Assignment Lecture 3

Convolutional Neural Networks

Transfer Learning

Deep learning frameworks

Neural Networks on GPU

Håkon Hukkelås | hakon.hukkelas@ntnu.no

Computer Hardware

Slide source: CS231n Lecture 8

We have two hardware choices:

- NVIDIA GPU
- Google Tensor Processing Unit (TPU)

- AMD? Really not used.

GPU vs CPU

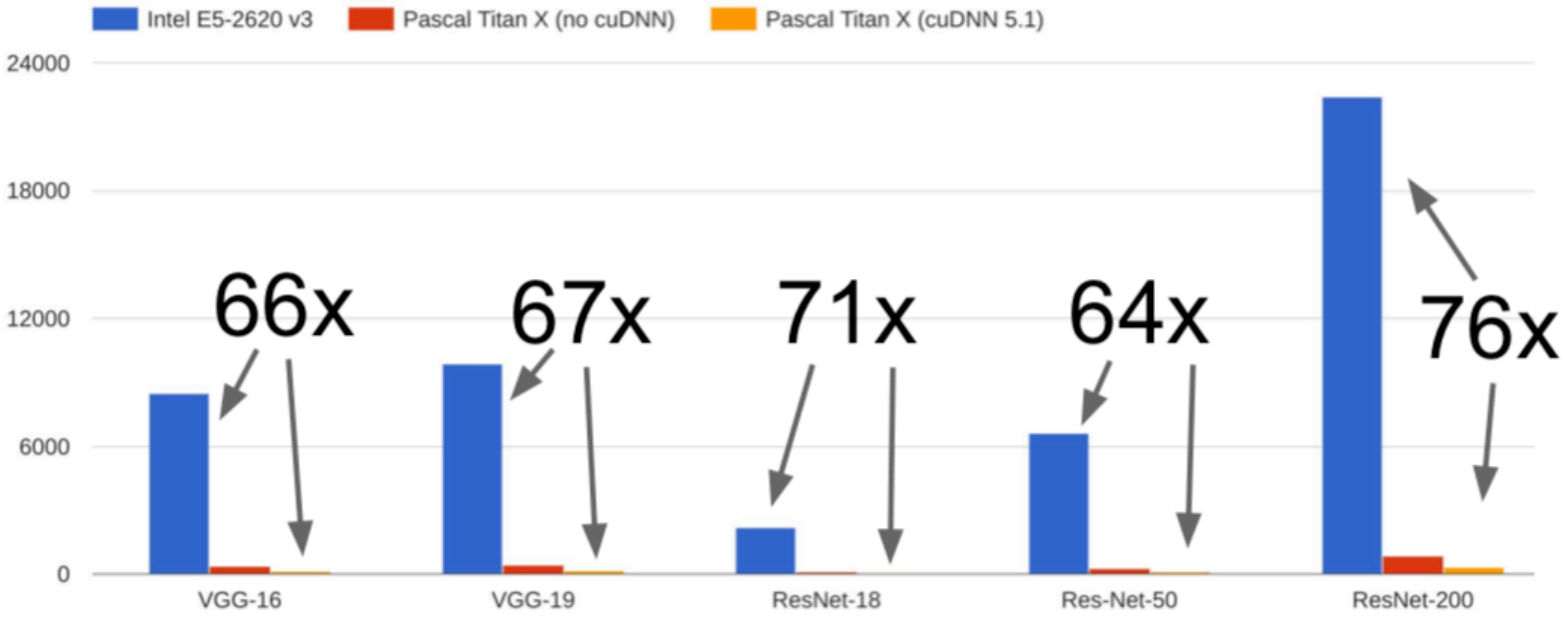
	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	$699	~11.4 TFLOPs FP32

GPU: thousands of “dumb” cores: Great for parallel tasks

Neural Networks: “Only” matrix multiplication, easy in parallel

Slide source: CS231n Lecture 8

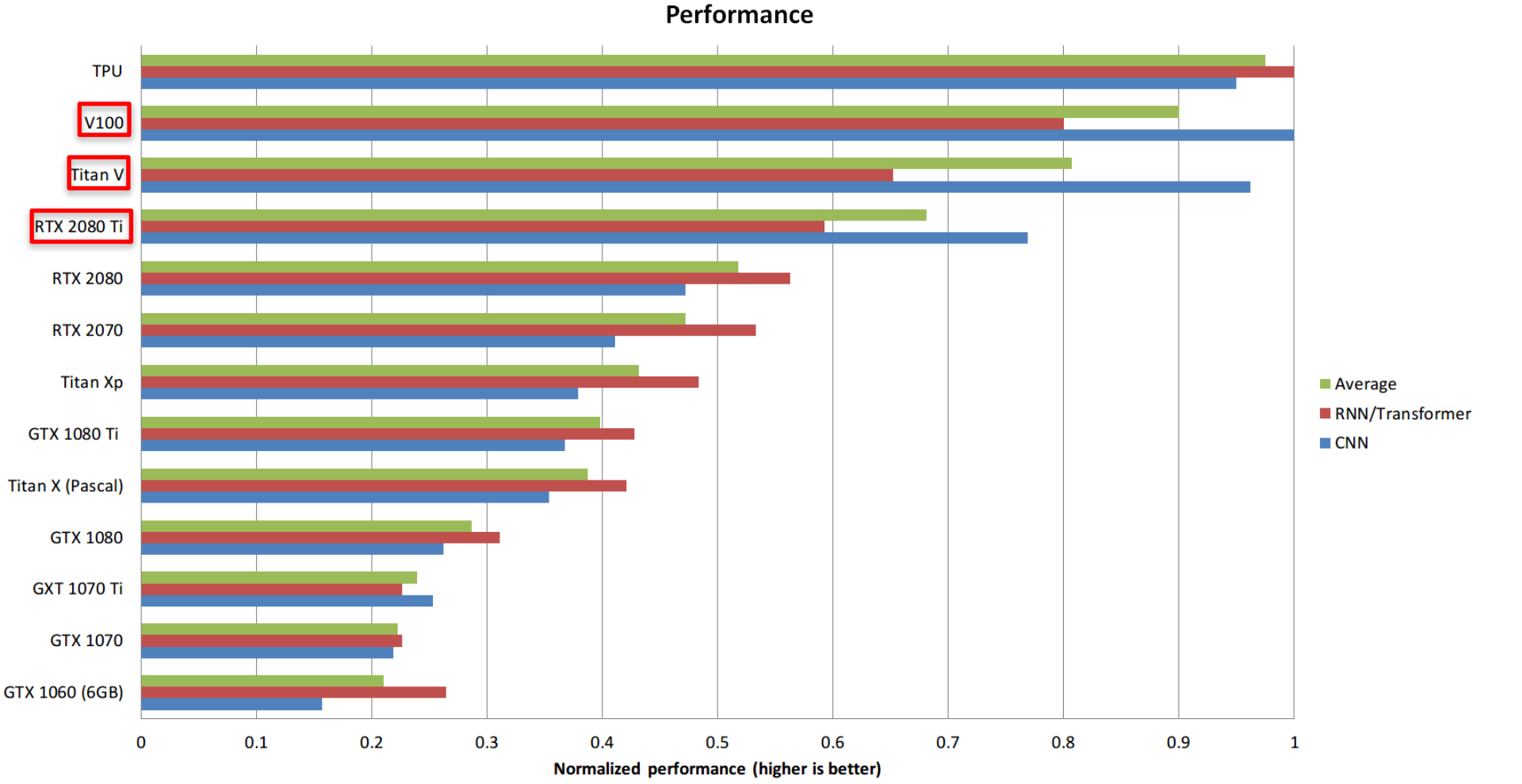
GPU vs CPU for CNNs



Slide source: CS231n Lecture 8

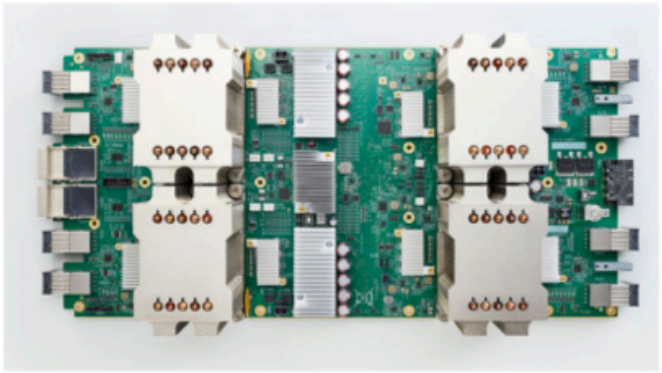
GPU vs CPU: ResNet-200

- Forward pass:
  - Pascal Titan X: 104ms
  - CPU: Dual Xeon E5-2630 v3: 8,666ms (83x slower)
- Backward pass:
  - Pascal Titan X: 191 ms
  - CPU: Dual Xeon E5-2630 v3: 13,758 ms (72x slower)


Performance
TPU
V100
Titan V
RTX 2080 Ti
RTX 2080
RTX 2070
Titan Xp
GTX 1080 Ti
Titan X (Pascal)
GTX 1080
GXT 1070 Ti
GTX 1070
GTX 1060 (6GB)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Normalized performance (higher is better)
Average
RNN/Transformer
CNN

NTNU

Tensor Processing Units (TPU)

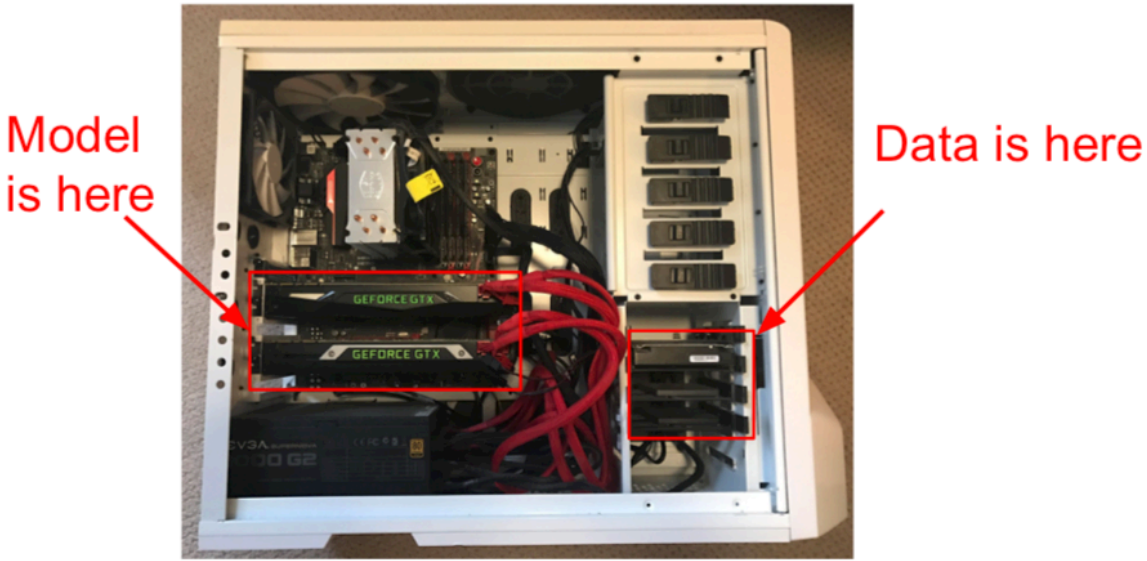


Google Cloud TPU
= 180 TFLOPs of compute!

NVIDIA Tesla V100
= 125 TFLOPs of compute

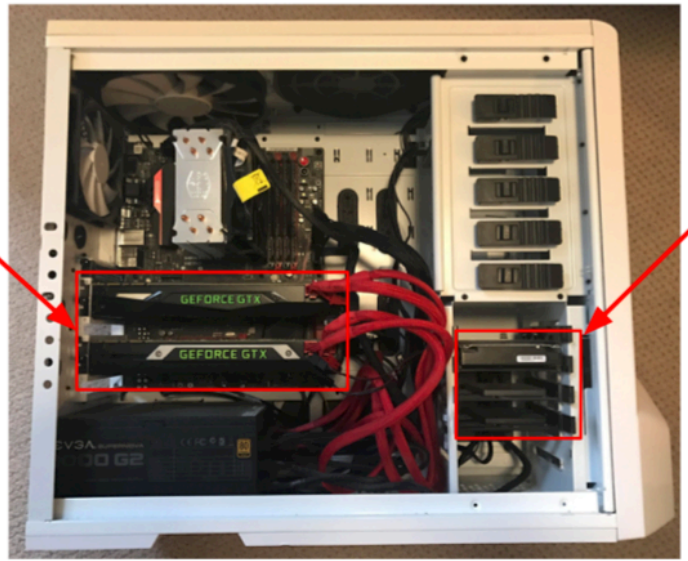
NVIDIA Tesla P100 = 11 TFLOPs of compute
GTX 580 = 0.2 TFLOPs

Issue: CPU <-> GPU Communication



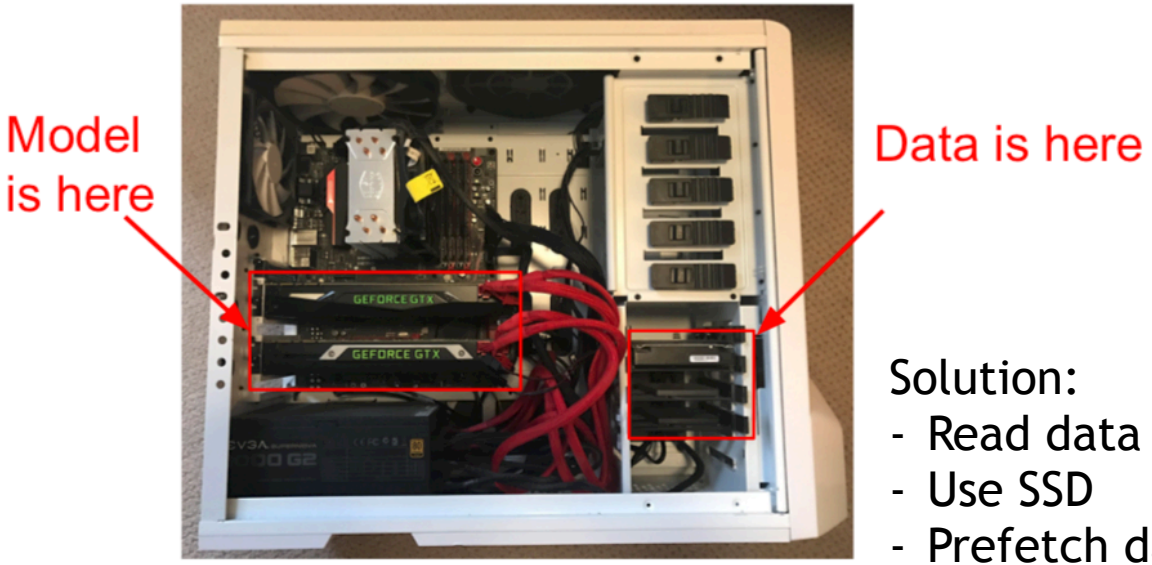
Slide source: CS231n Lecture 8

Issue: CPU <-> GPU Communication



Transferring/reading data
can bottleneck your GPU

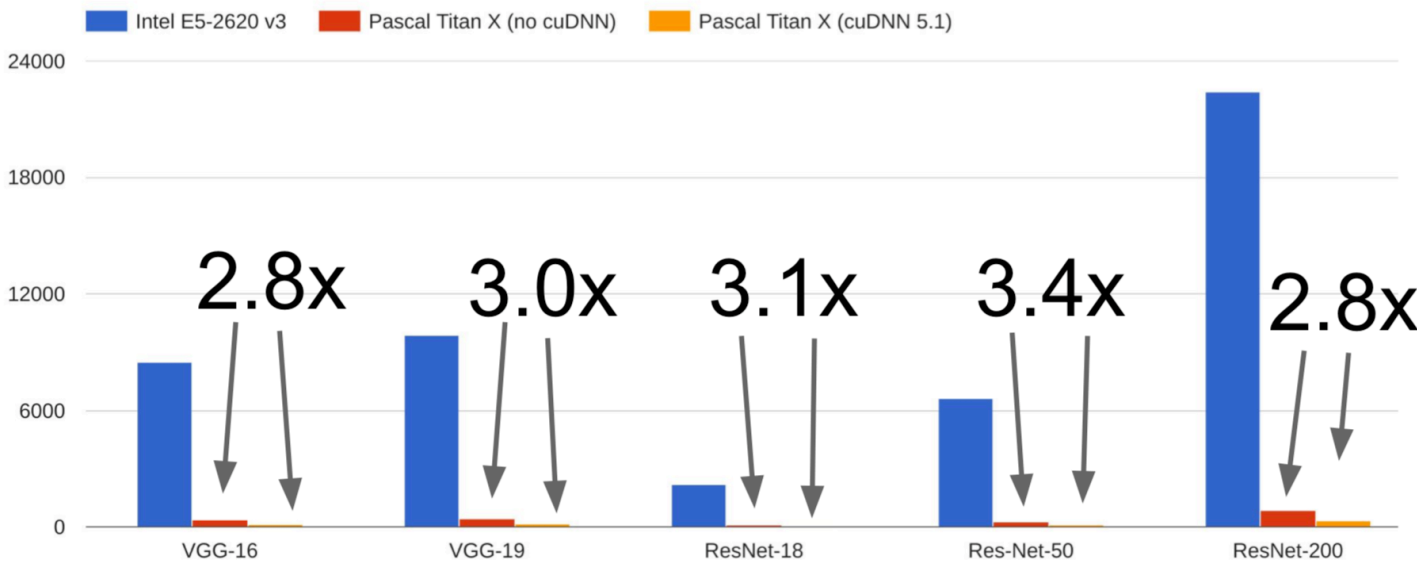
Issue: CPU <-> GPU Communication



Solution:
- Read data into RAM
- Use SSD
- Prefetch data with threads
- Minimize GPU<->CPU communication

Deep Learning Software

- Deep learning software runs on (NVIDIA):
  - CUDA
  - cudNN



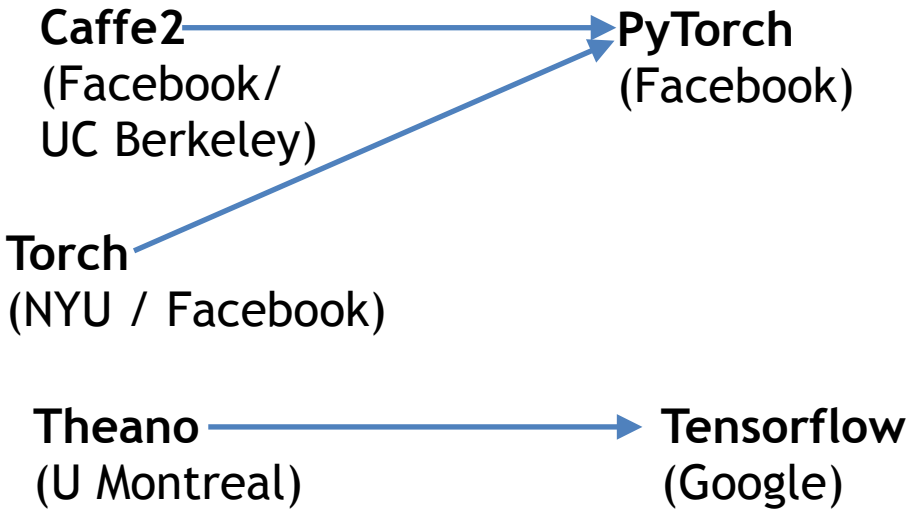
Frameworks

Caffe2
(Facebook/
UC Berkeley)

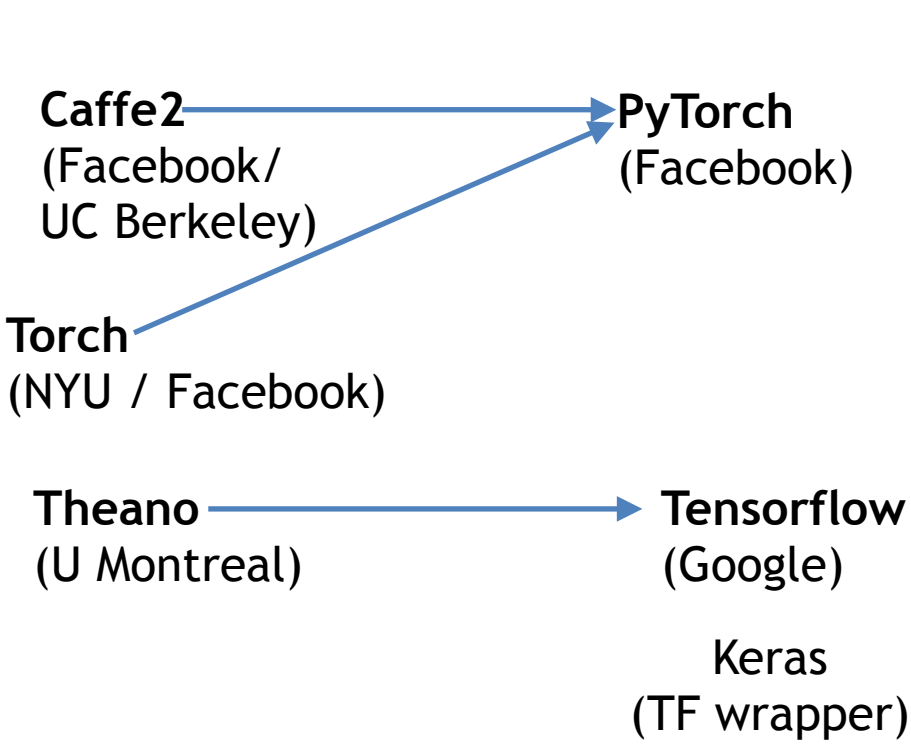
Torch
(NYU / Facebook)

Theano
(U Montreal)

Frameworks


Caffe2
(Facebook/
UC Berkeley)
PyTorch
(Facebook)
Torch
(NYU / Facebook)
Theano
(U Montreal)
Tensorflow
(Google)

Frameworks



Less popular stuff

PaddlePaddle
(Baidu)

Chainer

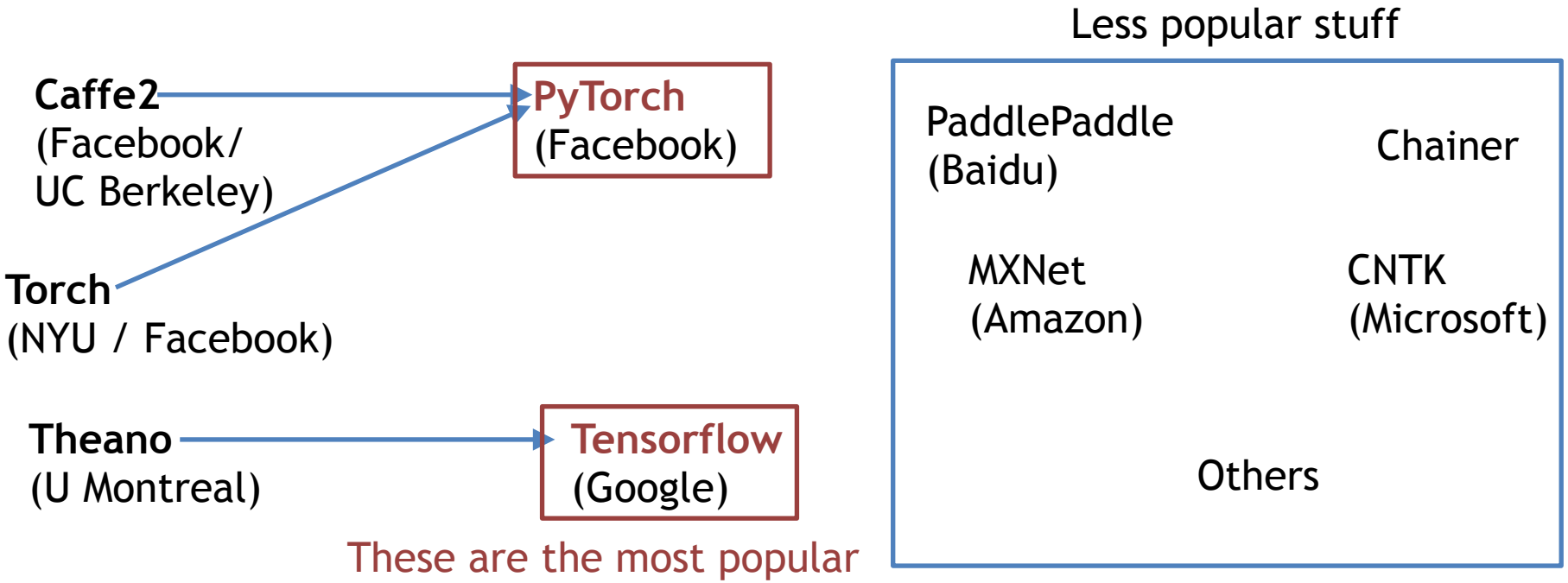
MXNet
(Amazon)

CNTK
(Microsoft)

Others

Slide source: CS231n Lecture 8

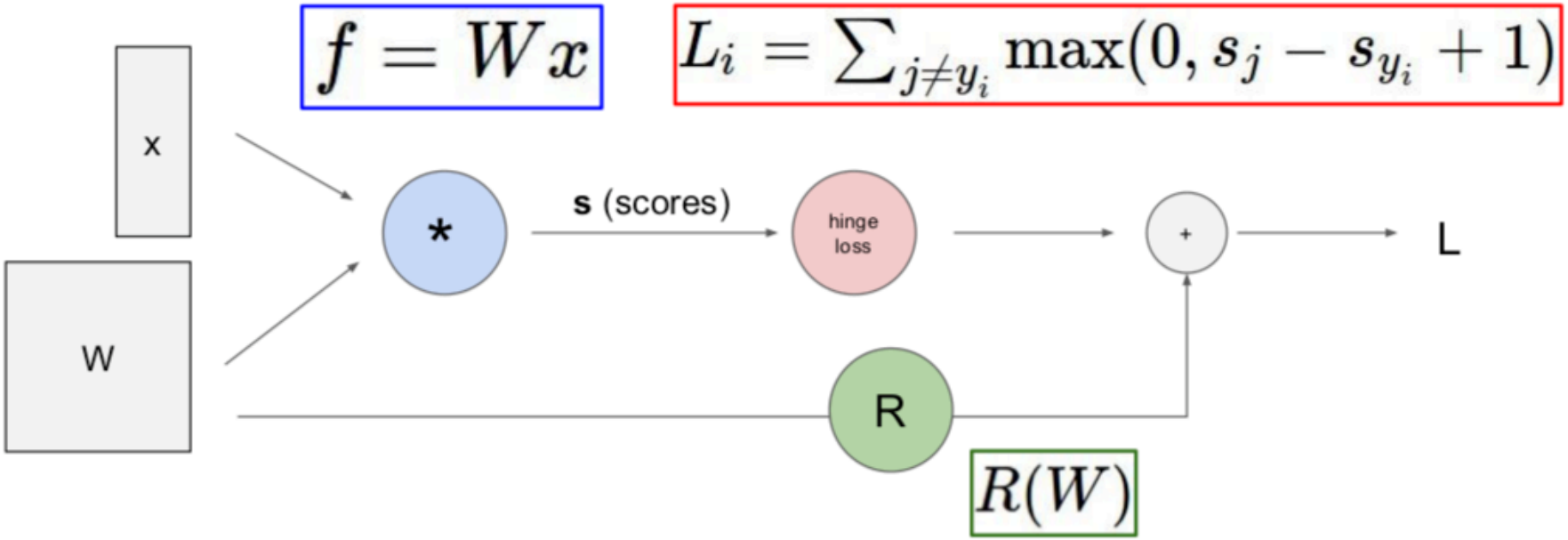
Frameworks


Caffe2
(Facebook/
UC Berkeley)
PyTorch
(Facebook)
Torch
(NYU / Facebook)
Theano
(U Montreal)
Tensorflow
(Google)
These are the most popular
Less popular stuff
PaddlePaddle
(Baidu)
Chainer
MXNet
(Amazon)
CNTK
(Microsoft)
Others

Requirement of deep learning frameworks

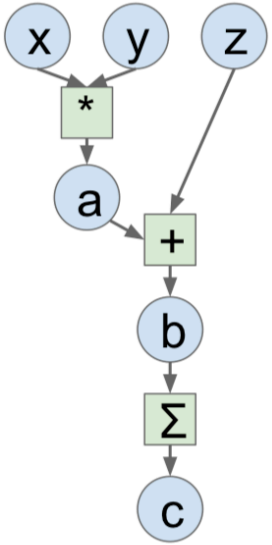
- Quickly implement and test ideas
- Automatically compute gradients
- Run it efficient

Neural Networks = Directed Acyclic Graphs



Slide source: CS231n Lecture 8

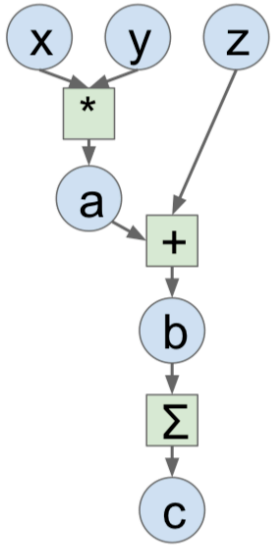
Computational Graph



Computational Graph

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)
```

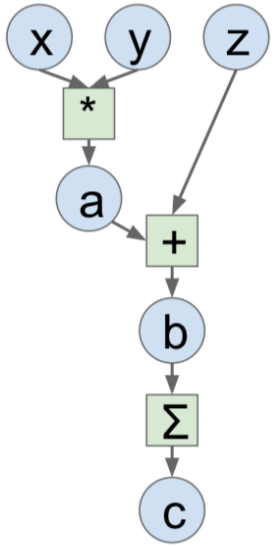


Slide source: CS231n Lecture 8

Computational Graph

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
```

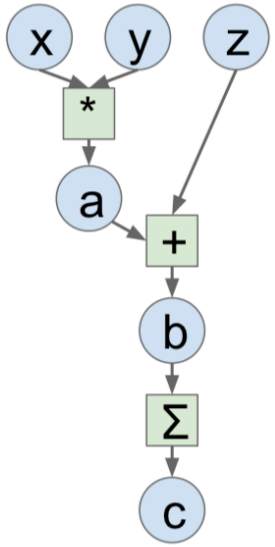


Slide source: CS231n Lecture 8

Computational Graph

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
```



Good:
- Simple, clean API

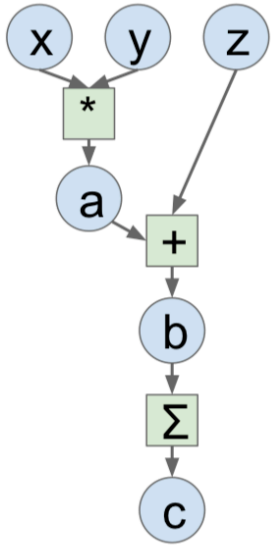
Bad:
- Have to compute gradients ourself
- Can't run on GPU

Computational Graph

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
```



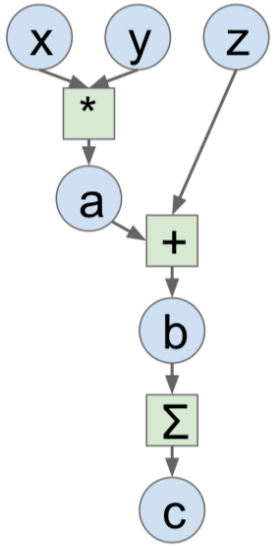
PyTorch

```
import torch

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

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

Computational Graph

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)
```

Torch handles all gradients!

Small sidetrack: Tensorflow vs Pytorch

Computational Graph

- Both tensorflow and pytorch define a directed acyclic graph
- Tensorflow/Keras: **Static graph**, defined before training.
- Pytorch: **Dynamic graph**, defined during training

Pytorch: Dynamic Graph

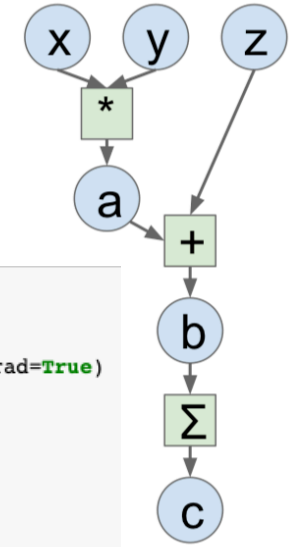
- Pros:
  - Intuitive code
  - “Numpy on GPU”
  - Easy debugging
  - “Pythonic code”
- Cons:
  - Dynamic graph is slower (really though?)
  - Smaller community, but its growing!

A graph is created on the fly


```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

GIF source

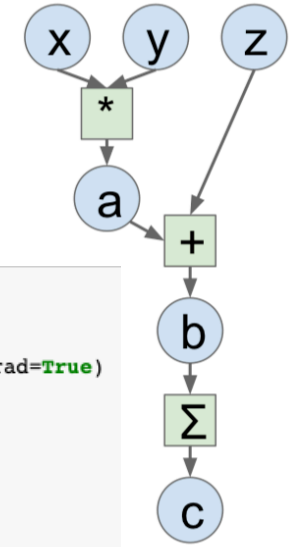
Computational Graph



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 = b.sum()

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

Computational Graph



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 = b.sum()

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

Tensorflow 1.2

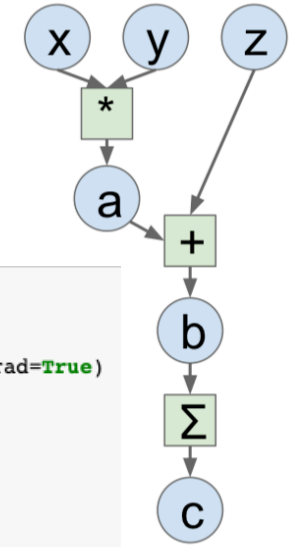
```
N, D = 3, 4

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

a = tf.multiply(x,y)
b = tf.add(a,z)
c = tf.reduce_sum(b)
grad_x = tf.gradients(c, [x])
```

First, define the graph

Computational Graph

PyTorch

```python
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 = b.sum()

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

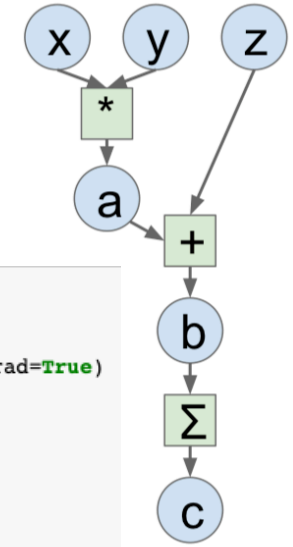


Tensorflow 1.2

```python
N, D = 3, 4

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

a = tf.multiply(x,y)
b = tf.add(a,z)
c = tf.reduce_sum(b)
grad_x = tf.gradients(c, [x])

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
```

Then, initialize a tf.Session

Computational Graph



PyTorch

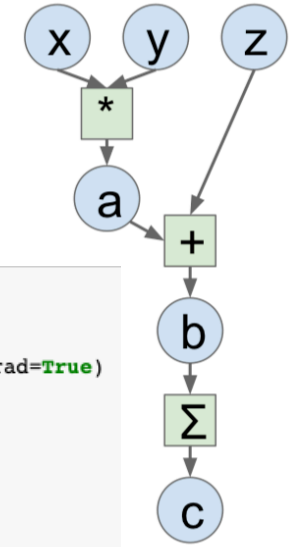
```
import torch
N, D = 3, 4

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 = b.sum()

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

Tensorflow 1.2

```
N, D = 3, 4

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

a = tf.multiply(x,y)
b = tf.add(a,z)
c = tf.reduce_sum(b)
grad_x = tf.gradients(c, [x])

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  values = {
    x: np.random.randn(N,D),
    y: np.random.randn(N,D),
    z: np.random.randn(N,D)
  }
  gradient_x = sess.run(grad_x, feed_dict=values)
```

Then, perform the forward pass

Computational Graph



PyTorch

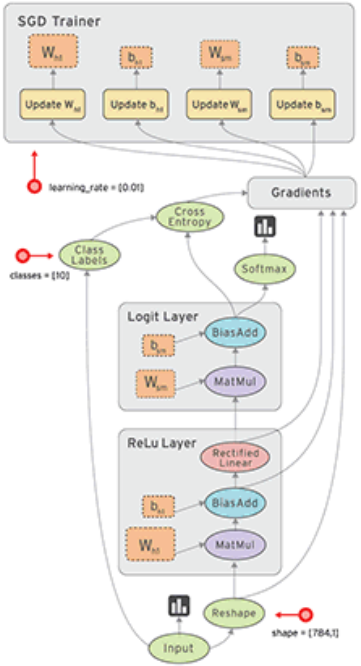
```
import torch
N, D = 3, 4

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 = b.sum()

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

Tensorflow 2.0

```
N, D = 3, 4
x = tf.Variable(tf.ones(shape=(N, D)), name="x")
y = tf.Variable(tf.ones(shape=(N, D)), name="y")
z = tf.Variable(tf.ones(shape=(N, D)), name="z")


with tf.GradientTape() as tape:
    a = x * y
    b = a + z
    c = tf.reduce_sum(b)
    grad = tape.gradient(c, [x])
print(grad)
```

Tensorflow/Keras: Static Graph

- Pros:
  - Faster?
  - Large community
  - Keras is a simpler API
- Cons:
  - Not intuitive
  - Hard to debug



GIF source

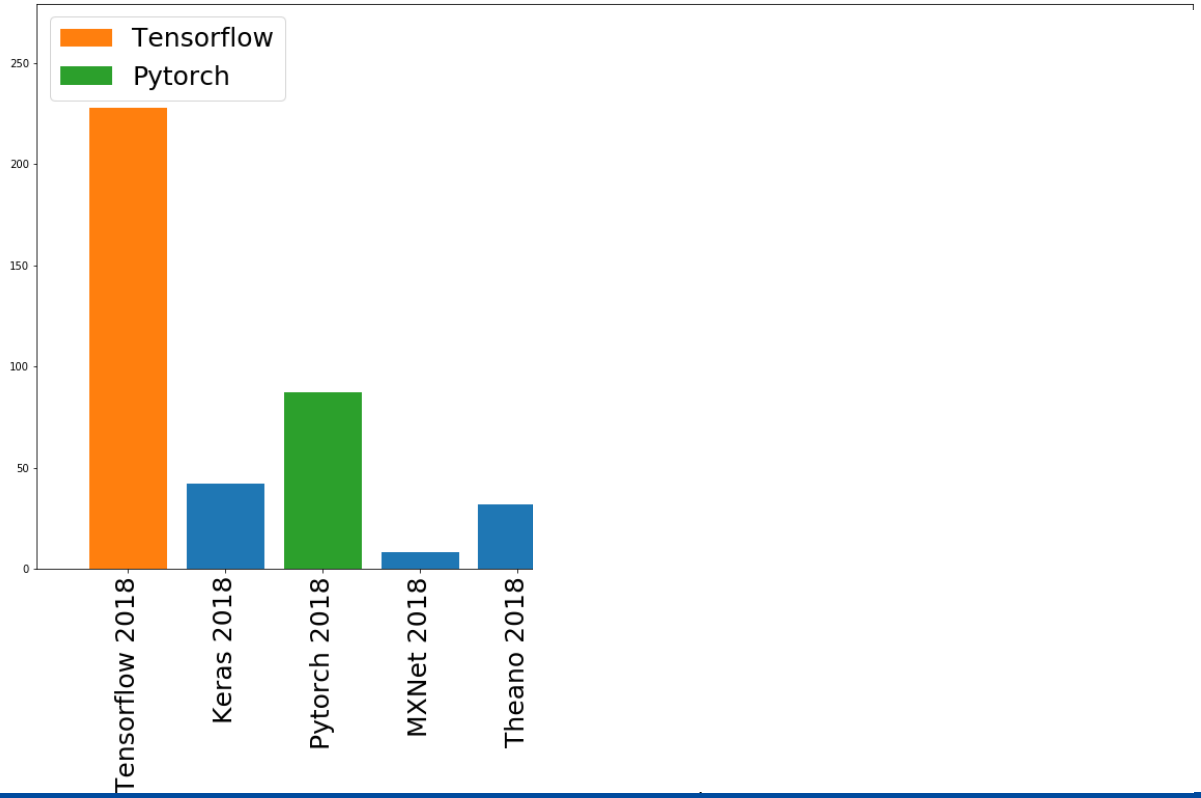
Dynamic vs Static Graphs

- Tensorflow is adding dynamic graphs in TF 2.0
- Pytorch 1.0 includes static graph

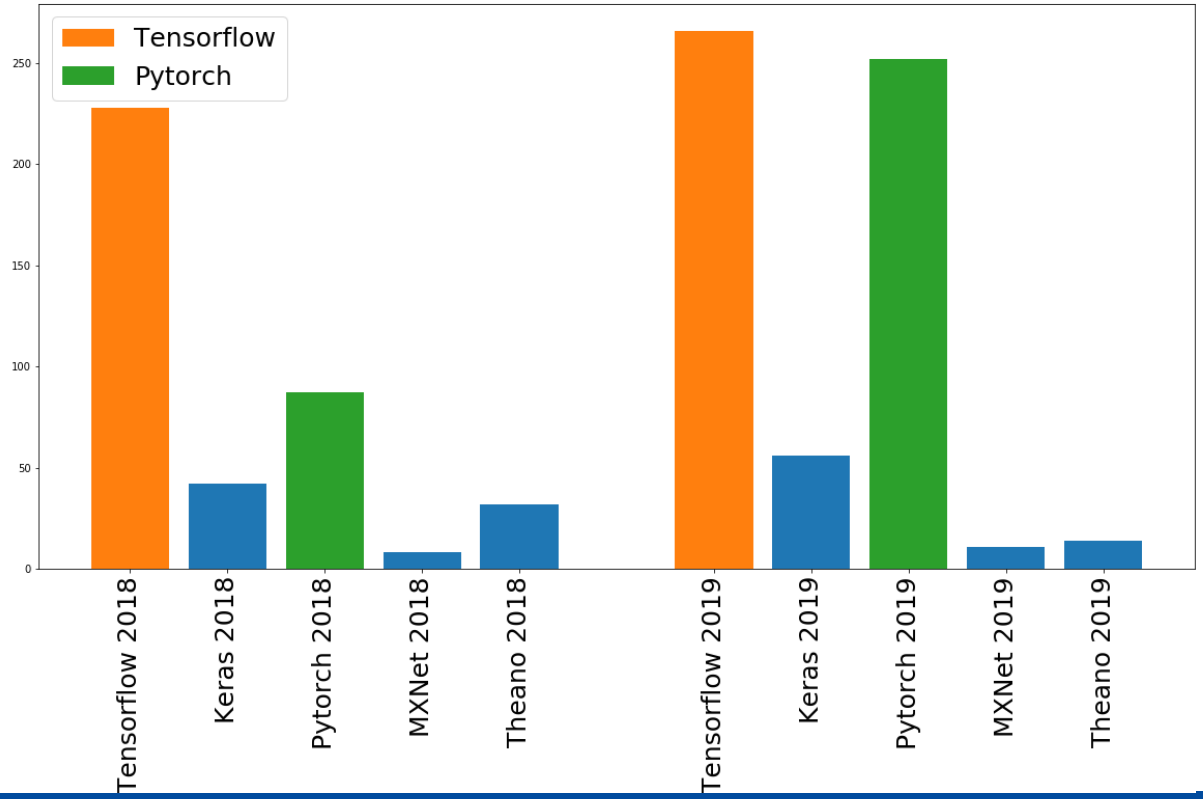
- Lines are blurring….

GIF source

What is actually used? (ICLR 2018/19)



What is actually used? (ICLR 2018/19)



My advice:

- **Pytorch(My favorite):** Has an intuitive API that works for both high level and low level neural networks. Dynamic graphs make it easy to develop and debug.

- **Tensorflow/Keras:** Has a larger community and a wide usage in companies. Is hard to debug and graph setup is initially hard to understand. Probably have to use Keras.

The Pytorch Ecosystem

The Pytorch Ecosystem

CUDnn

CUDA

The Pytorch Ecosystem

Pytorch (torch) Torchvision

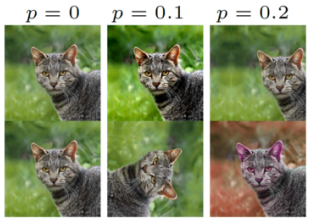
CUDnn

CUDA

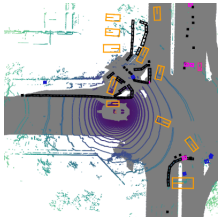
The Pytorch Ecosystem

Detectron2

Stylegan2



MMdetection3d



Pytorch (torch)

Torchvision

CUDnn

CUDA

Pytorch: Fundamental Concepts

- Tensor: Like a numpy array, but can run on GPUs

- Module: A neural network layer; stores states and learnable weights

Pytorch version

- We recommend Pytorch 1.0+, which was released November 2018
- Pytorch 0.4.0/0.4.1 is fine too, not much syntax change
- **NB:** Be careful when looking at older Pytorch code!

Pytorch: Tensors

Example: The 2-layer neural network from Assignment 2

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Initialize the weights randomly

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Do want to save gradients w.r.t weights

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Define our loss function:
Cross Entropy Loss

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Pytorch uses “dataloaders” to efficiently load datasets

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

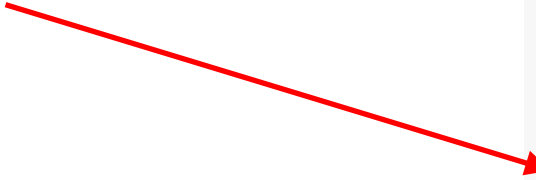
Bias trick and normalization

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Define the forward pass

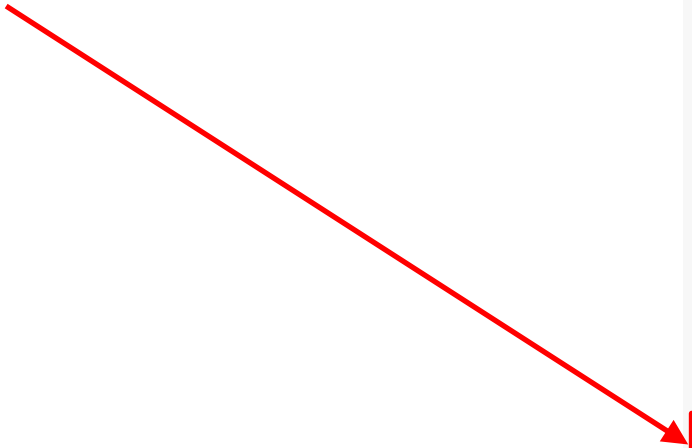


```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Compute the loss



```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Backpropagate the loss

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: Tensors

Make gradient step on weights

torch.no_grad() means “don’t build a computational graph here

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
w1 = torch.randn(I, J, requires_grad=True)
w2 = torch.randn(J, C, requires_grad=True)

loss_function = torch.nn.NLLLoss()
losses = []
for i in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    z_j = X.mm(w1)
    a_j = torch.sigmoid(z_j)
    z_k = a_j.mm(w2)
    y_k = torch.softmax(z_k, dim=1)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      w1 -= learning_rate * w1.grad
      w2 -= learning_rate * w2.grad
      w1.grad.zero_()
      w2.grad.zero_()
```


This is a running example! Do not implement your network like this!

Pytorch: torch.nn

Higher lever wrapper for defining neural networks

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```


This is a running example! Do not implement your network like this!

Pytorch: torch.nn

Higher lever wrapper for defining neural networks

Define each layer in model.

Each layer is a nn.Module() object, containing learnable weights.

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```


This is a running example! Do not implement your network like this!

Pytorch: torch.nn

Changed loss function to nn.CrossEntropyLoss.
This includes the softmax!

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```


This is a running example! Do not implement your network like this!

Pytorch: torch.nn

Simplifies our forward pass!

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```


This is a running example! Do not implement your network like this!

Pytorch: torch.nn

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```

Compute loss and perform backward pass

Each weight in model has requires_grad=True by default

This is a running example! Do not implement your network like this!

Pytorch: torch.nn

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    with torch.no_grad():
      for param in model.parameters():
        param -= learning_rate * param.grad
```

Perform our gradient step
(and disable gradients)

This is a running example! Do not implement your network like this!

Pytorch: torch.optim

Final piece you need to know

Implements **S**tochastic **G**radient **D**escent

Input: our learnable parameters (weights + biases) + learning rate

```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```


This is how its supposed to be!

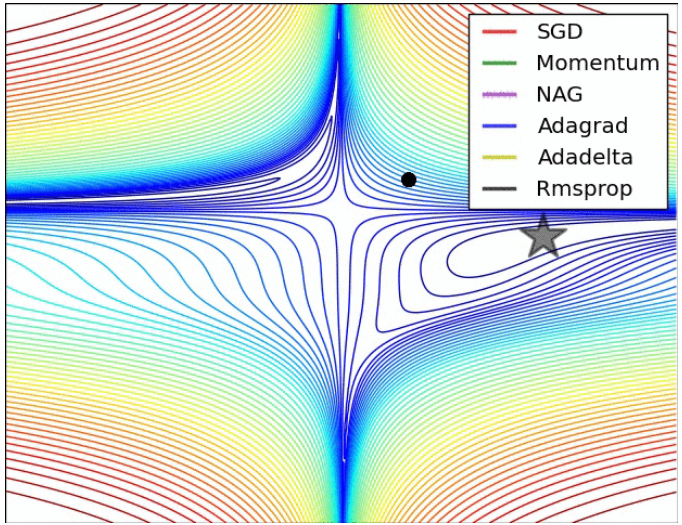
Pytorch: torch.optim

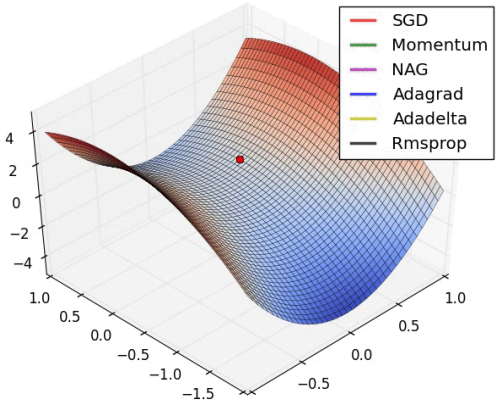
```
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Perform gradient step and reset the gradients

Pytorch Optim Docs

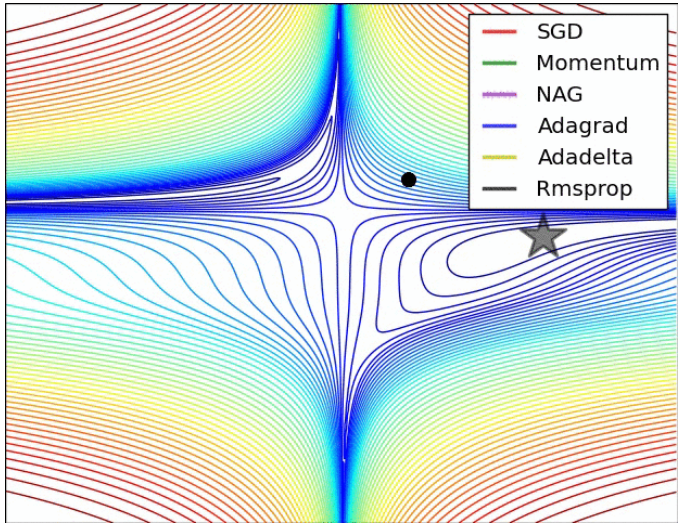
The choice of optimizer is important!

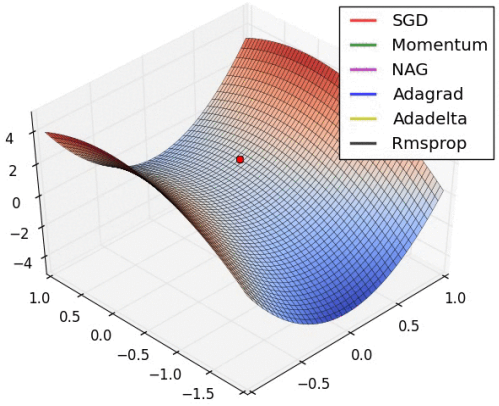




Pytorch Optim Docs

The choice of optimizer is important!





Try out optim.Adam in your assignment!

Pytorch Optim Docs

What about more complex models?

```python
I = 785
J = 64
C = 10
learning_rate = 1e-5
model = nn.Sequential(
  nn.Linear(I, J),
  nn.Sigmoid(),
  nn.Linear(J, C)
  # No need for softmax, since its included in
  # nn.CrossEntropyLoss()
)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(10):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```


Pytorch Optim Docs

Pytorch: nn.Module

A PyTorch **Module** is a neural network layer; it inputs and outputs tensors

Can contain weights or other modules

Required for more complex layers

Easily customizable layers

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
model = TwoLayerNet()

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

```

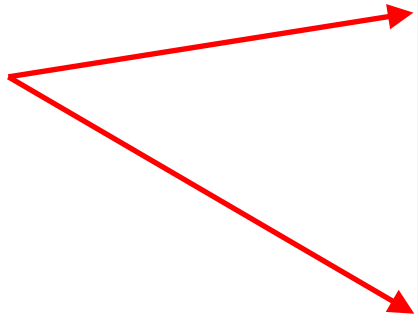
Pytorch: nn.Module

Start with defining the model

```python
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
model = TwoLayerNet()

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: nn.Module

Called when we initialize our model



```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
model = TwoLayerNet()

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: nn.Module

Called when we perform forward pass

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
model = TwoLayerNet()

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X,Y) in dataloader:
    X = pre_process_image(X)
    # forward pass
    y_k = model(X)
    # Compute loss
    loss = loss_function(y_k, Y)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: DataLoaders

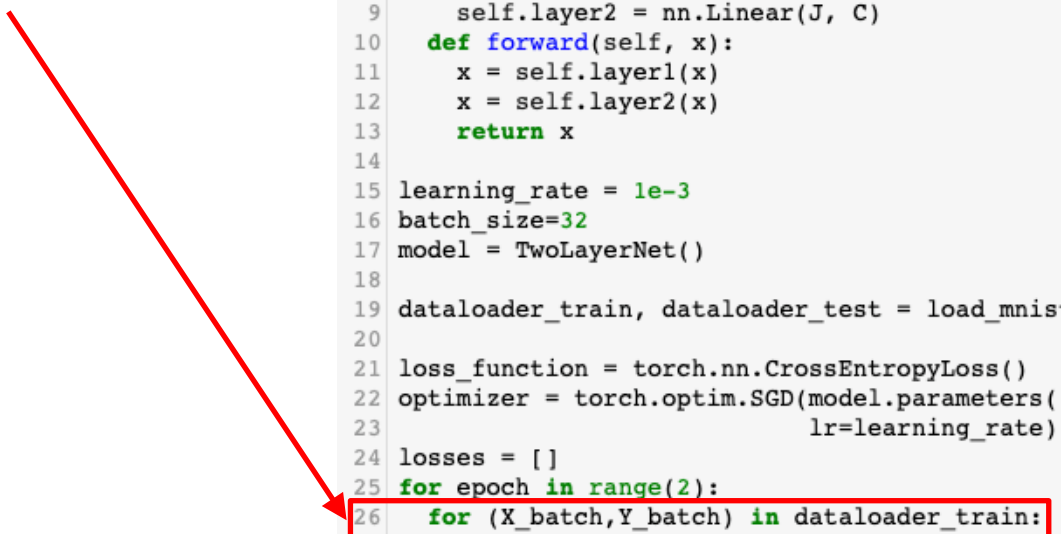
A **DataLoader** wraps a dataset and provides features such as:

- Data augmentation
- Data pre-processing
- mini-batch shuffling and splitting

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet()

dataloader_train, dataloader_test = load_mnist(batch_size

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: DataLoaders

Iterates over each batch in a epoch



```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet()

dataloader_train, dataloader_test = load_mnist(batch_size

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

Utilizing GPU resources is simple!

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet().cuda()

dataloader_train, dataloader_test = load_mnist(batch_size)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = X_batch.cuda(), Y_batch.cuda()
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

Utilizing GPU resources is simple!

.cuda() transfers weights/tensors to GPU VRAM

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet().cuda()

dataloader_train, dataloader_test = load_mnist(batch_size)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = X_batch.cuda(), Y_batch.cuda()
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

Utilizing GPU resources is simple!

.cuda() transfers weights/tensors to GPU VRAM

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet().cuda()

dataloader_train, dataloader_test = load_mnist(batch_size)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = X_batch.cuda(), Y_batch.cuda()
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

Utilizing GPU resources is simple!

.cuda() transfers weights/tensors to GPU VRAM

CAREFUL: Calling .cuda() without a NVIDIA GPU available will cause error!

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = TwoLayerNet().cuda()

dataloader_train, dataloader_test = load_mnist(batch_size)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = X_batch.cuda(), Y_batch.cuda()
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: On GPU

Instead: Implement a to_cuda() function

```python
def to_cuda(elements):
    if torch.cuda.is_available():
        if type(elements) == tuple or type(elements) == list:
            return [x.cuda() for x in elements]
        return elements.cuda()
    return elements
```

```python
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.layer1 = nn.Sequential(
      nn.Linear(I,J),
      nn.Sigmoid()
    )
    self.layer2 = nn.Linear(J, C)
  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

learning_rate = 1e-3
batch_size=32
model = to_cuda(TwoLayerNet())

dataloader_train, dataloader_test = load_mnist(batch_size)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch = pre_process_image(X_batch)
    X_batch, Y_batch = to_cuda([X_batch, Y_batch])
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Pytorch: CNNs

Define a set of convolutional layers to extract **features**

Requires:
- in_channels=1, since MNIST is grayscale
- out_channels= number of filters in layer
- kernel_size=filter width and height
- padding = number of 0's to pad on the side of image

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.feature_extractor = nn.Sequential(
      nn.Conv2d(in_channels=1,
                out_channels=32,
                kernel_size=3,
                padding=1),
      nn.MaxPool2d(kernel_size=2, stride=2),
      nn.Conv2d(in_channels=32,
                out_channels=64,
                kernel_size=3,
                padding=1),
      nn.MaxPool2d(kernel_size=2, stride=2)
    )
    self.classifier = nn.Sequential(
      nn.Linear(64*7*7, 64),
      nn.ReLU(),
      nn.Linear(64, 10)
    )
  def forward(self, x):
    x = self.feature_extractor(x)
    x = x.view(-1, 64*7*7) # Flatten image
    x = self.classifier(x)
    return x
```

Pytorch: CNNs

Need to flatten image before passing it to our fully-connected layer (classifier)

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.feature_extractor = nn.Sequential(
      nn.Conv2d(in_channels=1,
                out_channels=32,
                kernel_size=3,
                padding=1),
      nn.MaxPool2d(kernel_size=2, stride=2),
      nn.Conv2d(in_channels=32,
                out_channels=64,
                kernel_size=3,
                padding=1),
      nn.MaxPool2d(kernel_size=2, stride=2)
    )
    self.classifier = nn.Sequential(
      nn.Linear(64*7*7, 64),
      nn.ReLU(),
      nn.Linear(64, 10)
    )
  def forward(self, x):
    x = self.feature_extractor(x)
    x = x.view(-1, 64*7*7) # Flatten image
    x = self.classifier(x)
    return x
```

Pytorch: CNNs

```
learning_rate = 1e-3
batch_size=32
model = to_cuda(TwoLayerNet())

dataloader_train, dataloader_test = load_mnist(batch_size)

loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
losses = []
for epoch in range(2):
  for (X_batch,Y_batch) in dataloader_train:
    X_batch, Y_batch = to_cuda([X_batch, Y_batch])
    # forward pass
    y_k = model(X_batch)
    # Compute loss
    loss = loss_function(y_k, Y_batch)
    losses.append(loss)
    # Backpropagation
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

```

```
class TwoLayerNet(nn.Module):
  def __init__(self):
    super().__init__()
    I, J, C = 785, 64, 10
    self.feature_extractor = nn.Sequential(
      nn.Conv2d(in_channels=1,
                out_channels=32,
                kernel_size=3,
                padding=1),
      nn.MaxPool2d(kernel_size=2, stride=2),
      nn.Conv2d(in_channels=32,
                out_channels=64,
                kernel_size=3,
                padding=1),
      nn.MaxPool2d(kernel_size=2, stride=2)
    )
    self.classifier = nn.Sequential(
      nn.Linear(64*7*7, 64),
      nn.ReLU(),
      nn.Linear(64, 10)
    )
  def forward(self, x):
    x = self.feature_extractor(x)
    x = x.view(-1, 64*7*7) # Flatten image
    x = self.classifier(x)
    return x
```

Computing Resources

Three options:

- Visual Computing Cluster (tdt4265.idi.ntnu.no/ssh)
- Cybele Computers (at campus and remote)
- Google Colab?

Tutorials in assignment text!