Fully connected layers by output activation

Summary: Deep Learning

..on Big Data, ..in the Cloud

- popular frameworks: TensorFlow, pyTorch, Caffe2, MXNET
- algorithmic optimizations ➔ making networks smaller
  - quantization, pruning, mixed-precision
- hardware for deep learning
  - CPUs (SIMD), GPUs, TPUs
- parallel training: does deep learning scale?
  - Trivially Distributed: Hyper-parameter search (e.g. tensorflow-on-spark)
  - Multi-GPUs in one machine (P2P GPU communications - NVLink)
  - Distributed TensorFlow?