

25 YEARS ANNIVERSARY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Lesson 6: Software and hardware for deep learning

Outline

1. Hardware for deep learning (CPU vs GPU)
2. Deep learning frameworks
3. Accelerator and compression tools

Hardware for deep learning (CPU vs GPU)

A not so common computer

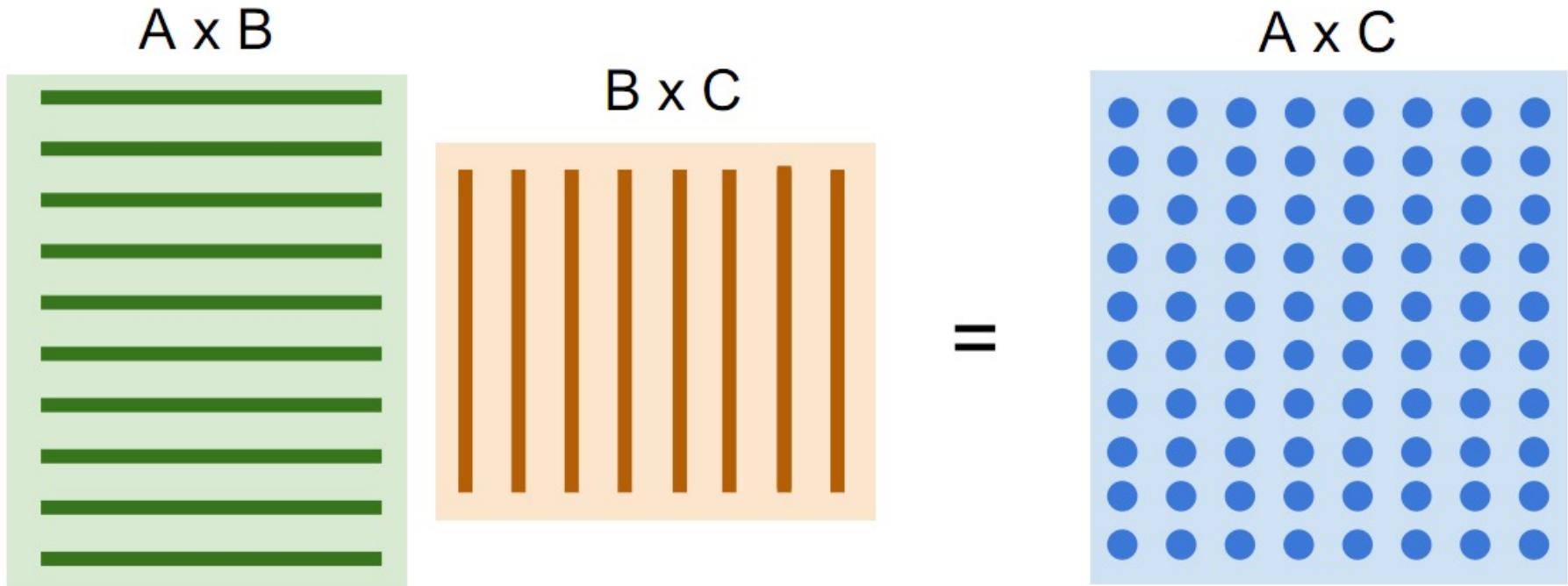


CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$385	~540 GFLOPs FP32
GPU (NVIDIA RTX 2080 Ti)	3584	1.6 GHz	11 GB GDDR6	\$1199	~13.4 TFLOPs FP32

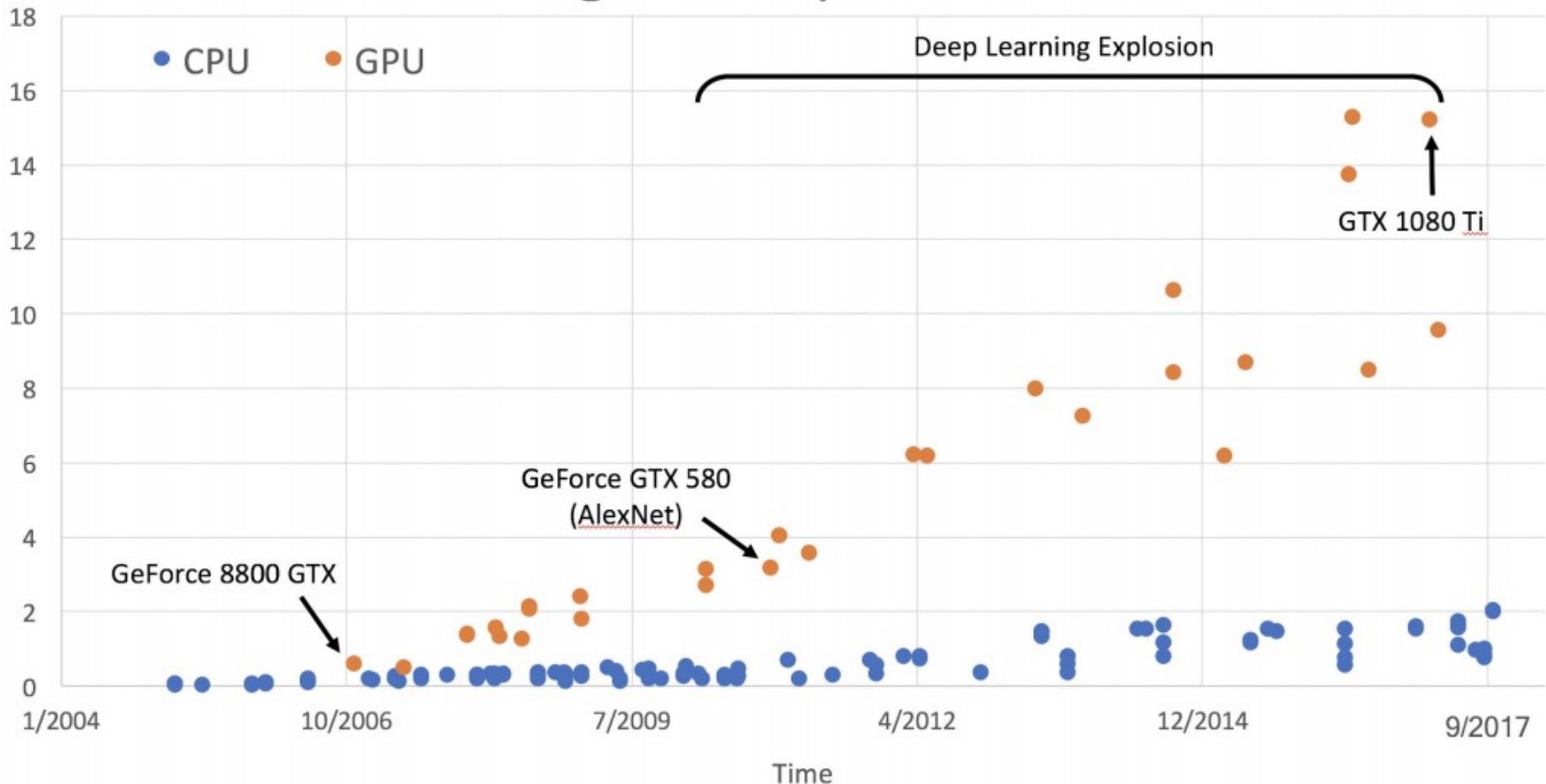
- CPU: fewer cores, but each core is much faster and much more capable, great at sequential tasks.
- GPU: more cores, but each core is much slower and “dumber”, great for parallel tasks.

Example: Matrix multiplication



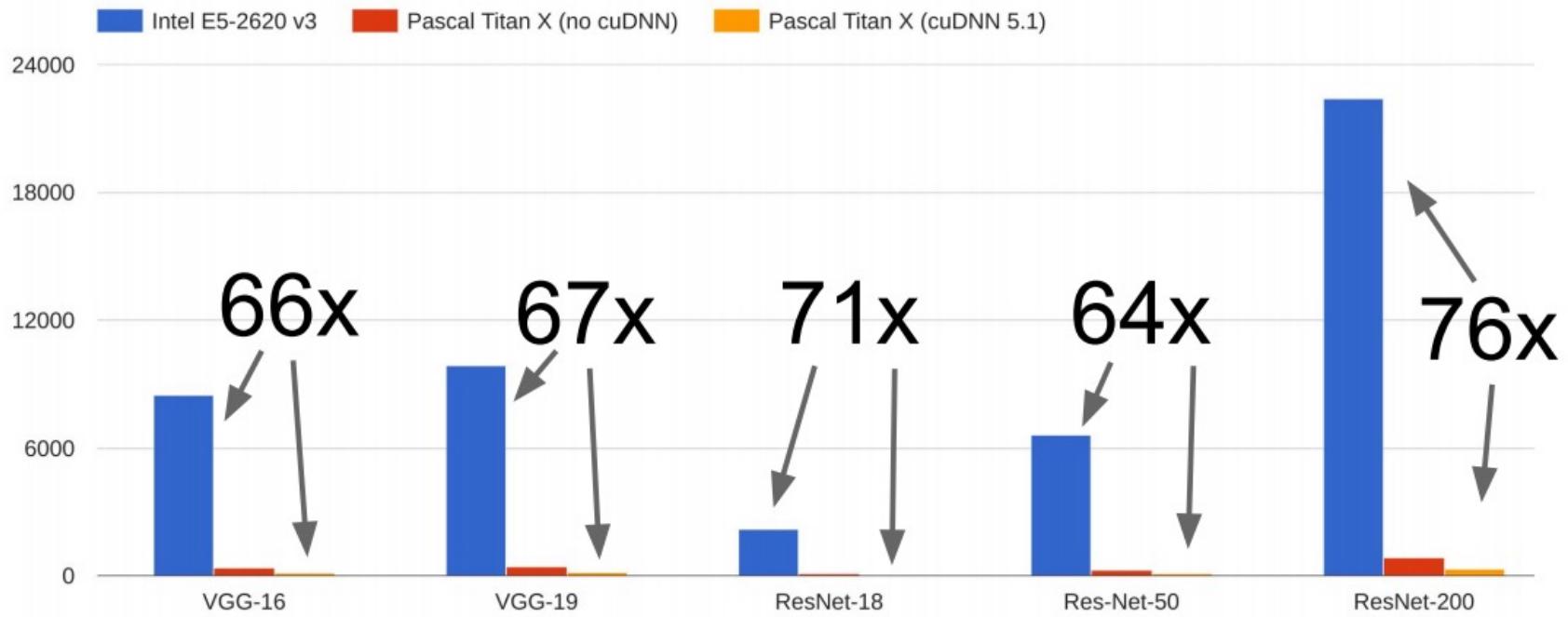
- More efficient with GPU

GigaFLOPs per 1\$



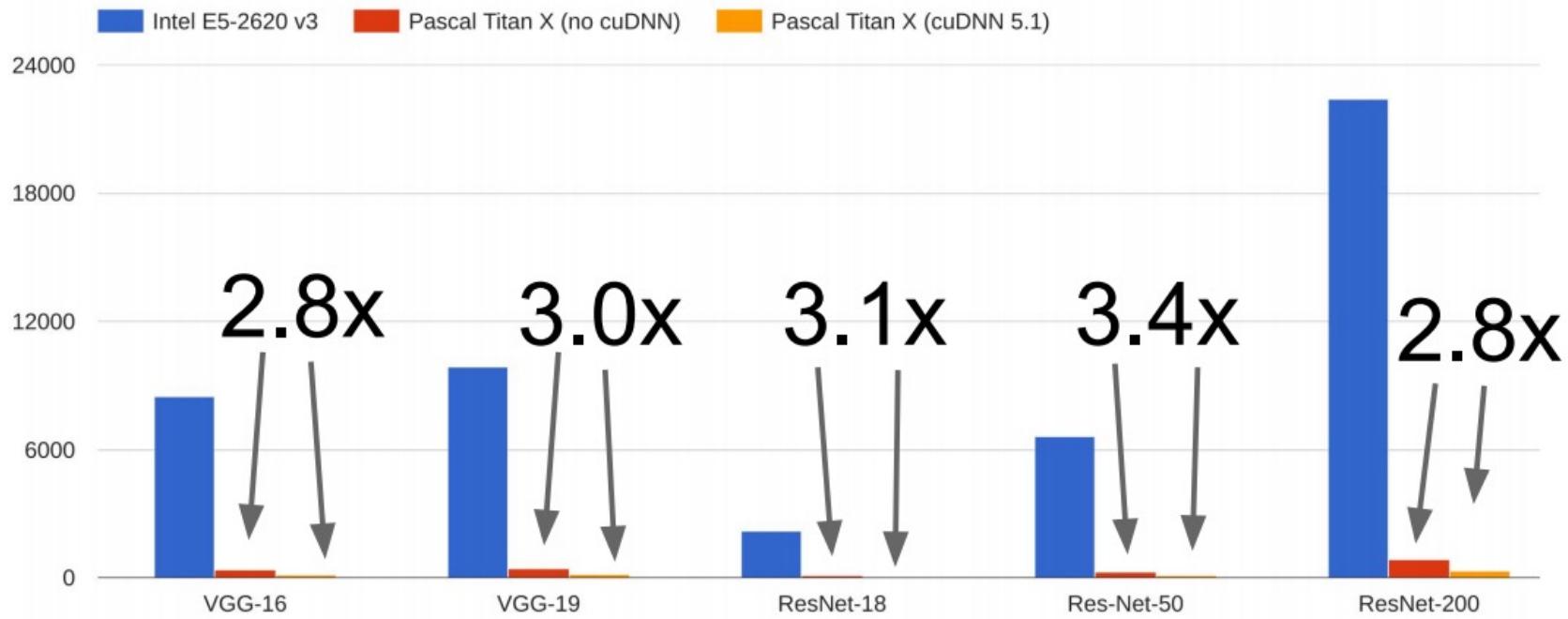
CPU vs GPU in practice

N=16 Forward + Backward time (ms)



CPU vs GPU in practice (2)

N=16 Forward + Backward time (ms)



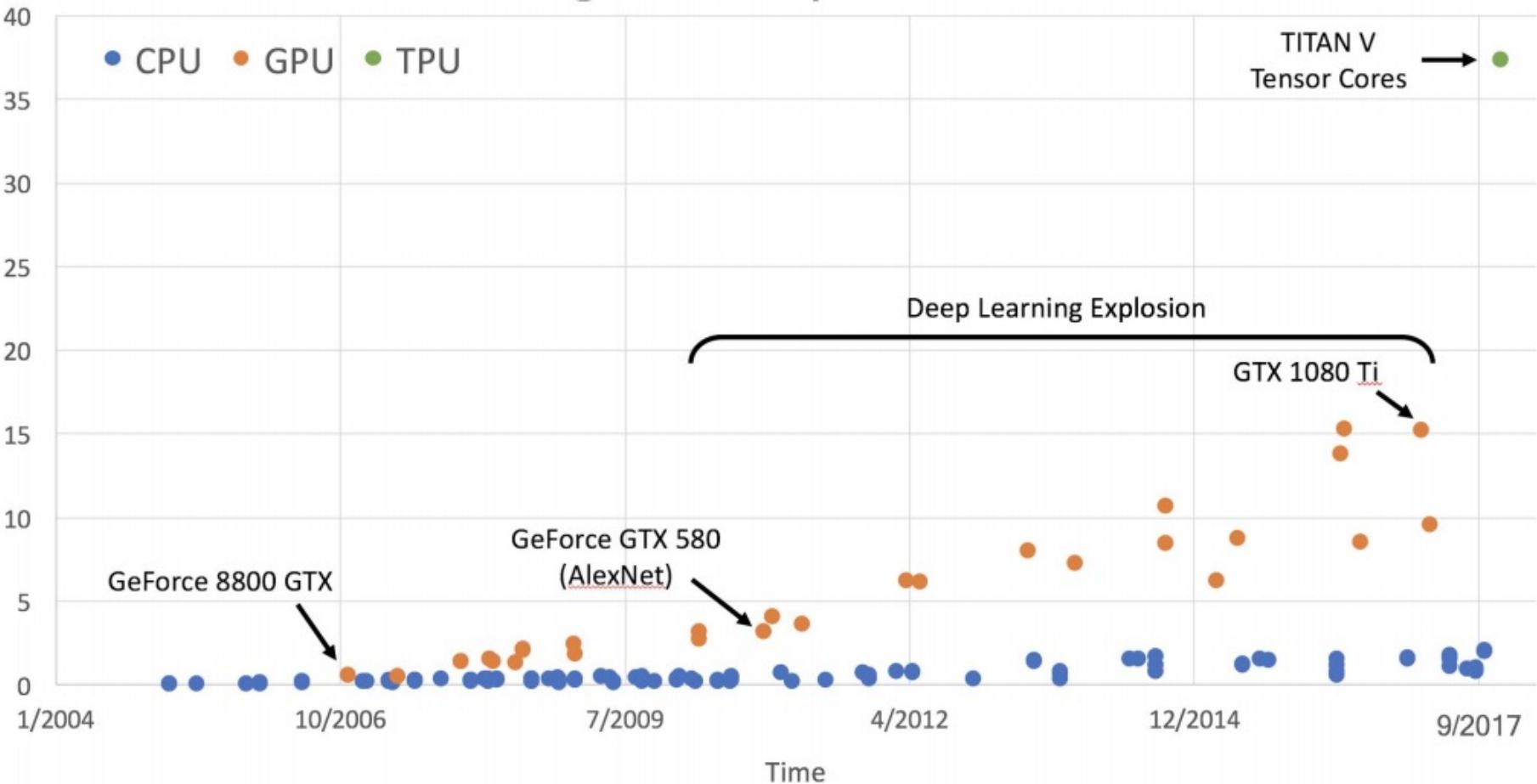
cuDNN much faster than “unoptimized” CUDA

CPU vs GPU vs TPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$385	~540 GFLOPs FP32
GPU (NVIDIA RTX 2080 Ti)	3584	1.6 GHz	11 GB GDDR6	\$1199	~13.4 TFLOPs FP32
TPU NVIDIA TITAN V	5120 CUDA, 640 Tensor	1.5 GHz	12GB HBM2	\$2999	~14 TFLOPs FP32 ~112 TFLOP FP16
TPU Google Cloud TPU	?	?	64 GB HBM	\$4.50 per hour	~180 TFLOP

- TPU: Specialized hardware for deep learning

GigaFLOPs per 1\$

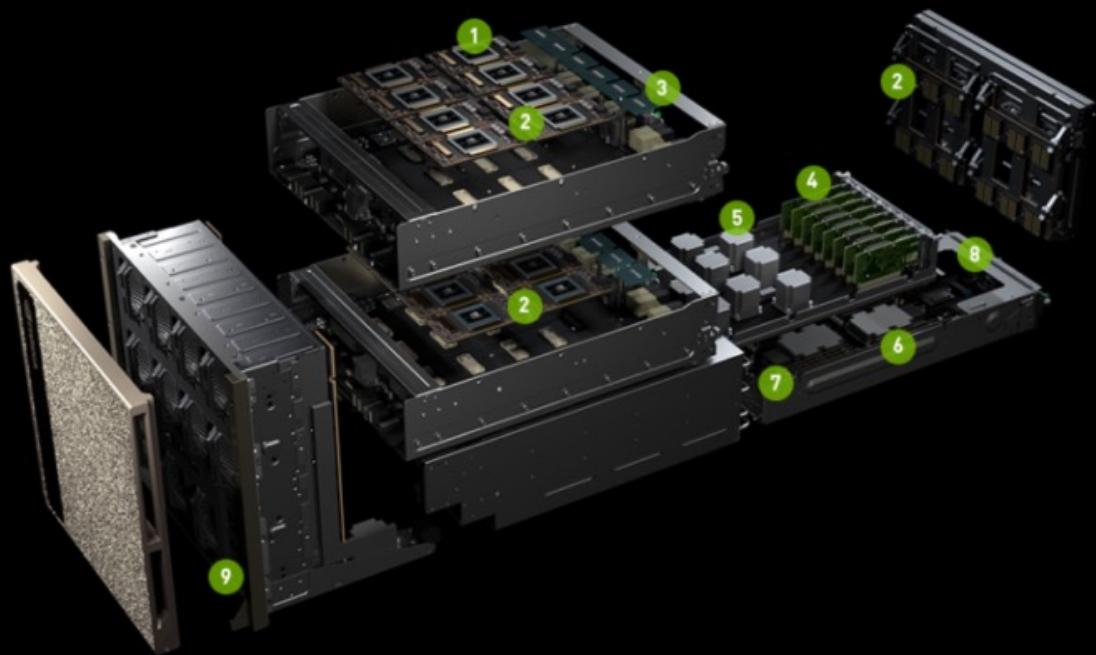


NVIDIA DGX-2

NVIDIA DGX-2

Explore the powerful components of DGX-2.

- ➊ NVIDIA TESLA V100 32GB, SXM3
- ➋ 16 TOTAL GPUS FOR BOTH BOARDS, 512GB TOTAL HBM2 MEMORY
Each GPU board with 8 NVIDIA Tesla V100.
- ➌ 12 TOTAL NVSWITCHES
High Speed Interconnect, 2.4 TB/sec bisection bandwidth.
- ➍ 8 EDR INFINIBAND/100 GbE ETHERNET
1600 Gb/sec Bi-directional Bandwidth and Low-Latency.
- ➎ PCIE SWITCH COMPLEX
- ➏ TWO INTEL XEON PLATINUM CPUS
- ➐ 1.5 TB SYSTEM MEMORY
- ➑ DUAL 10/25 GbE ETHERNET
- ➒ 30 TB NVME SSDS INTERNAL STORAGE



NVidia edge computing

Jetson Nano

129 USD

NVIDIA Maxwell™ architecture with 128
NVIDIA CUDA® cores

Jetson TX2 Series

Starting at 249 USD

NVIDIA Pascal™ architecture with 256
NVIDIA CUDA cores

Jetson Xavier NX

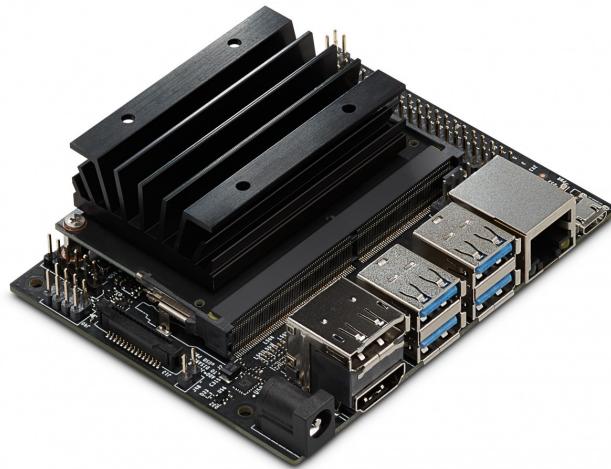
Starting at 399 USD

NVIDIA Volta™ architecture with 384
NVIDIA CUDA cores and 48 Tensor
cores

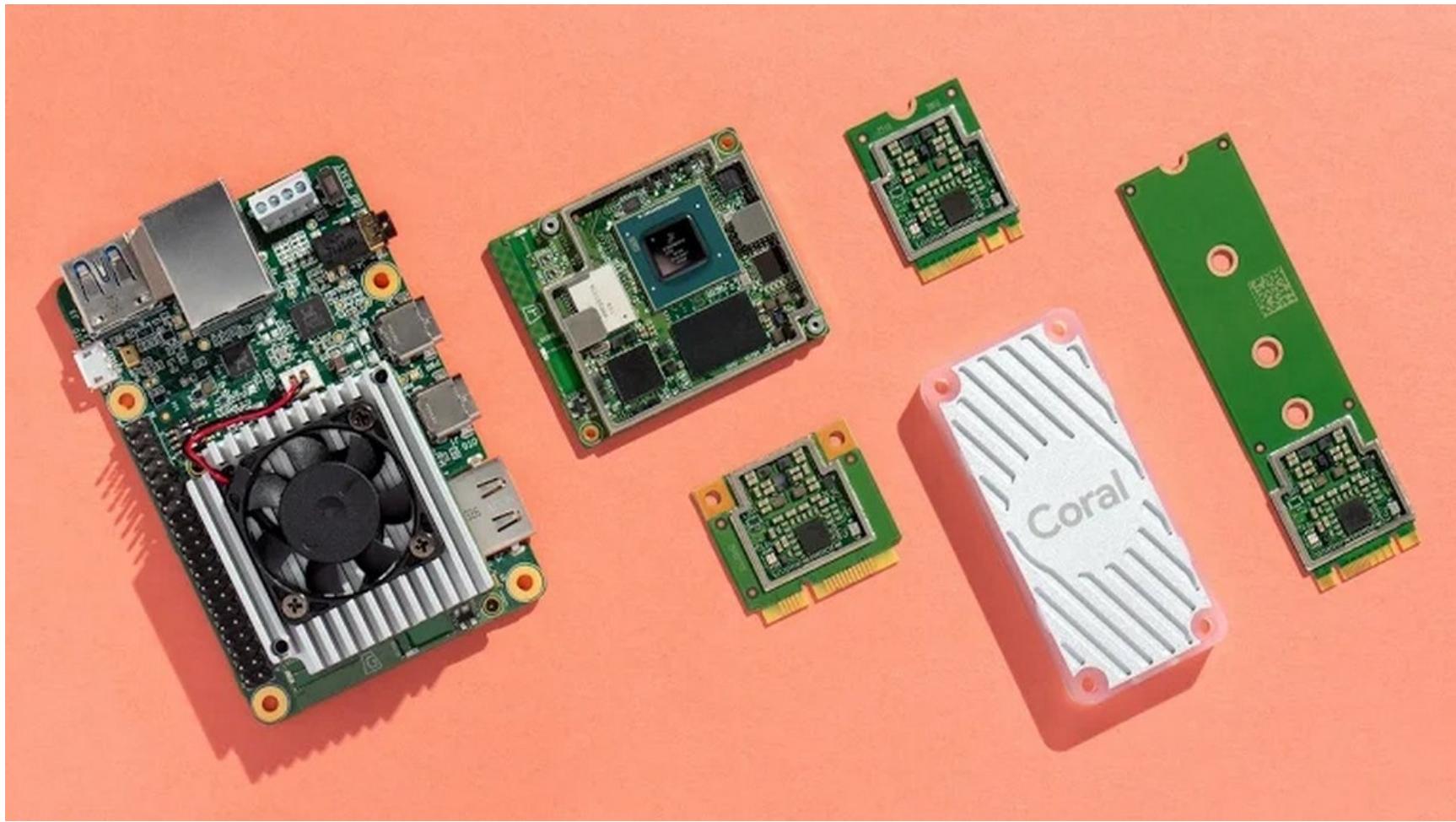
Jetson AGX Xavier Series

Starting at 599 USD

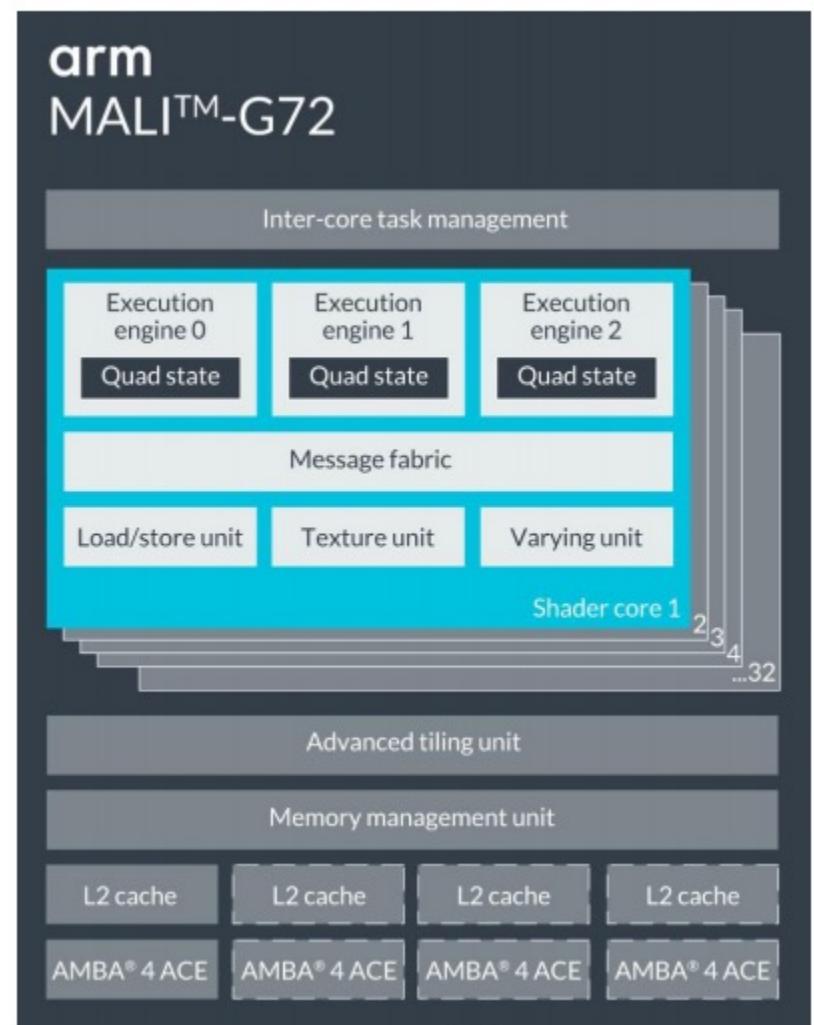
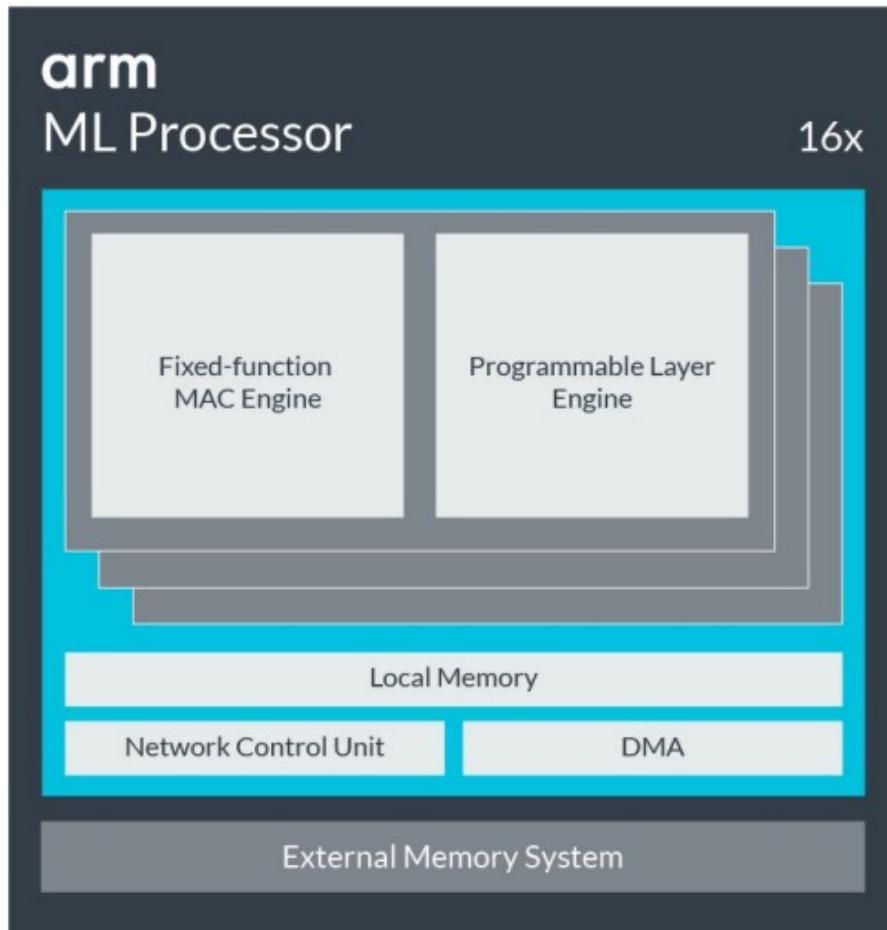
NVIDIA Volta™ architecture with up to
512 NVIDIA CUDA cores and up to 64
Tensor cores



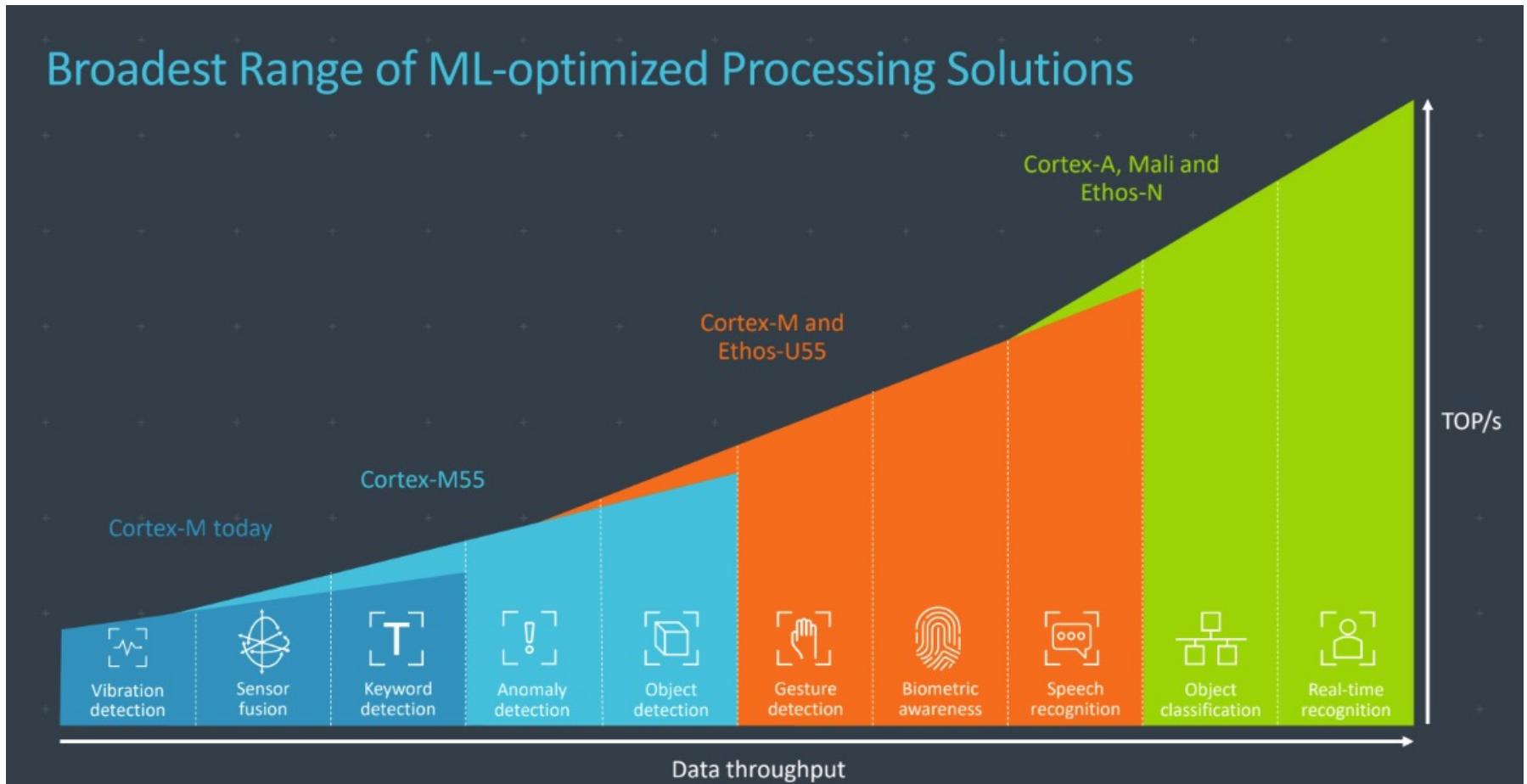
Google Coral



ARM edge computing



ARM NPU

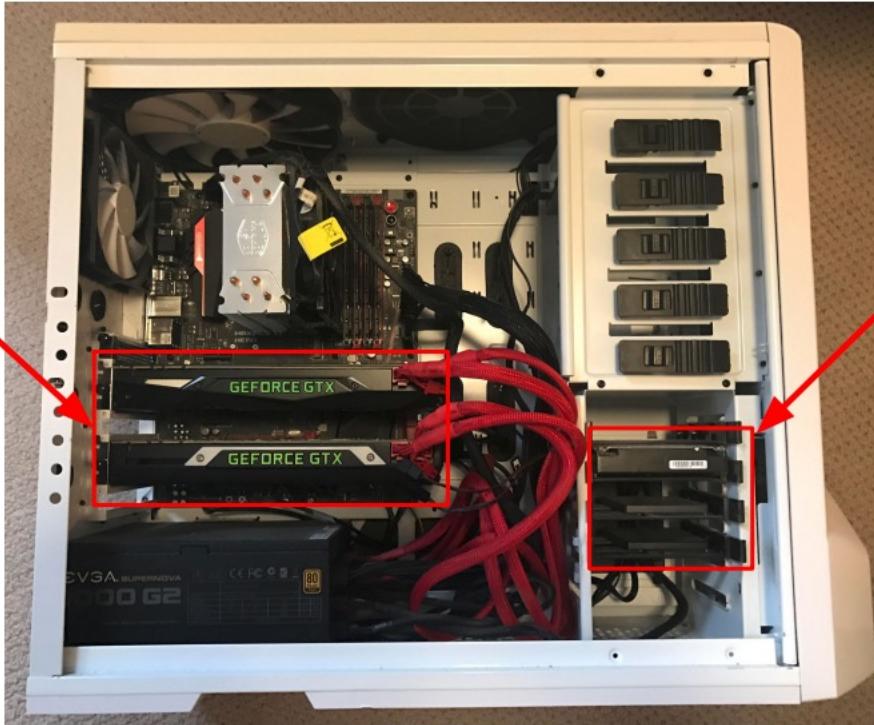


Programming GPUs

- CUDA (NVIDIA only)
 - Write C-like code that runs directly on the GPU
 - Optimized APIs: cuBLAS, cuFFT, cuDNN, etc
- OpenCL
 - Similar to CUDA, but runs on anything
 - Usually slower on NVIDIA hardware
- HIP <https://github.com/ROCm-Developer-Tools/HIP>
 - New project that automatically converts CUDA code to something that can run on AMD GPUs

CPU / GPU Communication

Model
is here

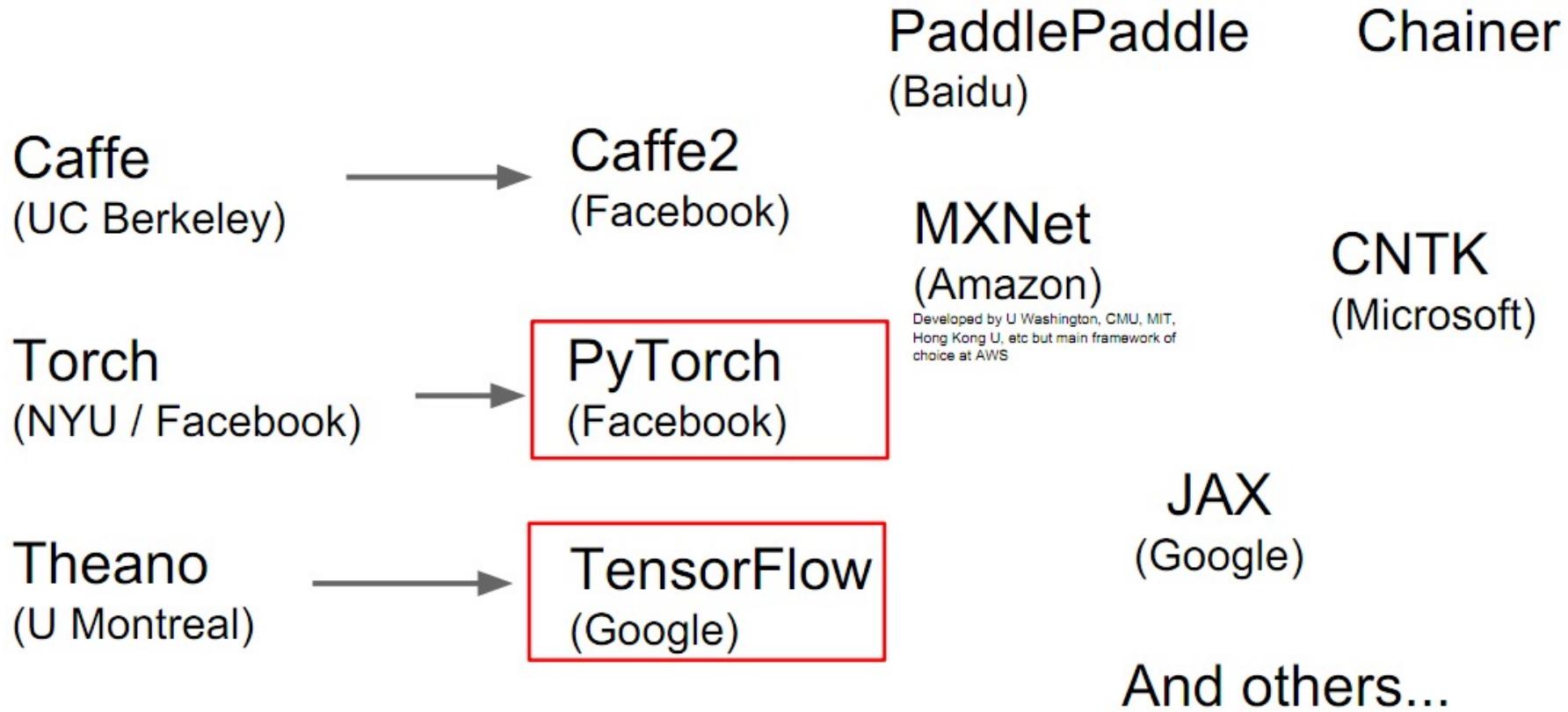


Data is here

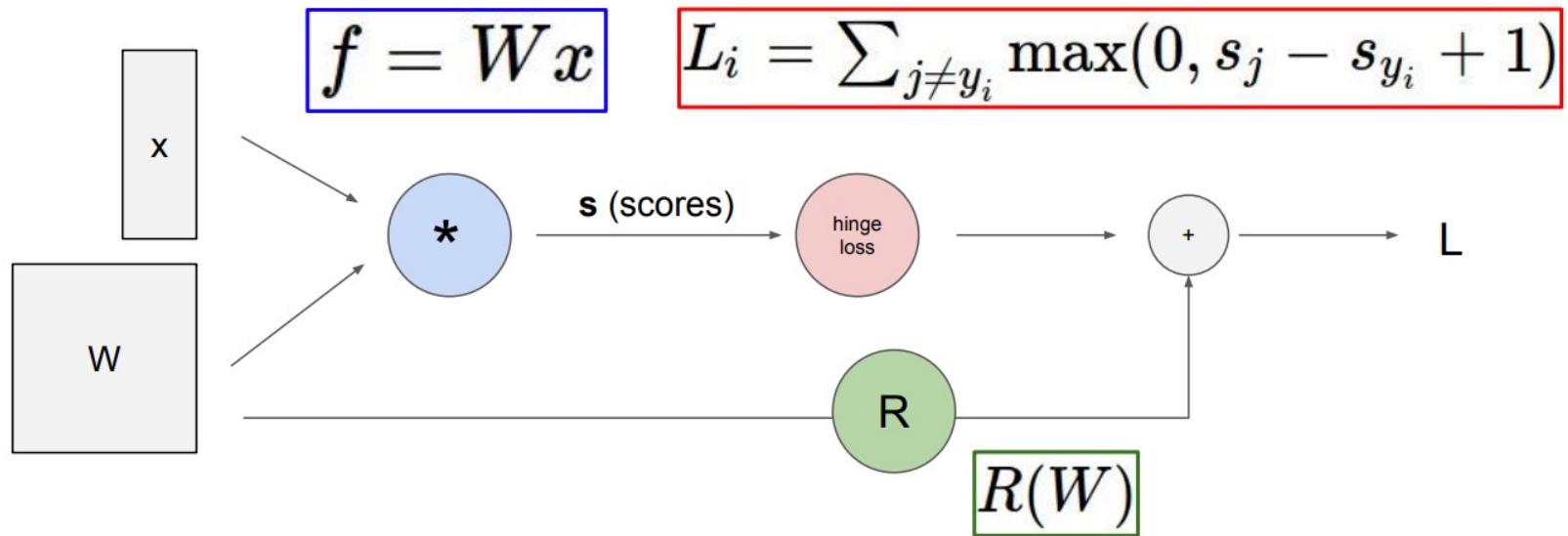
- If you aren't careful, training can bottleneck on reading data and transferring to GPU!
- Solutions:
 - Read all data into RAM
 - Use SSD instead of HDD
 - Use multiple CPU threads to prefetch data

Deep learning frameworks

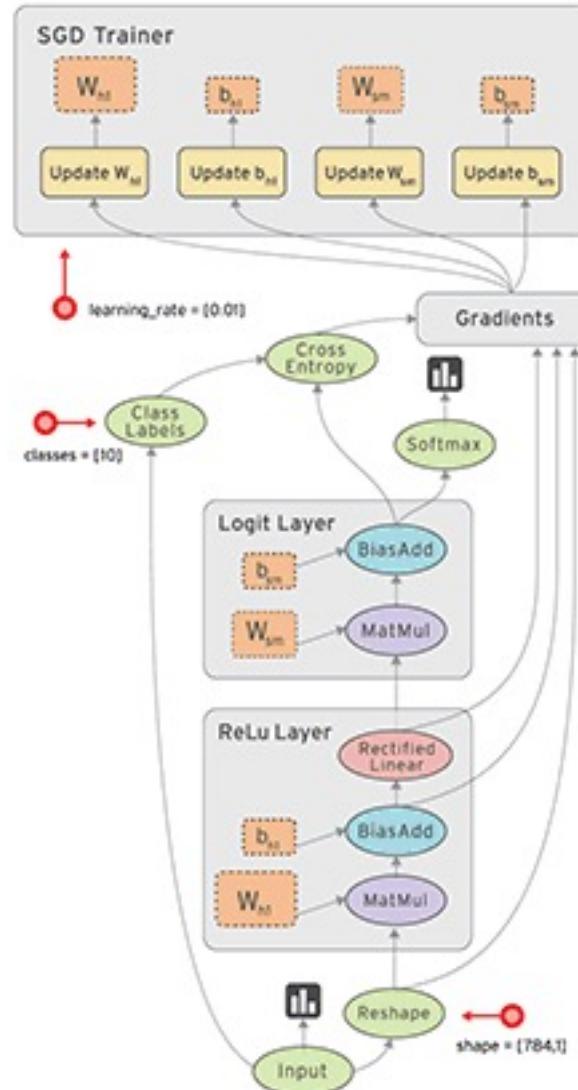
There are many ...



Computational graphs



Computational graphs (2)



Computational graphs (3)

Numpy

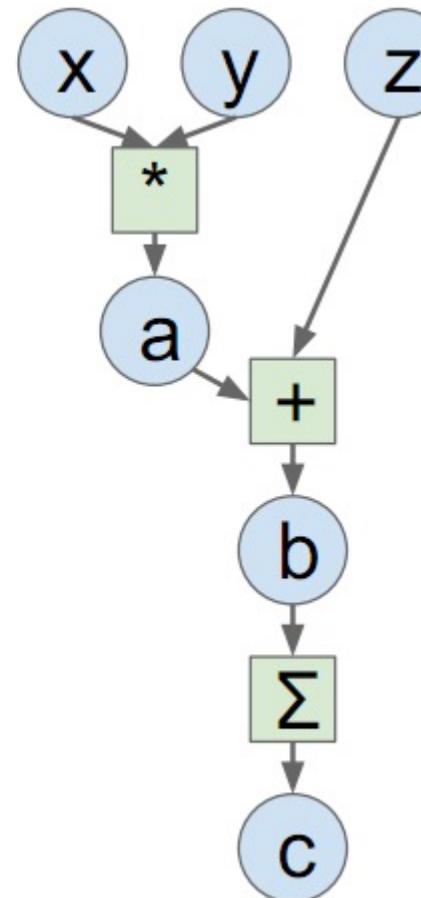
```
import numpy as np
np.random.seed(0)

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
```



- **Pros:**
 - Clean API, easy to write numeric code
- **Cons:**
 - Have to compute our own gradients
 - Can't run on GPU

Computational graphs (4)

Numpy

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

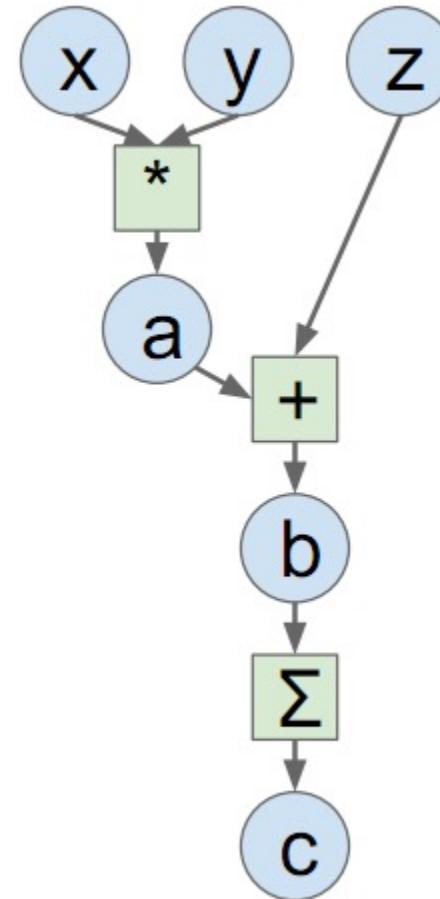
N, D = 3, 4

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
```

- Lập trình giống như Numpy!



PyTorch

```
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

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

Computational graphs (5)

Numpy

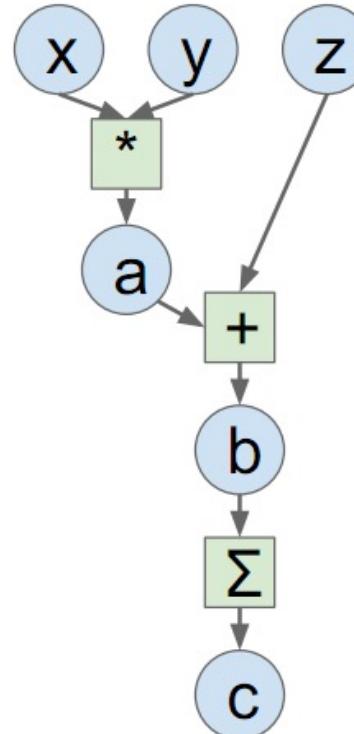
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

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

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

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

c.backward()
print(x.grad)
```

- PyTorch can do auto gradient

The point of deep learning frameworks

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

PyTorch: Tensors

- Like a numpy array, but can run on GPU
- Data model and API are too much the same!
- Here is an example of training a two-layer neural network using PyTorch Tensors

```
import torch

device = torch.device('cpu')

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

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

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

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

PyTorch: Autograd

- Create Tensors with `requires_grad=True` to enable autograd
- `Torch.no_grad` means don't include this in the calculation graph
- We need to set the gradients to zero before starting to do back propagation because PyTorch *accumulates the gradients* on subsequent backward passes. This is convenient while training RNNs.

```
import torch

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

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

    loss.backward()

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

PyTorch: nn

- Higher-level wrapper to work with neural networks
- Make programming easier

```
import torch

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

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

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

PyTorch: nn (2)

- It is possible to define new modules in PyTorch
- Modules can contain weights or other modules
- PyTorch automatically handles Autograd for new modules

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

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

model = TwoLayerNet(D_in, H, D_out)

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

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

PyTorch: optim

- Optimization algorithms are available in PyTorch, such as Adam

```
import torch

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

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

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

PyTorch: Pretrained models

- PyTorch has several pre-trained models available.
- These models can be used directly.

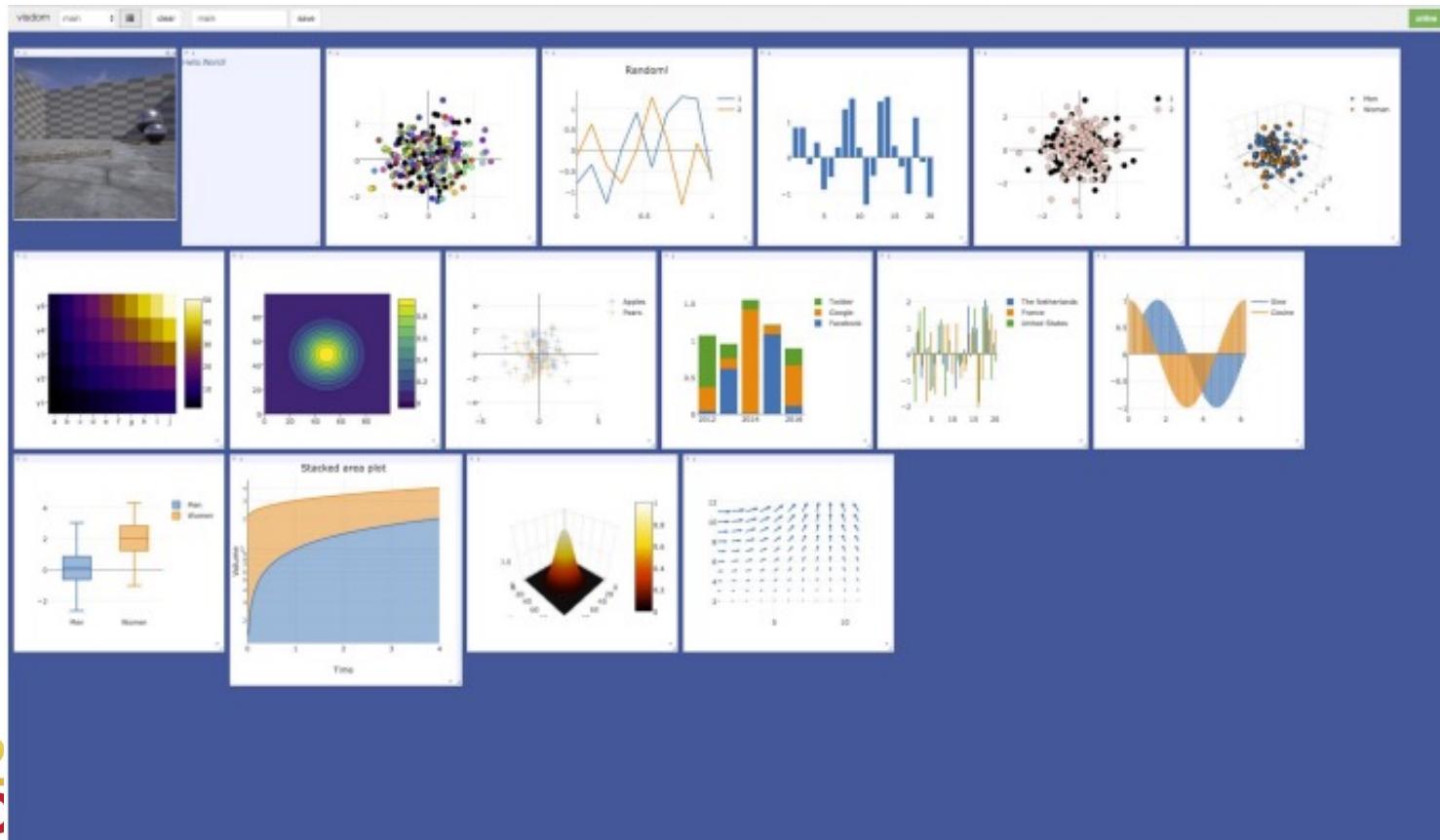
Super easy to use pretrained models with torchvision
<https://github.com/pytorch/vision>

```
import torch
import torchvision

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

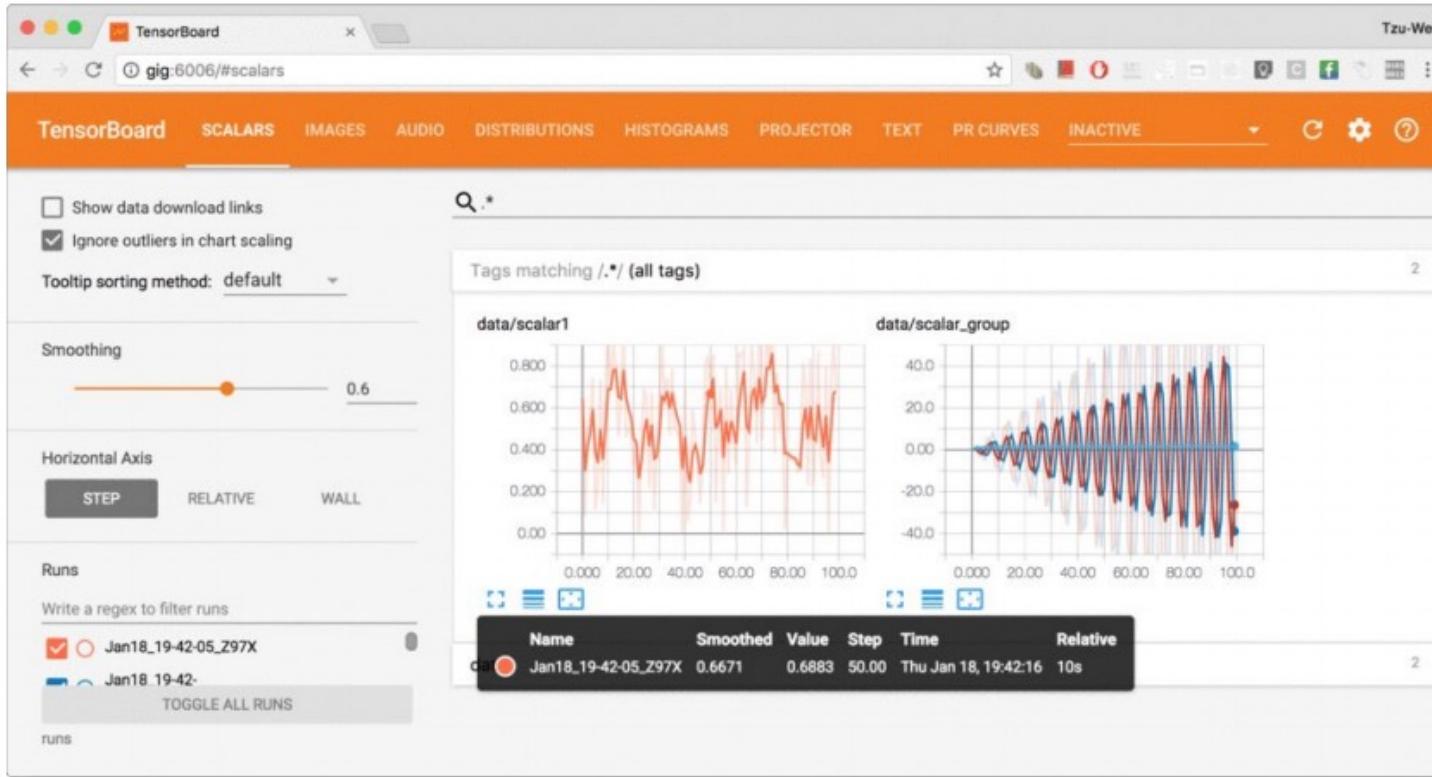
PyTorch: Visdom

- Tool to help visualize the calculation process
- Currently does not support the visualization of computational graph structures



PyTorch: tensorboardx

- A python wrapper around Tensorflow's web-based visualization tool.
- pip install tensorboardx
- <https://github.com/lanpa/tensorboardX>



PyTorch: Dynamic computational graph

- Create tensor



```
import torch

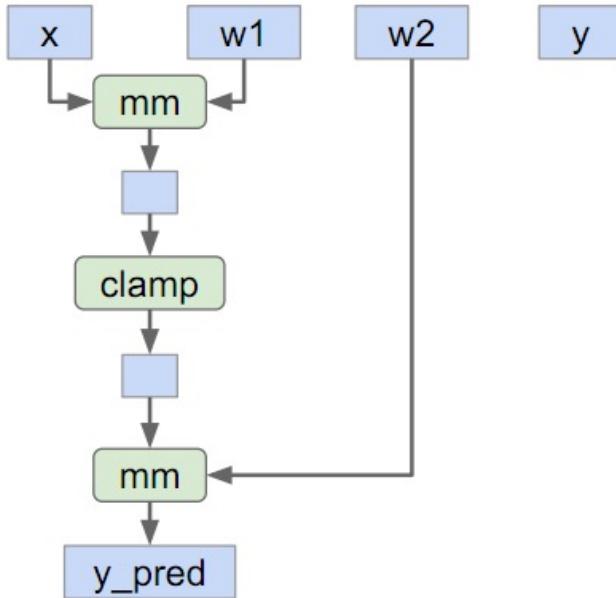
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

    loss.backward()
```

PyTorch: Dynamic computational graph (2)

- Build graph and perform computation



```
import torch

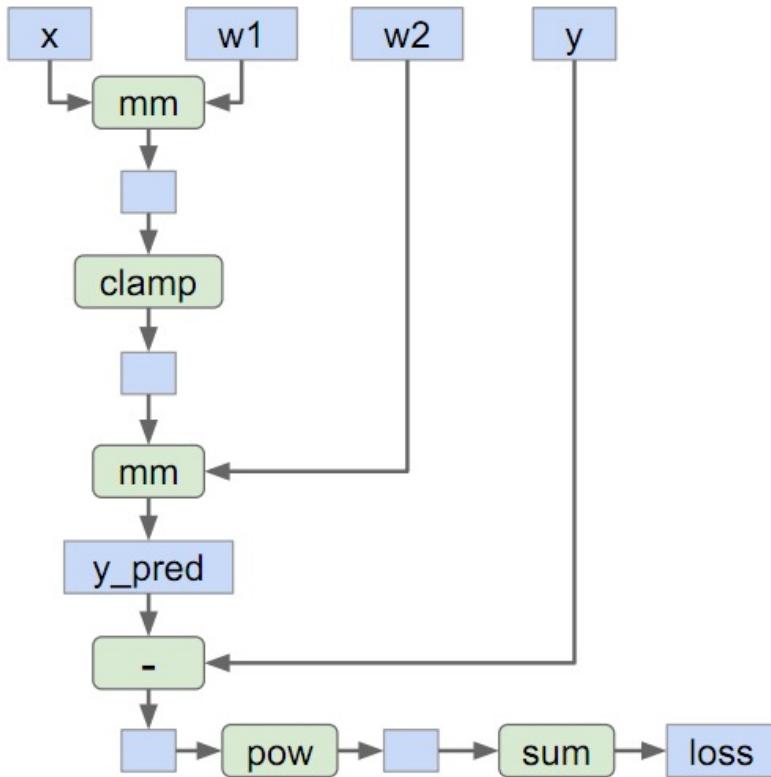
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

    loss.backward()
```

PyTorch: Dynamic computational graph (3)

- Build graph and perform computation



```
import torch

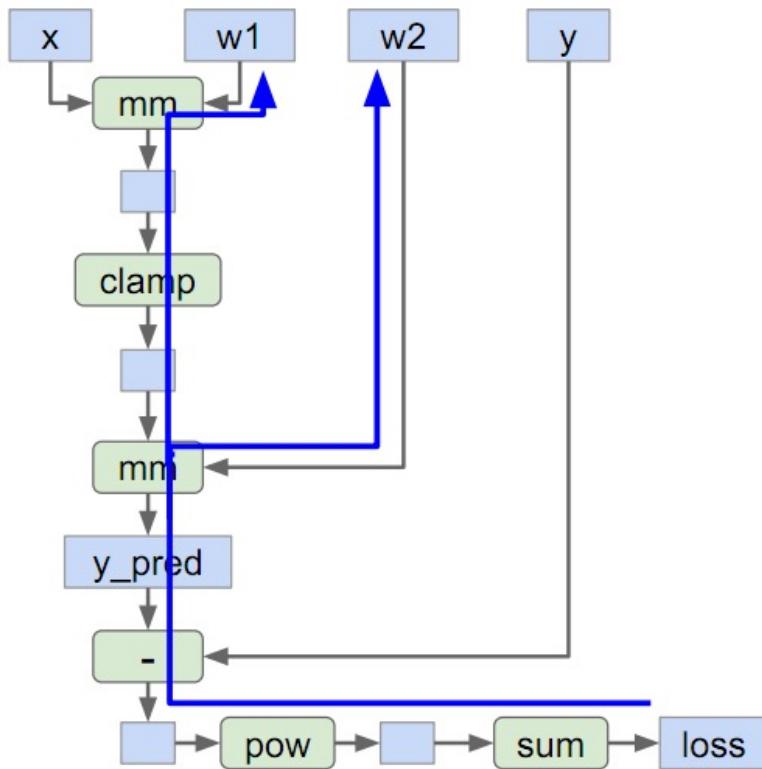
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

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

    loss.backward()
```

PyTorch: Dynamic computational graph (4)

- Find the path on the graph from the objective function to w1 and w2 for backprop, then do the calculation



```
import torch

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

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

    loss.backward()
```

PyTorch: Dynamic computational graph (5)

- On the next iteration, delete all the graphs and backpropagation in the previous step, rebuild all from scratch
- Seems inefficient, especially when building the same graph multiple times...

x

w1

w2

y

```
import torch

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

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

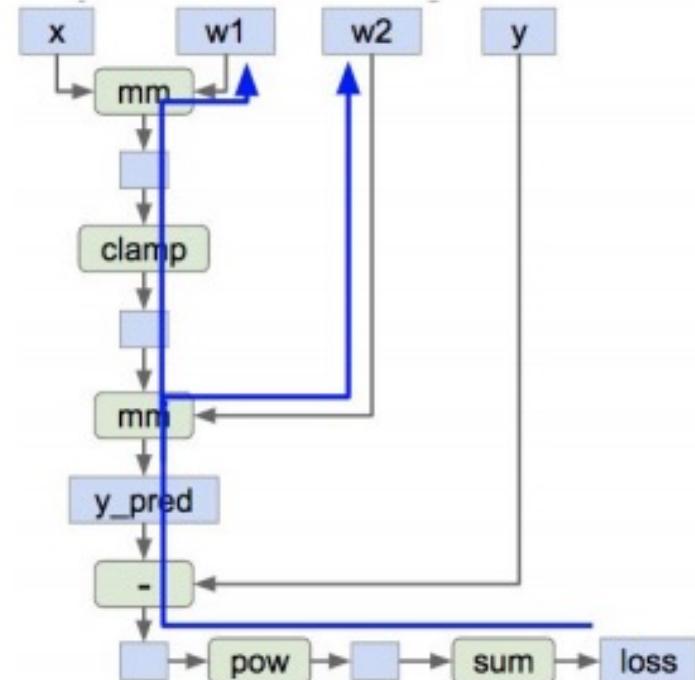
    loss.backward()
```

PyTorch: Static computation graphs

Static graph

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

Step 2: Reuse the same graph on every iteration



```
graph = build_graph()

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

Tensorflow Pre2.0

- Step 1:
Build a
calculation
graph
- Step 2: Run
this
calculation
graph
several
times

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

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Tensorflow 2.0

- TensorFlow's Eager Execution mode is an imperative programming environment that allows immediate operations to be executed without the need to build a computation graph
 - operations return specific values instead of building a graph of the calculation and run it later.
- This makes it easier to get started with TensorFlow models and easier to debug.

Tensorflow 2.0 vs Pre2.0

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

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2]).
```

Tensorflow 2.0:
“Eager” Mode by default

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

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D)}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Tensorflow 1.13

Tensorflow 2.0 vs Pre2.0

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

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
    gradients = tape.gradient(loss, [w1, w2]).
```

Tensorflow 2.0:
“Eager” Mode by default

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

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                  feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Tensorflow 1.13

Tensorflow 2.0: Neural Network

- Turn numpy array into TF tensor

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

```
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights
```

```
with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2]).
```

Tensorflow 2.0: Neural Network

- Use `tf.GradientTape()` to build dynamic computational graphs

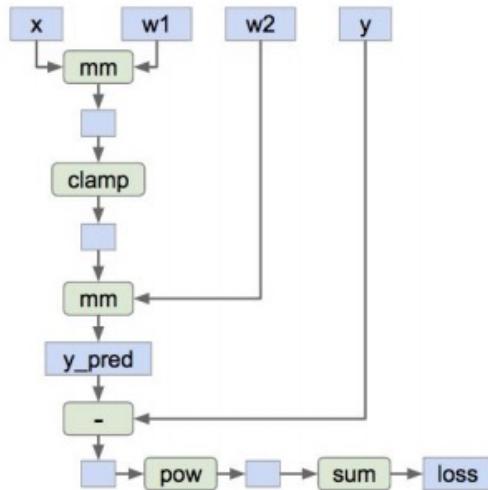
```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
    gradients = tape.gradient(loss, [w1, w2]).
```

Tensorflow 2.0: Neural Network

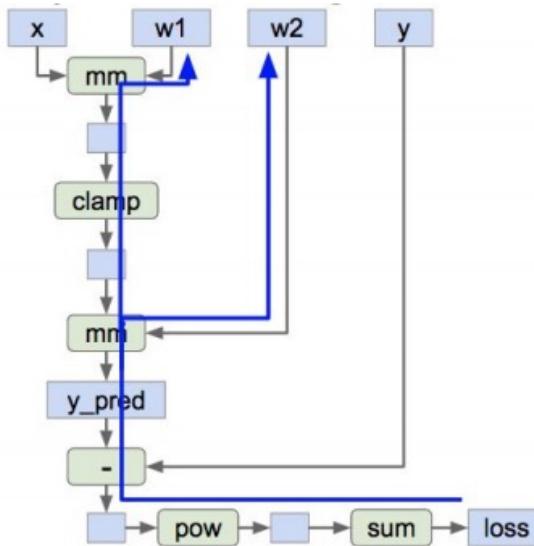
- All operations in the forward step are tracked for later gradient calculations.



```
N, D, H = 64, 1000, 100  
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)  
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)  
w1 = tf.Variable(tf.random.uniform((D, H))) # weights  
w2 = tf.Variable(tf.random.uniform((H, D))) # weights  
  
with tf.GradientTape() as tape:  
    h = tf.maximum(tf.matmul(x, w1), 0)  
    y_pred = tf.matmul(h, w2)  
    diff = y_pred - y  
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))  
    gradients = tape.gradient(loss, [w1, w2]).
```

Tensorflow 2.0: Neural Network

- `tape.gradient()` uses the previously tracked calculation graph to calculate the gradient.



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

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2])
```

Tensorflow 2.0: Neural Network

- Neural network training: loop over computational graphs, use gradients to update weights

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

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        h = tf.maximum(tf.matmul(x, w1), 0)
        y_pred = tf.matmul(h, w2)
        diff = y_pred - y
        loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
    gradients = tape.gradient(loss, [w1, w2])
    w1.assign(w1 - learning_rate * gradients[0])
    w2.assign(w2 - learning_rate * gradients[1])
```

Tensorflow 2.0: Neural Network

- The available optimization algorithm (optimizer) can be used to calculate the gradient and update the weights

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

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

optimizer = tf.optimizers.SGD(1e-6)

learning_rate = 1e-6
for t in range(50):
    with tf.GradientTape() as tape:
        h = tf.maximum(tf.matmul(x, w1), 0)
        y_pred = tf.matmul(h, w2)
        diff = y_pred - y
        loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
    gradients = tape.gradient(loss, [w1, w2])
    optimizer.apply_gradients(zip(gradients, [w1, w2])).
```

Tensorflow 2.0: Neural Network

- Can use predefined objective function

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

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H))) # weights
w2 = tf.Variable(tf.random.uniform((H, D))) # weights

optimizer = tf.optimizers.SGD(1e-6)

for t in range(50):
    with tf.GradientTape() as tape:
        h = tf.maximum(tf.matmul(x, w1), 0)
        y_pred = tf.matmul(h, w2)
        diff = y_pred - y
        loss = tf.losses.MeanSquaredError()(y_pred, y)
    gradients = tape.gradient(loss, [w1, w2])
    optimizer.apply_gradients(zip(gradients, [w1, w2]))
```

Keras: High-Level wrapper

- Keras is a layer on top of TensorFlow, makes common things easy to do (Used to be third-party, now merged into TensorFlow)

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

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

losses = []
for t in range(50):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = tf.losses.MeanSquaredError()(y_pred, y)
    gradients = tape.gradient(
        loss, model.trainable_variables)
    optimizer.apply_gradients(
        zip(gradients, model.trainable_variables))
```

Keras: High-Level wrapper

Define model as a sequence of layers

Get output by calling the model

Apply gradient to all trainable variables (weights) in the model

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

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
losses = []
for t in range(50):
    with tf.GradientTape() as tape:
        y_pred = model(x)
        loss = tf.losses.MeanSquaredError()(y_pred, y)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

Keras can handle the training loop for you!

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

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
model.compile(loss=tf.keras.losses.MeanSquaredError(),
              optimizer=optimizer)

history = model.fit(x, y, epochs=50, batch_size=N).
```



Tensorflow 2.0: @tf.function

- tf.function decorator (implicitly) compiles python functions to static graph for better performance
- Here we compare the forward-pass time of the same model under dynamic graph mode and static graph mode

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

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

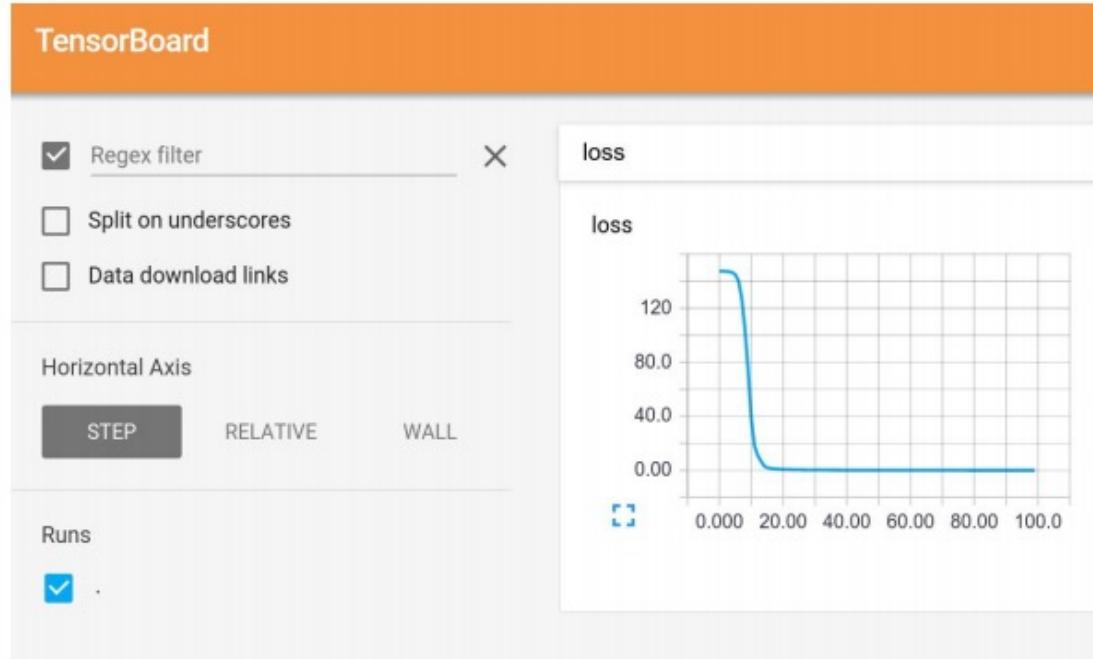
dynamic graph:  0.02520249200000535
static graph:  0.03932226699998864
```

TensorFlow: Pretrained Models

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

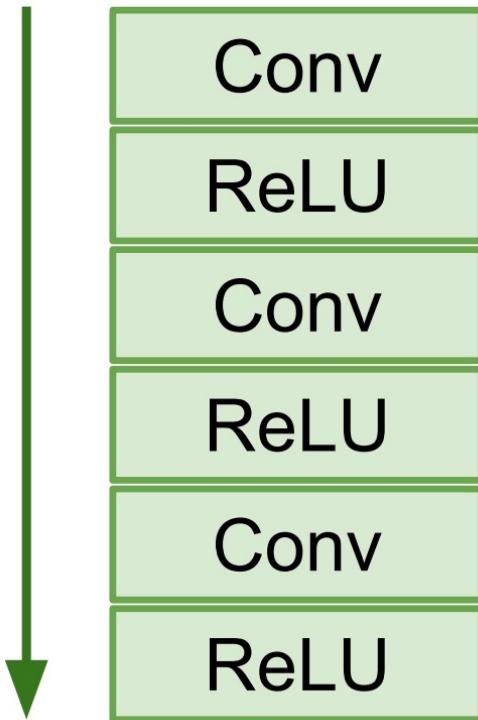
TensorFlow: Tensorboard

- Add log in the code to observe the objective function, parameters...
- Run server tensorboard and see the result

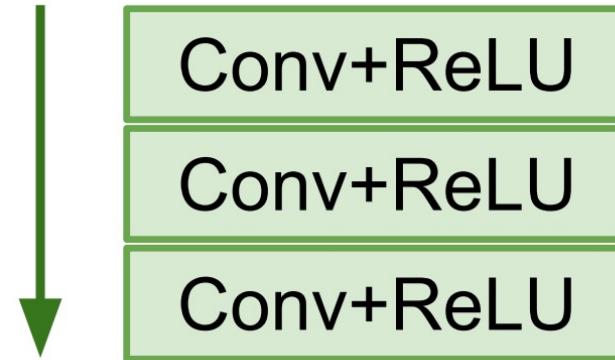


Static vs Dynamic

The graph you wrote



Equivalent graph with
fused operations



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

Static vs Dynamic: Optimization

- **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

PyTorch
Dynamic Graphs

TensorFlow

Pre-2.0: Default
Static Graph
2.0+: Default
Dynamic Graph

Static PyTorch

- Caffe2:

<https://caffe2.ai/>

- ONNX:

<https://github.com/onnx/onnx>

PyTorch

Dynamic Graphs

Static: ONNX, Caffe2

TensorFlow

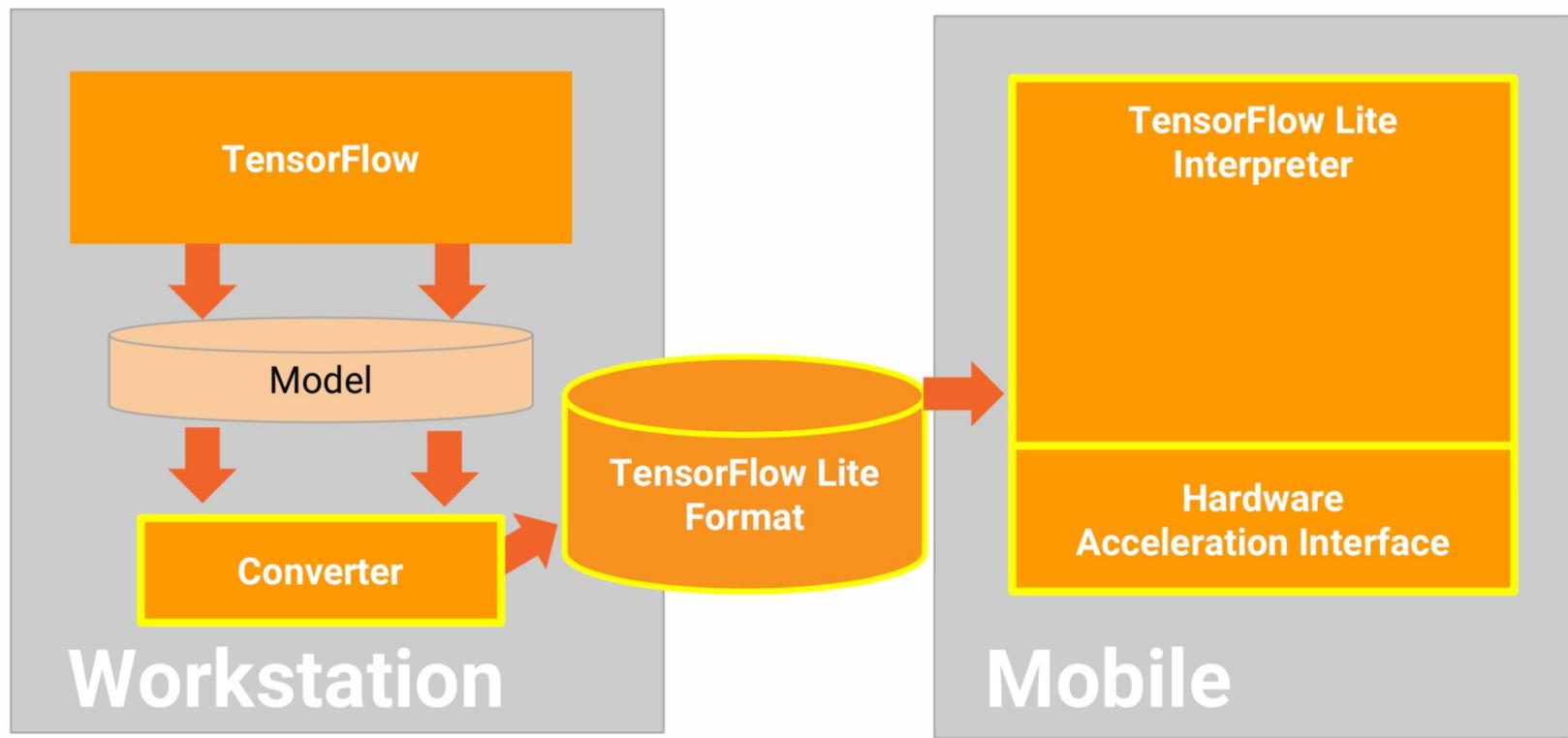
Dynamic: Eager

Static: @tf.function

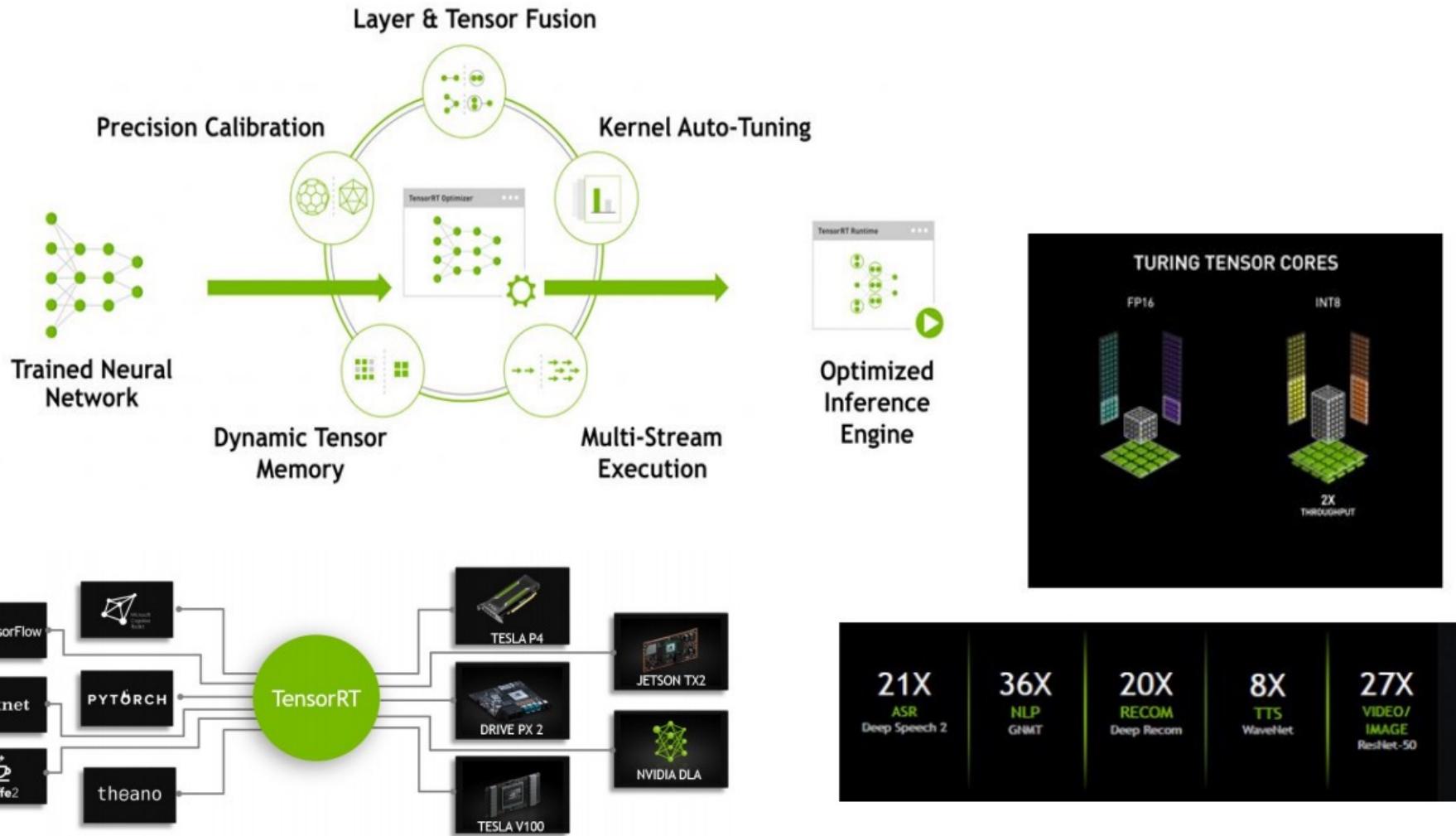
Accelerator and compression tools

Tensorflow Lite

- Tensorflow Lite is a set of tools to optimize Tensorflow models, make models more compact and infer faster on mobile platforms.

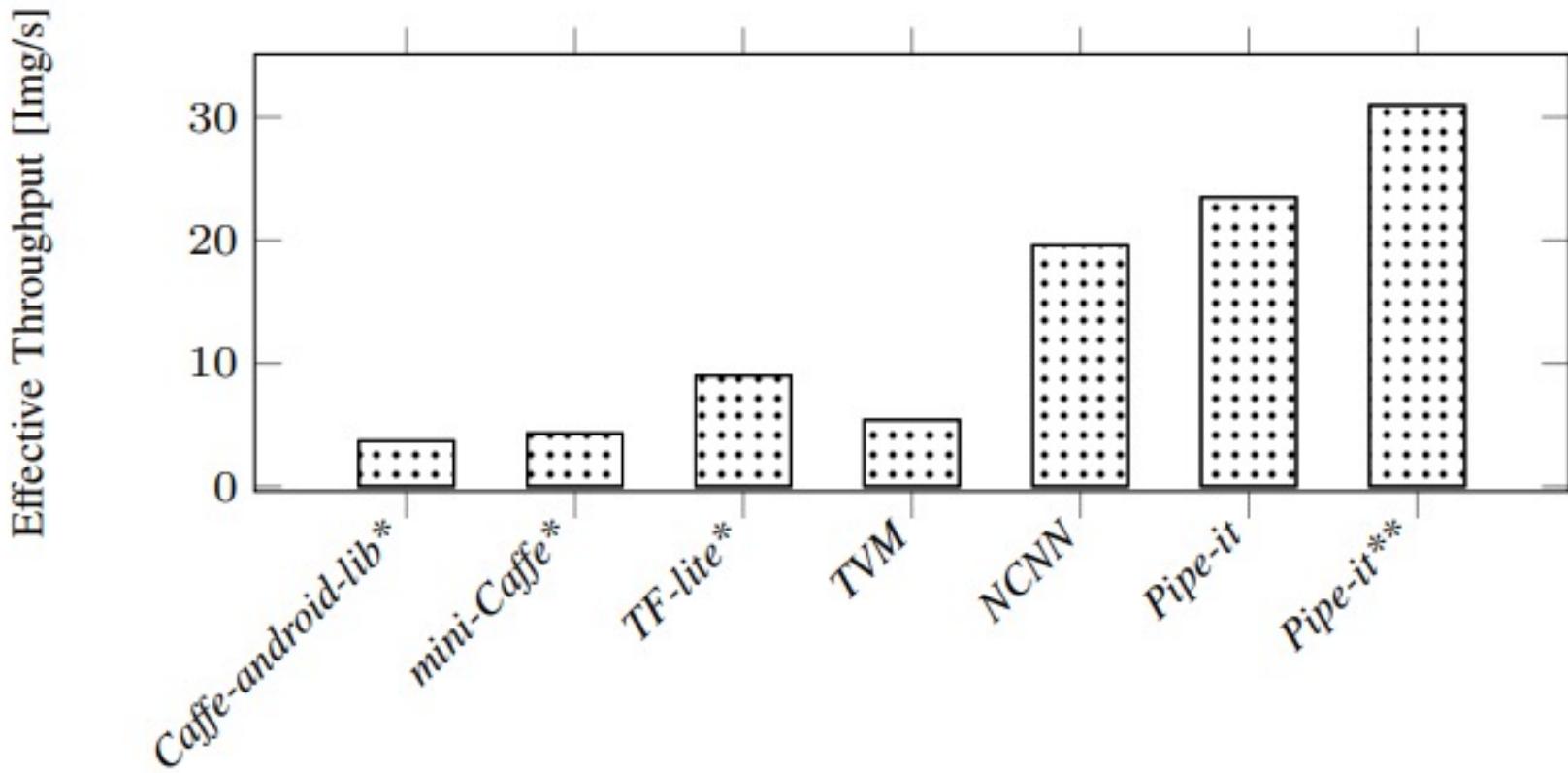


NVIDIA TensorRT



Other tools

- Pocket flow: <https://github.com/Tencent/PocketFlow>
- Tencent NCNN: <https://github.com/Tencent/ncnn>



References

1. The lecture is based on Stanford's cs231n

<http://cs231n.stanford.edu>

2. Tensorflow vs Keras vs PyTorch:

<https://databricks.com/session/a-tale-of-three-deep-learning-frameworks-tensorflow-keras-pytorch>

3. NVIDIA TensorRT:

[Fast Neural Network Inference with TensorRT on Autonomous](#)

4. ARM chip:

[Design And Reuse 2018 Keynote](#)



25
YEARS ANNIVERSARY
SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you
for your
attention!!!

