DEEP LEARNING

Lecture 4: Hardware and Software

Dr. Yang Lu

Department of Computer Science and Technology

luyang@xmu.edu.cn

HARDWARE

Computer Hardware

Image source: Lecture 6, cs231n

CPU vs GPU

- Central processing unit (CPU): Fewer cores, but each core is much faster and much more capable; great at sequential tasks.
- Graphics processing unit (GPU): More cores, but each core is much slower and "dumber"; great for parallel tasks.

	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

Image source: Lecture 6, cs231n

Nvidia vs. AMD

- Nvidia and AMD are two big GPU manufacturers.
  - Nvidia is founded in 1993 and focuses on GPU.
  - AMD is founded in 1969 and focuses on all kinds of chips.
- Now, deep learning related hardware is dominated by Nvidia.

Image source: https://www.gamingscan.com/nvidia-vs-amd/ https://cdn.wccftech.com/wp-content/uploads/2019/10/amd-lisa-su-nvidia-jensen-huang-feature.jpg https://memezila.com/saveimage/Nvidia-before-vs-Nvidia-now-meme-6521

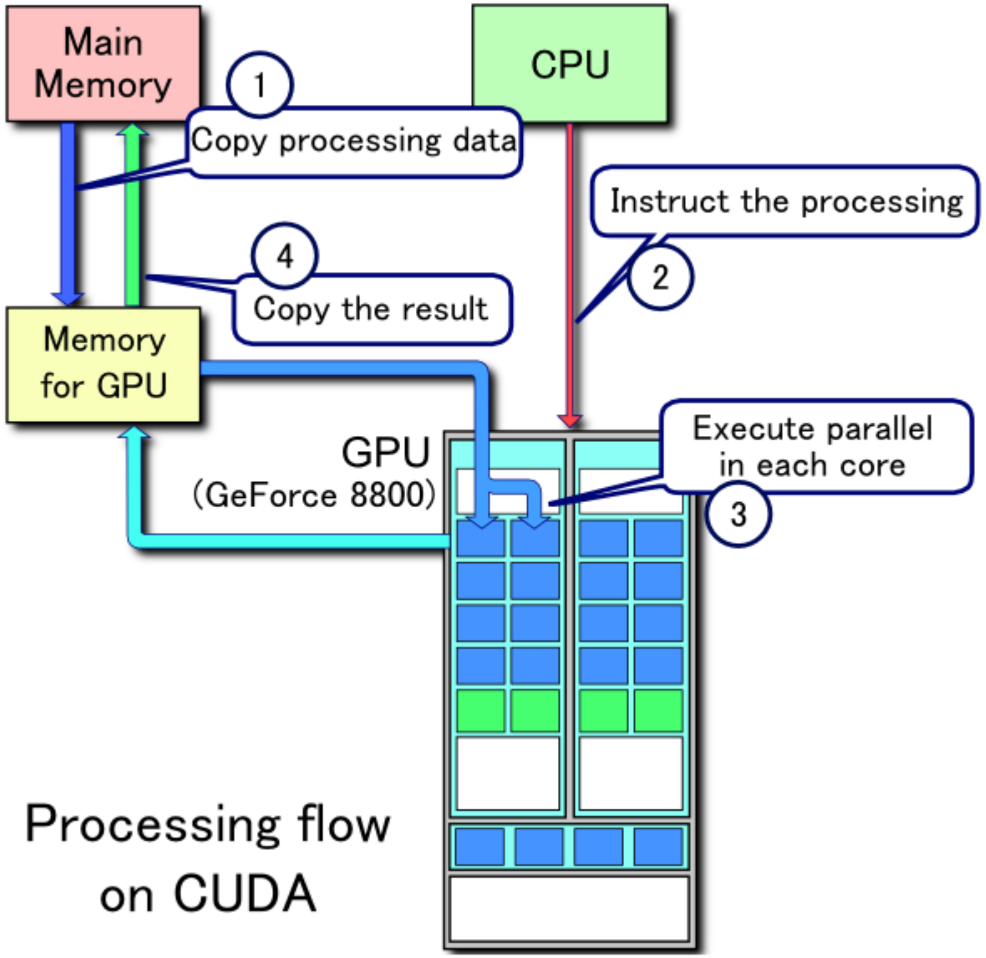
CUDA

- However, deep learning is not related to graphic computing at all. How to utilize GPU to train deep learning models?
- General-purpose computing on graphics processing units (GPGPU) is the use of GPU to perform computation in applications traditionally handled by CPU.
- Compute unified device architecture (CUDA) is an API for GPGPU with Nvidia GPUs.

Image source: https://en.wikipedia.org/wiki/CUDA

Example of CUDA Processing Flow

1. Copy data from main memory to GPU memory.
2. CPU initiates the GPU compute kernel.
3. GPU's CUDA cores execute the kernel in parallel.
4. Copy the resulting data from GPU memory to main memory.



CPU vs. GPU in Practice

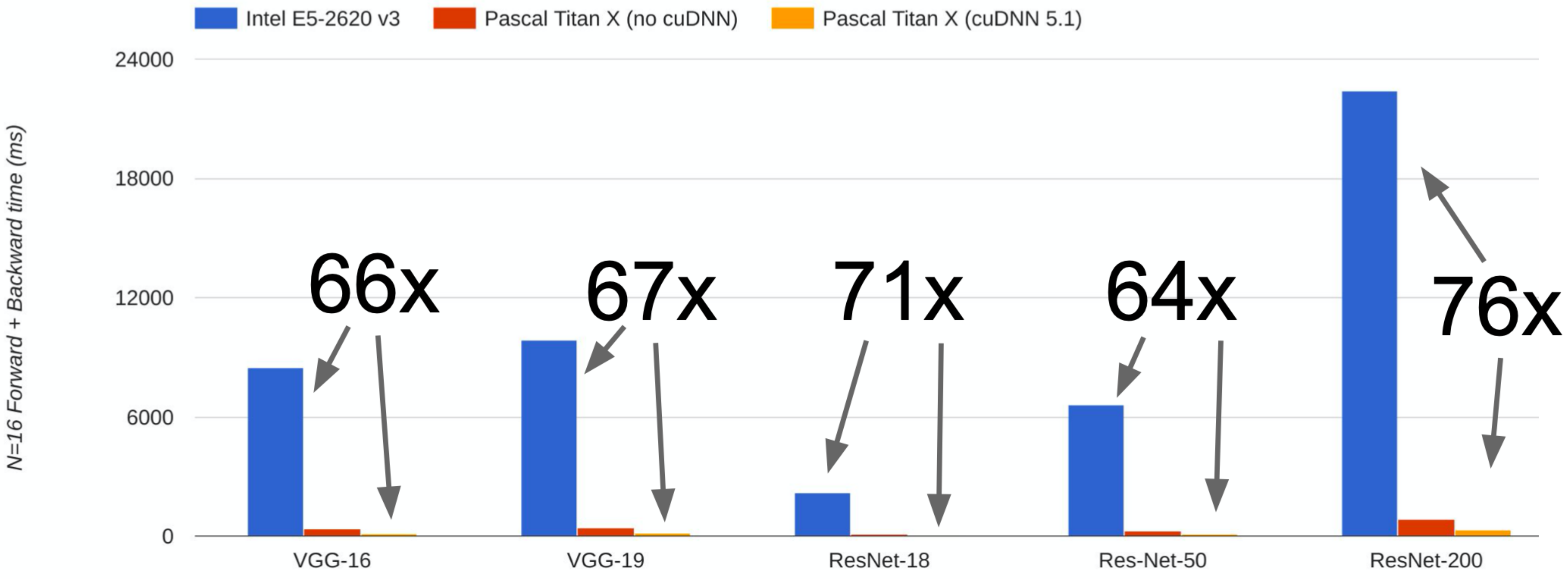


Image source: Lecture 6, cs231n

cuDNN

- CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks.
- cuDNN provides highly tuned implementations for standard routines.
  - Forward and backward convolution, pooling, normalization, activation layers, and so on.

Image source: https://insidehpc.com/2014/09/nvidia-cudnn-speeds-deep-learning-applications/?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+InsideHPC+%28insideHPC.com%29

cuDNN in Practice

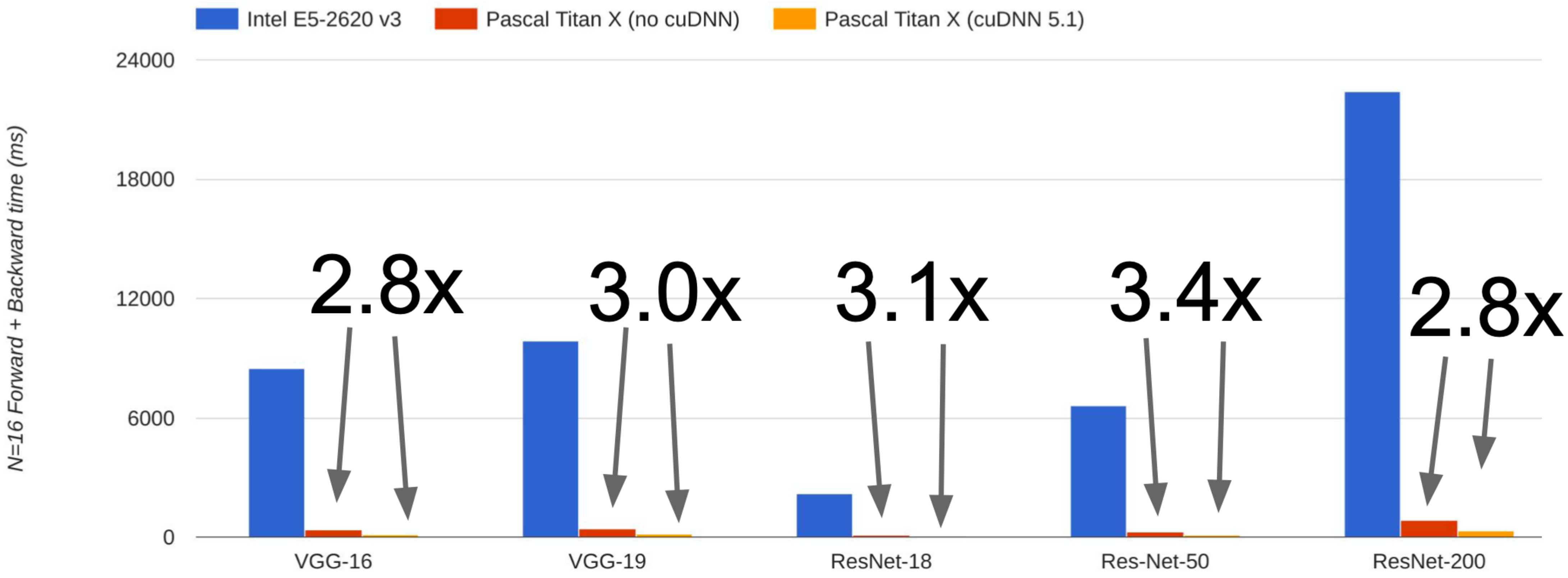
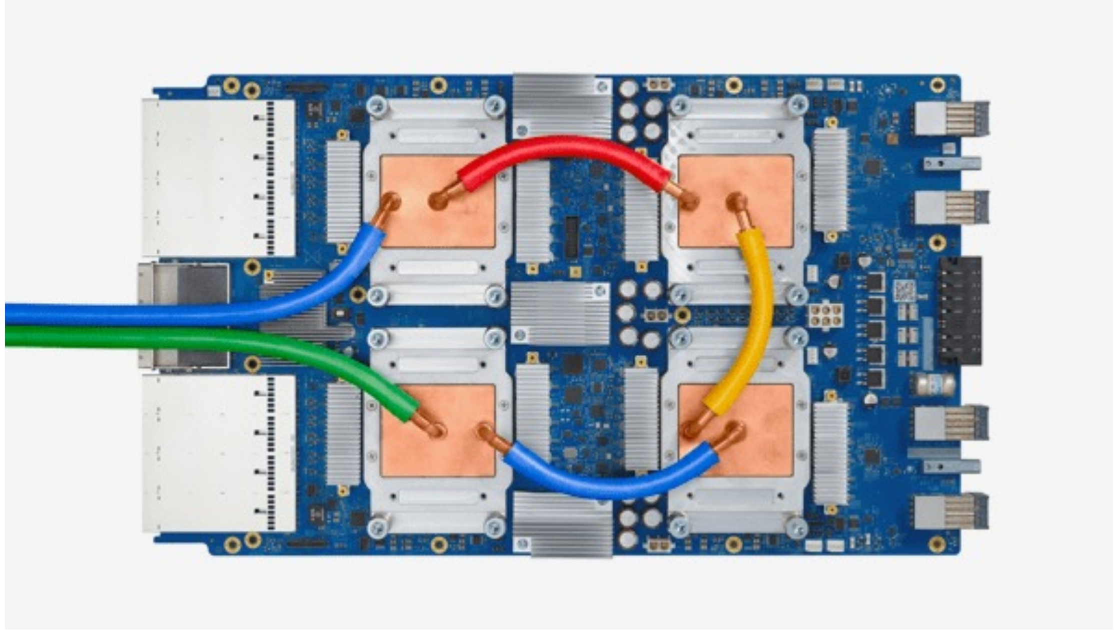


Image source: Lecture 6, cs231n

TPU

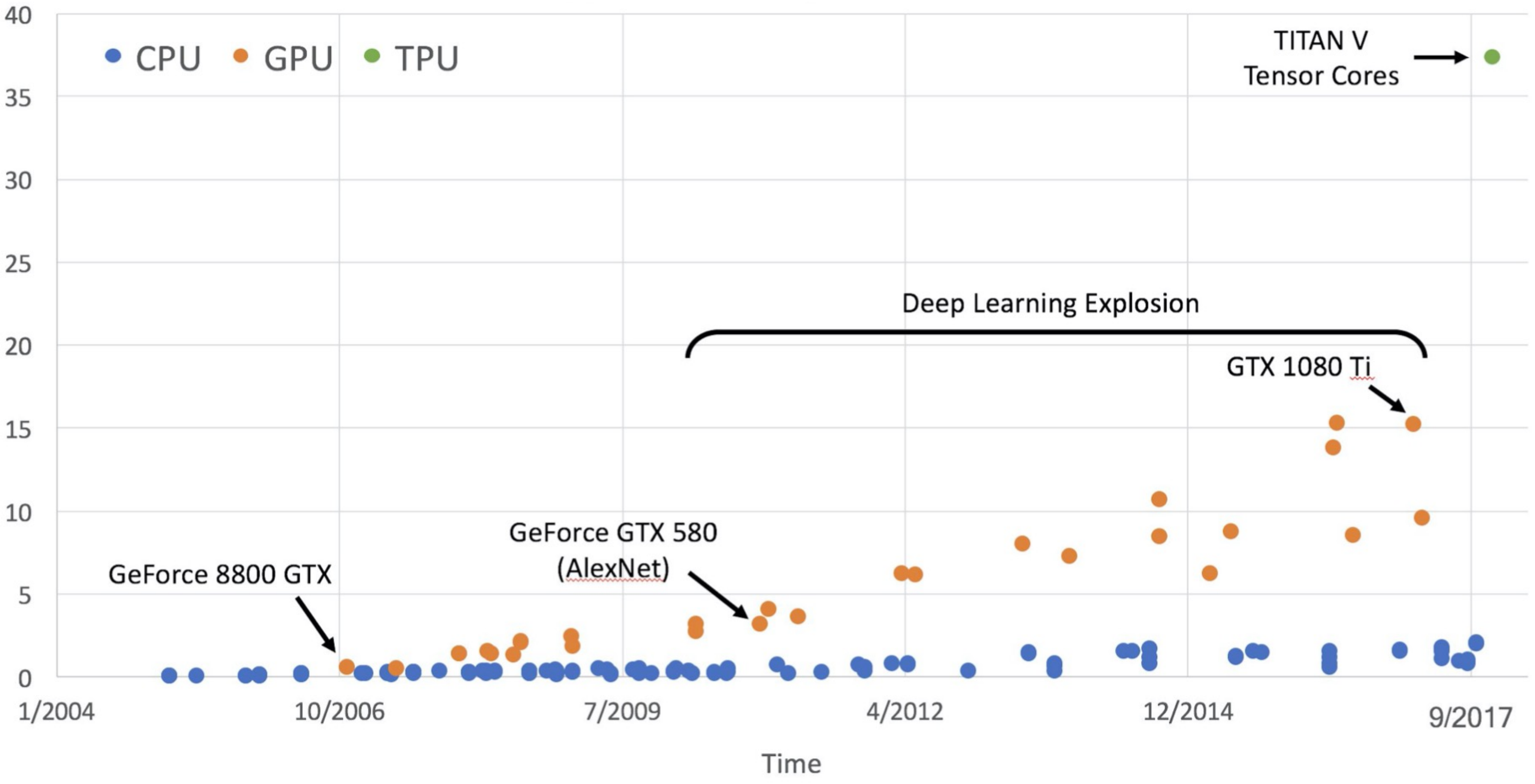
- Tensor Processing Unit (TPU) is an AI accelerator application-specific integrated circuit (ASIC) developed by Google.
  - Specifically for neural network machine learning.
  - Particularly for Google's own TensorFlow software.



Cloud TPU v3

Image source: https://cloud.google.com/images/products/tpu/cloud-tpu-v3-img_2x.png

Price of Computation Power



GigaFLOPs per Dollar

Image source: Lecture 6, cs231n

SOFTWARE

A Zoo of Frameworks

Caffe
(UC Berkeley)
→
Caffe2
(Facebook)
mostly features absorbed by PyTorch
↓

Torch
(NYU / Facebook)
→
PyTorch
(Facebook)

Theano
(U Montreal)
→
TensorFlow
(Google)

We'll focus on these

PaddlePaddle
(Baidu)

Chainer
(Preferred Networks)
The company has officially migrated its research infrastructure to PyTorch

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

CNTK
(Microsoft)

JAX
(Google)

And others...

Image source: Lecture 6, cs231n

Why Do We Need Frameworks

- Run it easily on GPU.
- Automatically compute gradients.
- Quick to develop and test new ideas.

Image source: https://img.devrant.com/devrant/rant/r_1397964_58Lc4.jpg

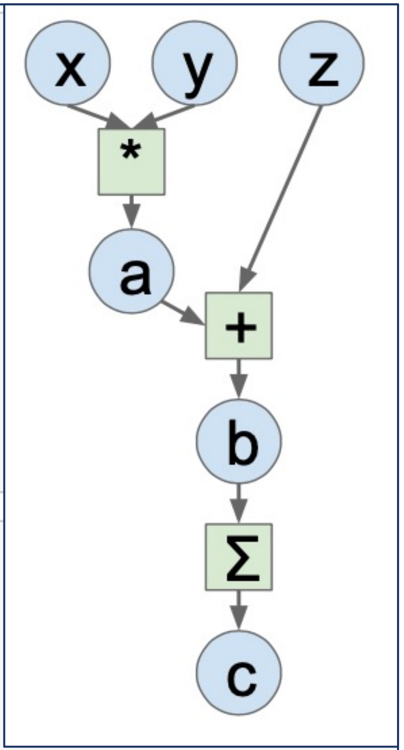
Example

Numpy

- Clean API, easy to write numeric code.
- However, we have to manually compute gradients and it can't be run on GPU.

```
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
```




```
grad_x
```

```
array([[ 0.76103773,  0.12167502,  0.44386323,  0.33367433],
       [ 1.49407907, -0.20515826,  0.3130677 , -0.85409574],
       [-2.55298982,  0.6536186 ,  0.8644362 , -0.74216502]])
```

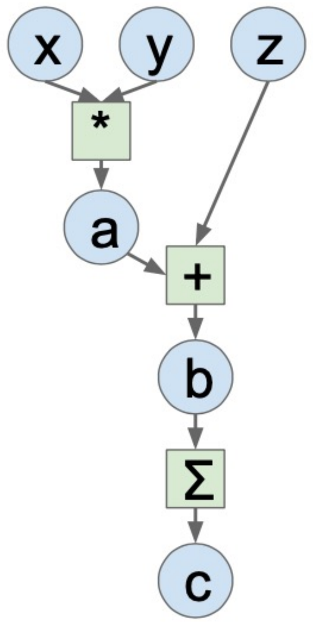
```
grad_y
```

```
array([[ 1.76405235,  0.40015721,  0.97873798,  2.2408932 ],
       [ 1.86755799, -0.97727788,  0.95008842, -0.15135721],
       [-0.10321885,  0.4105985 ,  0.14404357,  1.45427351]])
```

Example

PyTorch

- Looks exactly like numpy!
- PyTorch handles gradients for us!




```
import torch
from torch.autograd import Variable

x_torch = Variable(torch.from_numpy(x), requires_grad=True)
y_torch = Variable(torch.from_numpy(y), requires_grad=True)
z_torch = Variable(torch.from_numpy(z))

# or create new random number by:
# x_torch = torch.randn(N, D, requires_grad_True)
# y_torch = torch.randn(N, D)
# z_torch = torch.randn(N, D)

a_torch = x_torch * y_torch
b_torch = a_torch + z_torch
c_torch = torch.sum(b_torch)
c_torch.backward()
```

Only forward propagation is needed. Backpropagation is automatically calculated!

```
x_torch.grad
```

```
tensor([[ 0.7610,  0.1217,  0.4439,  0.3337],
        [ 1.4941, -0.2052,  0.3131, -0.8541],
        [-2.5530,  0.6536,  0.8644, -0.7422]], dtype=torch.float64)
```

```
y_torch.grad
```

```
tensor([[ 1.7641,  0.4002,  0.9787,  2.2409],
        [ 1.8676, -0.9773,  0.9501, -0.1514],
        [-0.1032,  0.4106,  0.1440,  1.4543]], dtype=torch.float64)
```

Example

- Extremely easy to move computation on GPU.
- All detailed implementations are hidden from the developers.

```
import torch
from torch.autograd import Variable

device = 'cuda:0'

# or create new random number by:
x_torch = torch.randn(N, D, requires_grad=True, device=device)
y_torch = torch.randn(N, D, requires_grad=True, device=device)
z_torch = torch.randn(N, D, device=device)

a_torch = x_torch * y_torch
b_torch = a_torch + z_torch
c_torch = torch.sum(b_torch)
c_torch.backward()
```

Example

Algorithm 1 Class-Balancing Reservoir Sampling

```
input: stream: {(x_i, y_i)}_{i=1}^n
for i = 1 to n do
  if memory is not filled then
    store (x_i, y_i)
  else
    if c ≡ y_i is not a full class then
      find all instances of the largest class
      select from them an instance at random
      overwrite the selected instance with (x_i, y_i)
    else
      m_c ← number of currently stored instances of
      class c ≡ y_i
      n_c ← number of stream instances of class c ≡
      y_i encountered thus far
      sample u ~ Uniform(0, 1)
      if u ≤ m_c/n_c then
        pick a stored instance of class c ≡ y_i at ran-
        dom
        replace it with (x_i, y_i)
      else
        ignore the instance (x_i, y_i)
      end if
    end if
  end if
end for
```

Old school pseudocode

Algorithm 1 Pseudocode of MoCo in a PyTorch-like style.

```
# f_q, f_k: encoder networks for query and key
# queue: dictionary as a queue of K keys (CxK)
# m: momentum
# t: temperature

f_k.params = f_q.params # initialize
for x in loader: # load a minibatch x with N samples
   x_q = aug(x) # a randomly augmented version
   x_k = aug(x) # another randomly augmented version

   q = f_q.forward(x_q) # queries: NxC
   k = f_k.forward(x_k) # keys: NxC
   k = k.detach() # no gradient to keys

   # positive logits: Nx1
   l_pos = bmm(q.view(N,1,C), k.view(N,C,1))

   # negative logits: NxK
   l_neg = mm(q.view(N,C), queue.view(C,K))

   # logits: Nx(1+K)
   logits = cat([l_pos, l_neg], dim=1)

   # contrastive loss, Eqn.(1)
   labels = zeros(N) # positives are the 0-th
   loss = CrossEntropyLoss(logits/t, labels)

   # SGD update: query network
   loss.backward()
   update(f_q.params)

   # momentum update: key network
   f_k.params = m*f_k.params+(1-m)*f_q.params

   # update dictionary
   enqueue(queue, k) # enqueue the current minibatch
   dequeue(queue) # dequeue the earliest minibatch
```

New school pseudocode

Deep Learning Frameworks

Image source: Hung-yi Lee, Understanding Deep Learning in One Day

SOFTWARE

PYTORCH

PyTorch

- For this class we are using **PyTorch version ≥ 1.12**.
- Major API change in release 1.0.
- Be careful if you are looking at older PyTorch code (<1.0)

Fundamental Concepts

- Tensor: A PyTorch Tensor is conceptually identical to a numpy array: an n-dimensional array.
- It is a complicated data structure, rather than a simple array.
  - PyTorch provides many functions for operating on these Tensors.
  - Tensors can keep track of a computational graph and gradients.
  - Also unlike numpy, PyTorch Tensors can utilize GPUs to accelerate their numeric computations.

PyTorch: Tensors

- Running example: Train a two-layer ReLU network on random data with L^2 loss.
  - Create random tensors for data and weights.
  - Forward pass: compute predictions and loss.
  - Backward pass: manually compute gradients.
  - Update weights using gradient descent.

```python
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().item()

    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
```


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: Tensors

- Check official documentation for data type or function usage.

`Tensor.mm(mat2)` → Tensor

See `torch.mm()`

`torch.mm(input, mat2, out=None)` → Tensor

Performs a matrix multiplication of the matrices `input` and `mat2`.

If `input` is a $(n \times m)$ tensor, `mat2` is a $(m \times p)$ tensor, `out` will be a $(n \times p)$ tensor.

• NOTE

This function does not broadcast. For broadcasting matrix products, see `torch.matmul()`.

Parameters

- **input** (*Tensor*) – the first matrix to be multiplied
- **mat2** (*Tensor*) – the second matrix to be multiplied
- **out** (*Tensor, optional*) – the output tensor.

```
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().item()

    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
```


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: Tensors

- Check official documentation for data type or function usage.

`Tensor.clamp`(*min=None*, *max=None*) → Tensor

See `torch.clamp()`

`torch.clamp`(*input*, *min*, *max*, *out=None*) → Tensor

Clamp all elements in `input` into the range [`min`, `max`] and return a resulting tensor:

$$y_i = \begin{cases} \min & \text{if } x_i < \min \\ x_i & \text{if } \min \leq x_i \leq \max \\ \max & \text{if } x_i > \max \end{cases}$$

If `input` is of type *FloatTensor* or *DoubleTensor*, args `min` and `max` must be real numbers, otherwise they should be integers.

Parameters

- **input** (*Tensor*) – the input tensor.
- **min** (*Number*) – lower-bound of the range to be clamped to
- **max** (*Number*) – upper-bound of the range to be clamped to
- **out** (*Tensor, optional*) – the output tensor.

```python
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().item()

    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
```


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: Tensors

- After running, the values of the Tensors can be directly displayed in Jupyter Notebook.

```
print(x)

tensor([[ 0.1167, -0.1729, -0.9560,  ...,  0.3038,  0.0946, -0.0566],
        [-1.7715,  0.7274, -2.4168,  ..., -0.3387,  0.0713,  0.0496],
        [-1.0656, -0.0815,  0.6822,  ..., -0.7126,  0.1126, -0.6691],
        ...,
        [ 0.0553, -2.1470,  0.4969,  ...,  0.9881,  0.7723,  1.0676],
        [ 0.1760, -0.4440, -1.0164,  ...,  1.2869,  0.6865,  1.3140],
        [ 1.0921,  0.4093,  1.2611,  ..., -0.8258, -0.8017,  0.7185]])

print(grad_w1)

tensor([[ 0.0043,  0.0071, -0.0060,  ...,  0.0038, -0.0089,  0.0018],
        [-0.0043,  0.0165,  0.0086,  ..., -0.0212,  0.0056,  0.0024],
        [ 0.0035,  0.0118, -0.0061,  ...,  0.0071,  0.0043, -0.0070],
        ...,
        [ 0.0122, -0.0208, -0.0063,  ...,  0.0184, -0.0080,  0.0107],
        [-0.0020,  0.0171,  0.0098,  ...,  0.0004,  0.0052, -0.0059],
        [-0.0049, -0.0157,  0.0218,  ..., -0.0054,  0.0064,  0.0049]])
```

PyTorch: Autograd

- In the above examples, we had to manually implement both the forward and backward passes of our neural network.
  - It quickly gets very hairy for large complex networks.
- How can we automatically calculate the gradient?
- The autograd package in PyTorch provides exactly this functionality.
  - When using autograd, the forward pass of your network will define a computational graph (typically a DAG).
  - Nodes in the graph will be Tensors, and edges will be functions.
  - Autograd automatically compute gradients through this graph.

PyTorch: Autograd

```python
import torch

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

- Creating Tensors with `requires_grad=True` enables autograd.
- It causes PyTorch to build a computational graph.

PyTorch: Autograd

- We don't want gradients (of loss) with respect to data.
- We want gradients with respect to weights.
- We don't need to track intermediate values. PyTorch keeps track of them for us in the graph.

```
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: Autograd

- Automatically compute gradient of loss with respect to `w1` and `w2`.
- `torch.no_grad()` means "don't build a computational graph for this part".
- Simply call `.grad` to get the gradients.

```python
import torch

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

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

    loss.backward()

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


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: Autograd

- Here, each loop in `for t in range(500)` is a epoch, not a minibatch.
- We need to reset the gradient to zero after each epoch. Otherwise, `backward()` will accumulate the gradient.
- This is done by `zeros_()`.
  - PyTorch methods that end in underscore modify the Tensor in-place, which don't return a new Tensor.

```python
import torch

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

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

    loss.backward()

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


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: Defining New Autograd Functions

- Define your own autograd functions by writing `forward` and `backward` functions for Tensors.
- `ctx` is a context object that can be used to stash information for backward computation.
- Define a helper function to make it easy to use the new function.

```python
import torch
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)
```


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: Defining New Autograd Functions

```python
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 = my_relu(x.mm(w1)).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_()
```

```python
import torch
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_y):
        x, = ctx.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input

def my_relu(x):
    return MyReLU.apply(x)
```


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: nn

- Although computational graphs and autograd are a very powerful paradigm, building models is still not as easy as playing LEGO.
  - Forward propagation is still hand-made.
- We frequently think of arranging the neural networks into layers.
- In PyTorch, the `nn` package serves this same purpose. The `nn` package defines a set of Modules, which are roughly equivalent to neural network layers.
- Use this. It will make your life easier!

PyTorch: nn

- Define our model as a sequence of layers. Each layer is an object that holds learnable weights.
- Use `torch.nn.MSELoss` to define our loss.
- Make gradient step on each model parameter.
- All detailed computations are hidden by high-level wrappers.

```python
import torch

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


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

loss_fn = torch.nn.MSELoss()

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()

    loss.backward()

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

PyTorch: nn

CLASS `torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

This module supports TensorFloat32.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Source: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

PyTorch: nn

CLASS `torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')` [SOURCE]

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{`mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{`sum'.} \end{cases}$$

x and y are tensors of arbitrary shapes with a total of n elements each.

The mean operation still operates over all the elements, and divides by n.

The division by n can be avoided if one sets `reduction = 'sum'`.

PyTorch: optim

- Up to this point we have updated the weights of our models by manually mutating the learnable parameters.
- The `optim` package provides implementations of commonly used optimization algorithms.
  - AdaGrad, RMSProp, Adam…

PyTorch: optim

- Use Adam as the optimizer. The first argument tells the optimizer which Tensors it should update.
- Use the optimizer object to zero all of the gradients for the variables it will update.
- Calling the step function on an Optimizer makes an update to its parameters.

```python
import torch

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

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

loss_fn = torch.nn.MSELoss()

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

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()

    loss.backward()

    optimizer.step()
```


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: Custom nn Modules

- Sometimes you will want to specify models that are more complex than a sequence of existing Modules.
- You can define your own Modules by subclassing `nn.Module`.
- Simply define a `forward` function which receives input Tensors and produces output Tensors using other modules or other autograd operations on Tensors.

PyTorch: Custom nn Modules

- Subclass of nn.Module.
- Define modules containing parameters in the constructor as member variables.
- Define operators in forward function, whose input and output are both a Tensor.
- Instantiate model and directly pass data into it.

```python
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)
loss_fn = torch.nn.MSELoss()

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

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


Code source: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

PyTorch: Custom nn Modules

- Very common to mix and match custom Module subclasses and Sequential containers.
- Stack multiple instances of the component in a sequential.

```
import torch

class ParallelBlock(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(ParallelBlock, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)
        self.linear2 = torch.nn.Linear(D_in, D_out)
    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return(h1 * h2).clamp(min=0)

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(
    ParallelBlock(D_in, H),
    ParallelBlock(H, H),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```


Image source: Lecture 6, cs231n

PyTorch: Dataset & Dataloaders

- PyTorch provides two data primitives that allow you to use pre-loaded datasets as well as your own data:
  - torch.utils.data.Dataset: store the samples and their corresponding labels.
  - torch.utils.data.DataLoader: wrap an iterable around the Dataset to enable easy access to the samples.

```
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True,download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=batch_size,shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(
    root='./data', train=False,download=True, transform=transform)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=batch_size,shuffle=False, num_workers=2)
```


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

PyTorch: Dataset & Dataloaders

- We simply have to loop over our data iterator, and feed the inputs to the network and optimize.

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
```


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

PyTorch: Dataset & Dataloaders

- When doing testing, the progress is same except that we don’t need to calculate the loss and do backpropagation.

```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

PyTorch: Training and Evaluation Mode

`model.train()`

`train(mode=True)` [SOURCE]

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

`model.eval()`

`eval()` [SOURCE]

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

PyTorch: Saving and Loading Models

- In PyTorch, the learnable parameters (i.e. weights and biases) of an `torch.nn.Module` model are contained in the model's parameters (accessed with model.parameters()).
- A `state_dict` is simply a Python dictionary object that maps each layer to its parameter tensor.
- To save a model, we prefer to save its learned parameters, rather than the whole model.

PyTorch: Saving and Loading Models

```python
import torch.nn as nn
import torch.optim as optim

# Define model
class TheModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```python
# Initialize model
model = TheModelClass()

# Print model's state_dict
print("Model's state_dict:")
for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())
```

```
Model's state_dict:
conv1.weight     torch.Size([6, 3, 5, 5])
conv1.bias       torch.Size([6])
conv2.weight     torch.Size([16, 6, 5, 5])
conv2.bias       torch.Size([16])
fc1.weight       torch.Size([120, 400])
fc1.bias         torch.Size([120])
fc2.weight       torch.Size([84, 120])
fc2.bias         torch.Size([84])
fc3.weight       torch.Size([10, 84])
fc3.bias         torch.Size([10])
```

PyTorch: Saving and Loading Models

- Two ways to saving and loading models:

```
# save model parameters
torch.save(model.state_dict(), PATH)

# initialize a new model and load parameters into it
model = TheModelClass()
model.load_state_dict(torch.load(PATH))
```

Recommended in official documentation

```
# save the whole model
torch.save(model, PATH)

# load the whole model
model = torch.load(PATH)
```

Used as black box inference model

TensorBoard

- TensorBoard is developed by TensorFlow as a visualization tool.
- PyTorch also support TensorBoard as a package `torch.utils.tensorboard`.
- It lets you log PyTorch models and metrics into a directory for visualization within the TensorBoard UI.
  - Scalars, images, histograms, graphs, and embedding visualizations are all supported for PyTorch models and tensors.

TensorBoard

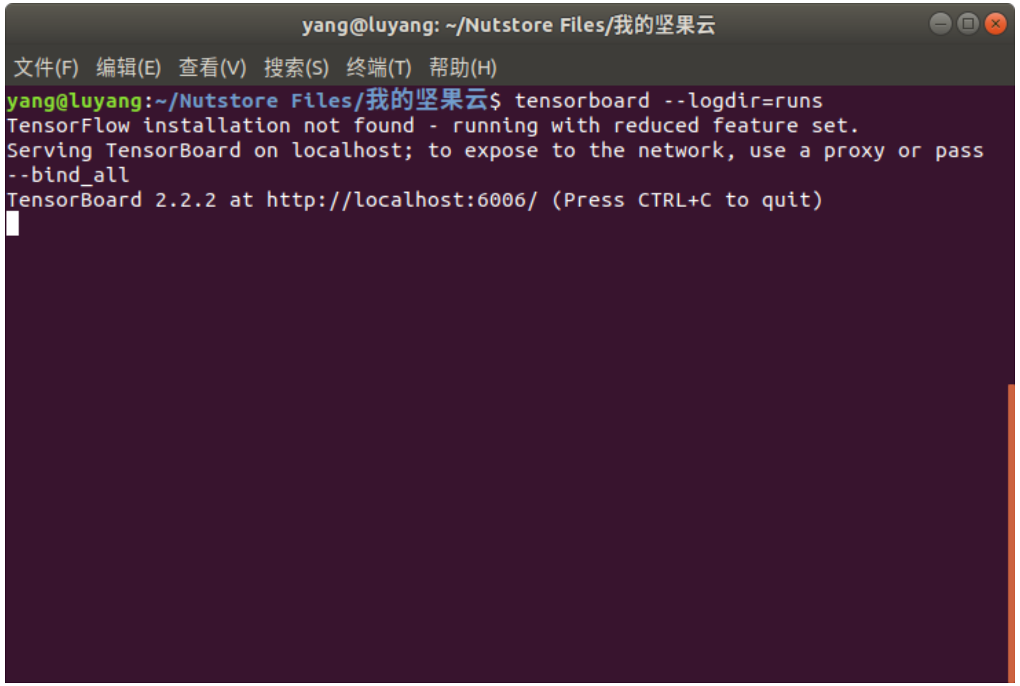
- Create a writer object with log path for tensorboard.
- Add the model and its input for computational graph visualization.
- Add the scalars that you want to track.

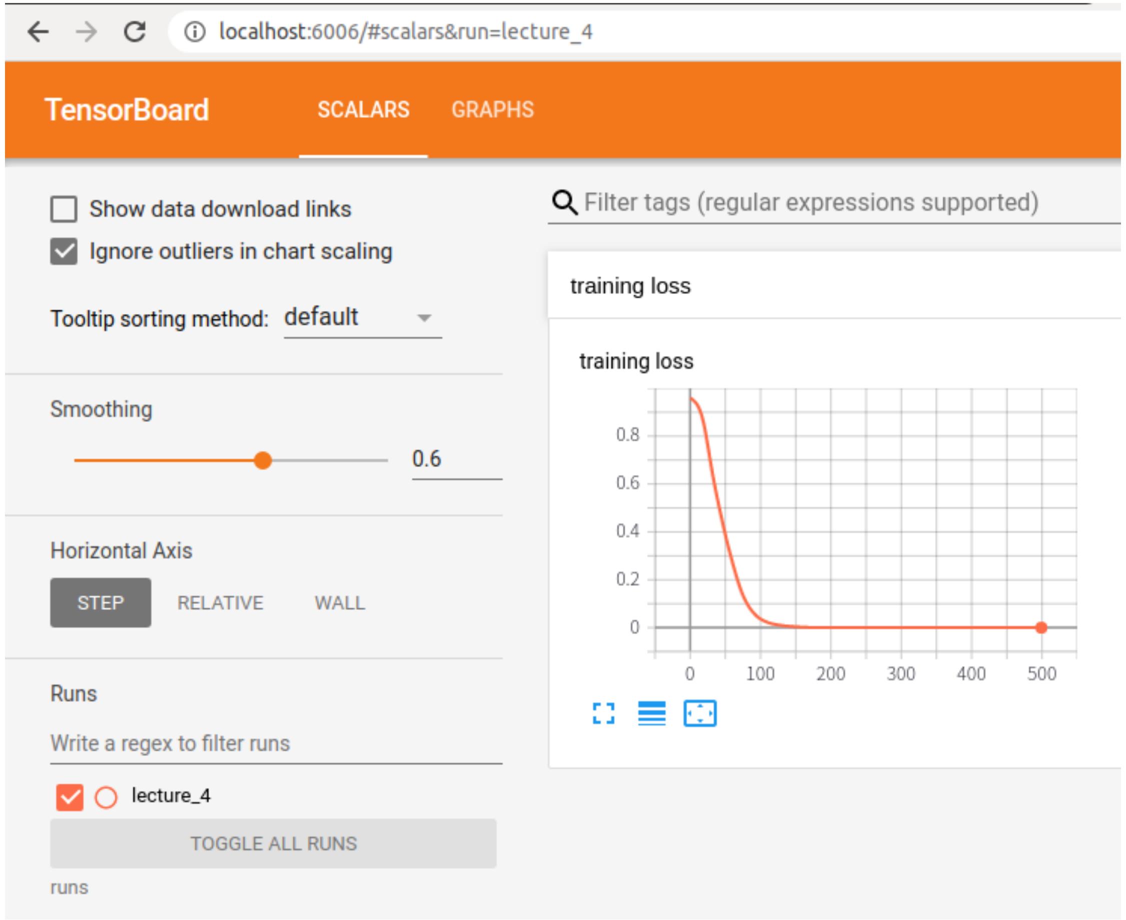
```
from torch.utils.tensorboard import SummaryWriter
import torch

writer = SummaryWriter('runs/lecture_4')

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(
    ParallelBlock(D_in, H),
    ParallelBlock(H, H),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss()

writer.add_graph(model, x)

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

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    writer.add_scalar('training loss', loss, t)

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

TensorBoard

- Run TensorBoard on terminal：

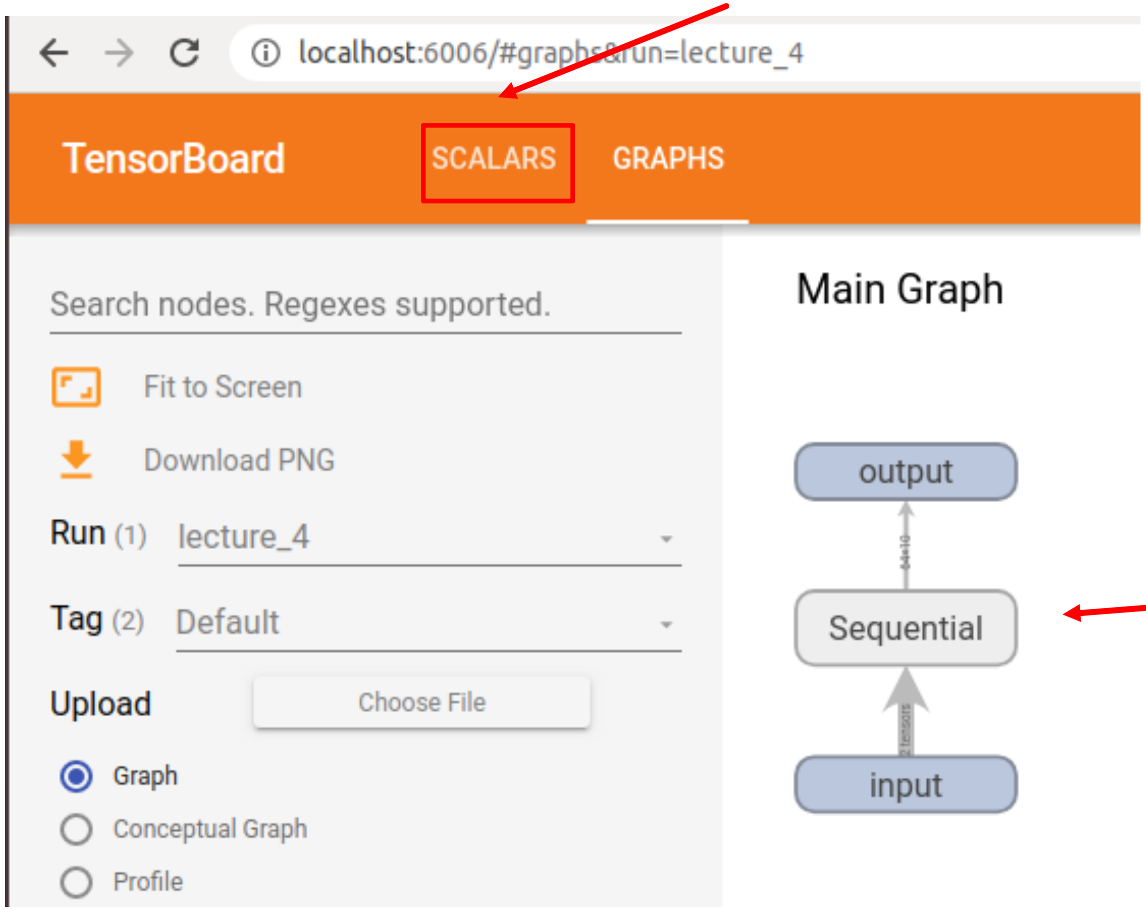
```
tensorboard --logdir=runs
```



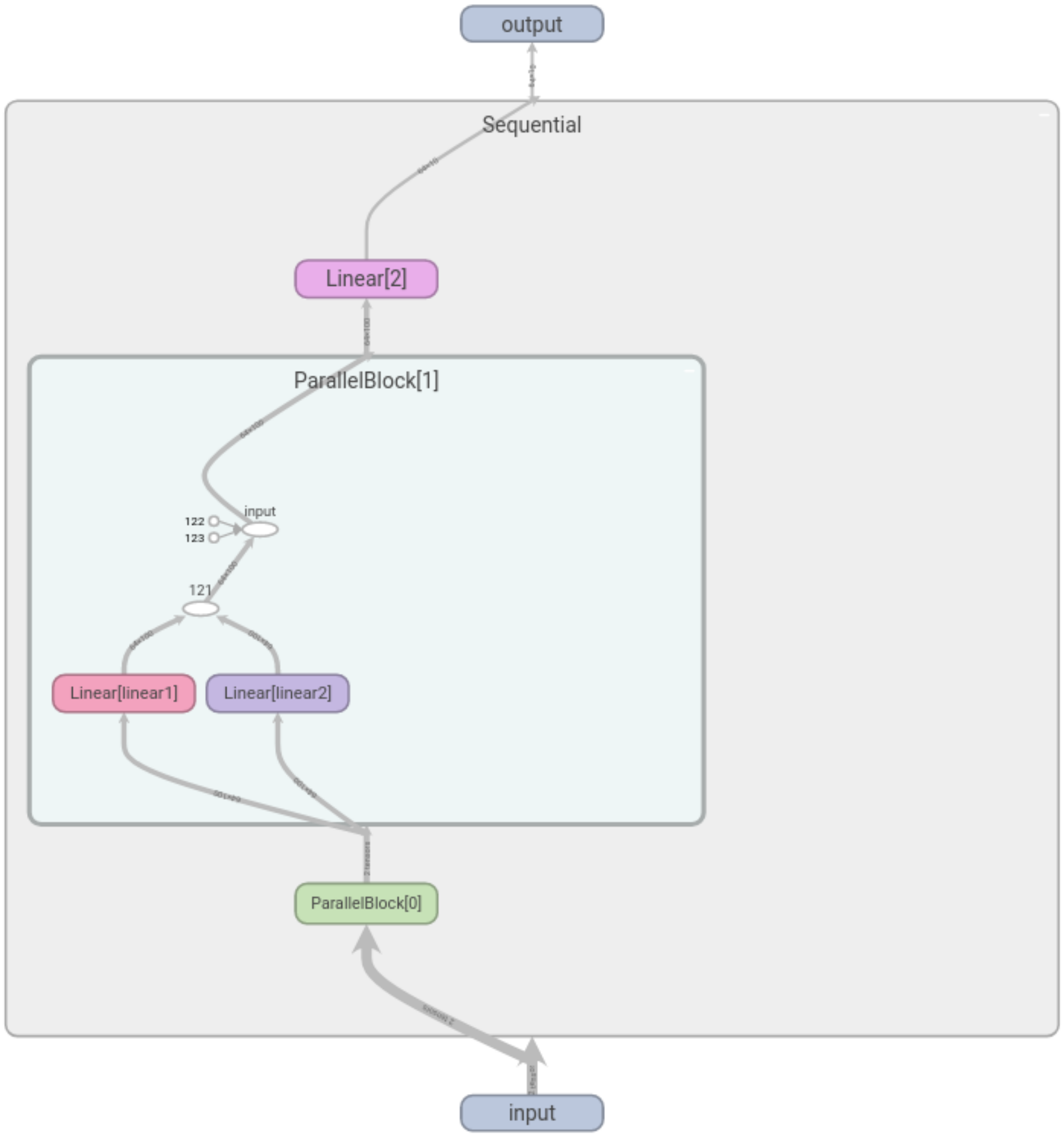


TensorBoard

Check your computational graph here



Double click to expand


output
Sequential
Linear[2]
ParallelBlock[1]
input
122
123
121
Linear[linear1]
Linear[linear2]
ParallelBlock[0]
input

SOFTWARE

TENSORFLOW

Dynamic Graph

- The computational graph is built up dynamically, immediately after we declare variables.
- This graph is thus rebuilt after each iteration of training.
- Dynamic graphs are flexible and allow us modify and inspect the internals of the graph at any time. The main drawback is that it can take time to rebuild the graph.

Static Graph vs. Dynamic Graph

Static graph

- Create and connect all the variables at the beginning, and initialize them into a static (unchanging) session.
- This session and graph persists and is reused: it is not rebuilt after each iteration of training, making it efficient.
- However, with a static graph, variable sizes have to be defined at the beginning, which can be non-convenient for some applications, such as NLP with variable length inputs.

Static Graph vs. Dynamic Graph

- In static graph, the graph is built first, and then the data is fed into it.
  - TensorFlow Pre-2.0 version.
- In dynamic graph, the graph and the data are processed at the same time.
  - PyTorch, TensorFlow Eager mode, TensorFlow 2.0+.

Image source: https://www.facebook.com/convolutionalmemes/photos/long-live-pytorch-death-to-tfthanks-martin-for-sending-this-in-henry/549971675457337/

TensorFlow Versions

- Pre-2.0 (1.14 latest): Default static graph, optionally dynamic graph (eager mode).
- 2.0+: Default dynamic graph, optionally static graph.
  - We use 2.3 (July 2020) in this class.

TensorFlow: Neural Net (Pre-2.0)

- We are using TensorFlow 2.3. If we want to use the funtions in 1.x, we should specify it.
- Define variable shape.
  - No real data is here. That's why it is called "placeholder".
- Define computational graph.
- Run the graph with data.

```python
import numpy as np
# import tensorflow as tf
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

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
```


Code source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/neural-network/tensorflow_basics

TensorFlow: 2.0+ vs. Pre-2.0

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

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)))
w2 = tf.Variable(tf.random.uniform((H, D)))

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])
```

```python
import numpy as np
# import tensorflow as tf
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

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
```


Code source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/neural-network/tensorflow_basics

TensorFlow: 2.0+ vs. Pre-2.0

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

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)))
w2 = tf.Variable(tf.random.uniform((H, D)))

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])
```

```python
import numpy as np
# import tensorflow as tf
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

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
```


Code source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/neural-network/tensorflow_basics

TensorFlow: Neural Net

- Convert input numpy arrays to TF tensors. Create weights as `tf.Variable`.
- Use `tf.GradientTape()` context to build dynamic computation graph.
  - All forward-pass operations in the contexts gets traced for computing gradient later.
- `tape.gradient()` uses the traced computation graph to compute gradient for the weights.

```
import numpy as np
import tensorflow as tf

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)))
w2 = tf.Variable(tf.random.uniform((H, D)))

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])
```


Image source: Lecture 6, cs231n

TensorFlow: Neural Net

```
In [6]: len(gradients)

Out[6]: 2

In [8]: gradients[0]

Out[8]: <tf.Tensor: shape=(1000, 100), dtype=float32, numpy=
        array([[ 77136.25   ,  50520.47   ,  46941.473  , ...,  77187.93   ,
                 76155.44   ,  65265.594  ],
               [ -7570.9473 ,  -2315.841  ,    967.99536, ..., -20049.777  ,
                -22515.719  ,  -8767.129  ],
               [ 98054.98   , 118292.88   , 138639.16   , ...,  95294.66   ,
                111306.97   , 121170.39   ],
               ...,
               [ 43516.918  ,  86071.49   ,  74825.086  , ...,  70804.06   ,
                 58628.035  ,  68784.42   ],
               [ 84807.055  ,  61783.2    ,  98520.46   , ...,  88406.66   ,
                 99416.92   ,  81866.72   ],
               [149381.47   , 142693.42   , 133367.5    , ..., 127278.83   ,
                108648.07   , 130923.3    ]], dtype=float32)>
```

TensorFlow: Neural Net

- Train the network: Run the training step over and over, use gradient to update weights.

```
import numpy as np
import tensorflow as tf

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)))
w2 = tf.Variable(tf.random.uniform((H, D)))

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])
```


Image source: Lecture 6, cs231n

TensorFlow: Optimizer and Loss

- Use an optimizer to compute gradients and update weights.
- Use predefined common losses.

```
import numpy as np
import tensorflow as tf

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)))
w2 = tf.Variable(tf.random.uniform((H, D)))

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.losses.MeanSquaredError()(y_pred, y)
    gradients = tape.gradient(loss, [w1, w2])
    optimizer.apply_gradients(zip(gradients, [w1, w2]))
```


Image source: Lecture 6, cs231n

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.
  - It is the recommended high-level API for TensorFlow 2.0.

Keras: High-Level Wrapper

Define model as a sequence of layers.

```
import numpy as np
import tensorflow as tf

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))
```

Get output by calling the model.

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

Image source: Lecture 6, cs231n

Keras: High-Level Wrapper

```
import numpy as np
import tensorflow as tf

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, steps_per_epoch=1, epochs=50, batch_size=N)
```

Keras can handle the training loop for you!

TensorFlow: High-Level Wrappers

- Keras (https://keras.io/)
- tf.keras (https://www.tensorflow.org/api_docs/python/tf/keras)
- tf.estimator (https://www.tensorflow.org/api_docs/python/tf/estimator)
- Sonnet (https://github.com/deepmind/sonnet)
- TFLearn (http://tflearn.org/)
- TensorLayer (https://tensorlayer.readthedocs.io/en/latest/)

PyTorch vs. TensorFlow: Academia

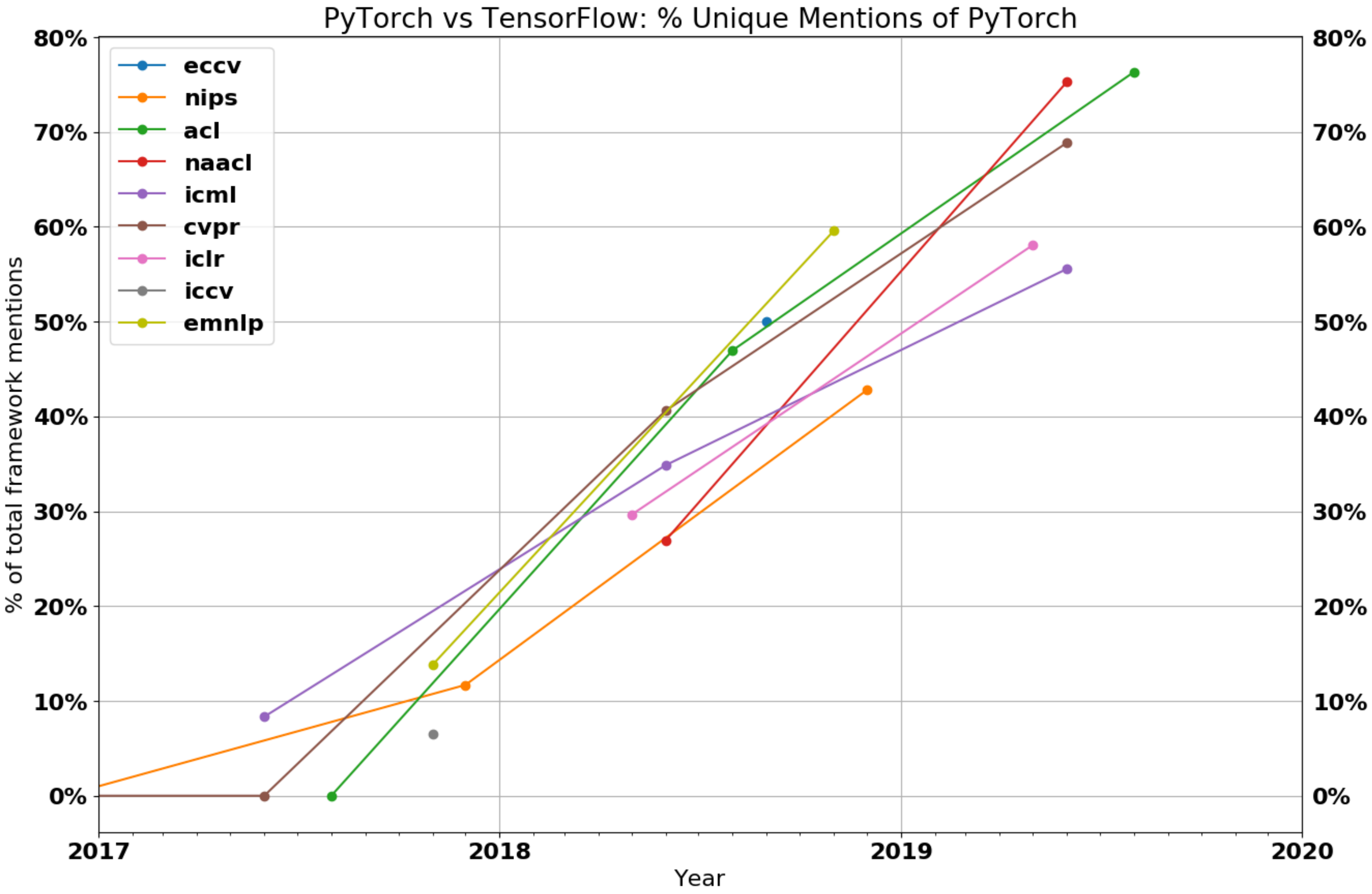


Image source: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

PyTorch vs. TensorFlow: Industry

- No official survey / study on the comparison.
- A quick search on a job posting website turns up 2,389 search results for TensorFlow and 1,366 for PyTorch.
- TensorFlow mostly dominates mobile deployment / embedded systems.

PyTorch vs. TensorFlow

- PyTorch has clean API
  - Native dynamic graphs make it very easy to develop and debug.
  - Can build model using the default API then compile static graph using JIT.
- TensorFlow is a safe bet for most projects.
  - Syntax became a lot more intuitive after 2.0. Not perfect but has huge community and wide usage.
  - Can use same framework for research and production. Probably use a high-level framework.

Deep learning researchers since 2019

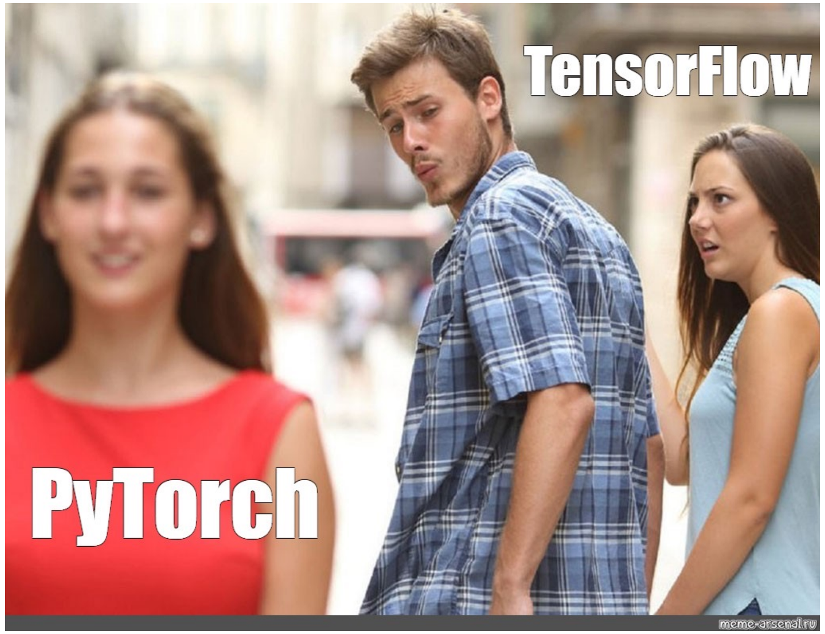


Image source: https://www.meme-arsenal.com/memes/cac8848a0b7c6de2080962f581a5c04b.jpg

Conclusion

After this lecture, you should know:

- Why do we need deep learning frameworks.
- How to build deep learning models by PyTorch and TensorFlow.
- What is high-level API.
- What is the difference between static and dynamic graph.

Suggested Reading

- [PyTorch Tutorial]
- [Keras Basics]
- [Colab Tutorial]

Assignment 1

- Assignment 1 is released. The deadline is 18:00, 14th October.

Thank you!

- Any question?
- Don't hesitate to send email to me for asking questions and discussion. ☺