



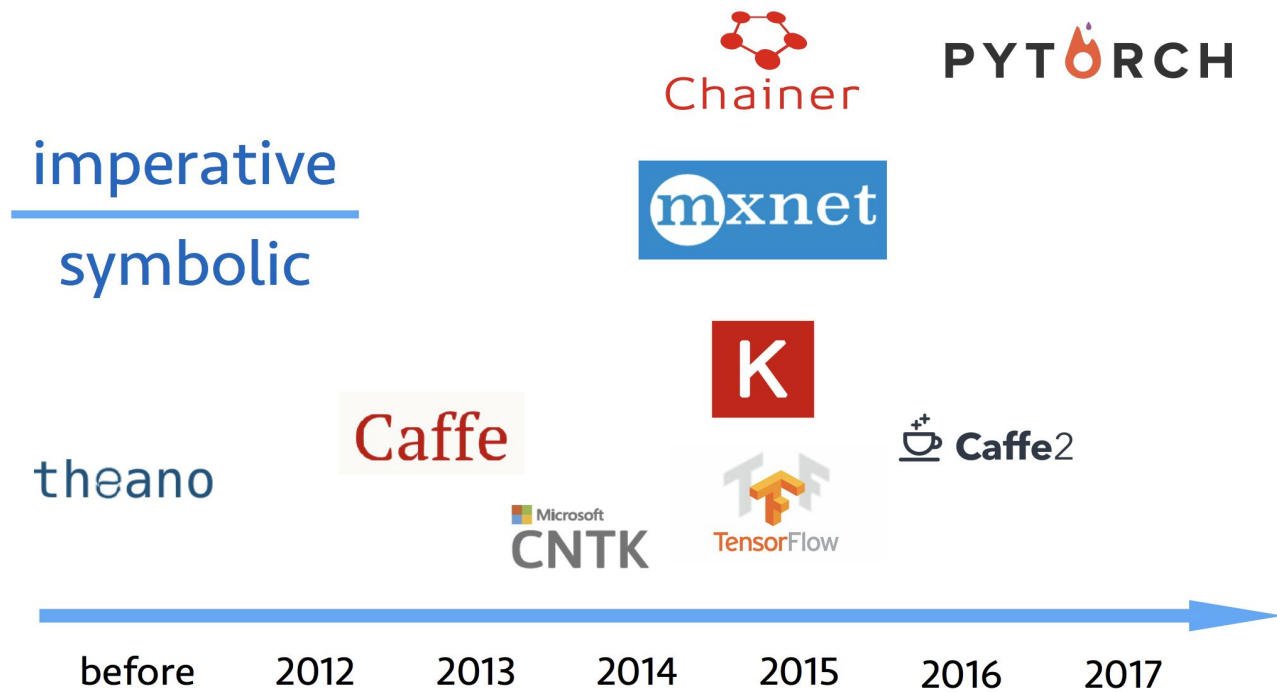
Pytorch Tutorial

Chongruo Wu

Agenda

1. Popular Frameworks
2. Pytorch, Basics
3. Helpful skills

Popular Deep Learning Frameworks



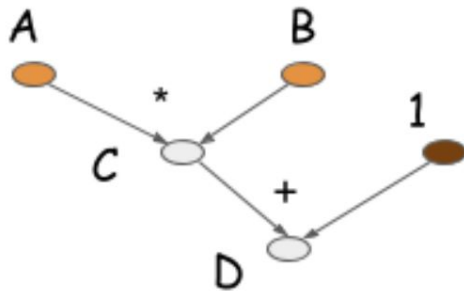
Popular Deep Learning Frameworks

Imperative: Imperative-style programs perform computation as you run them

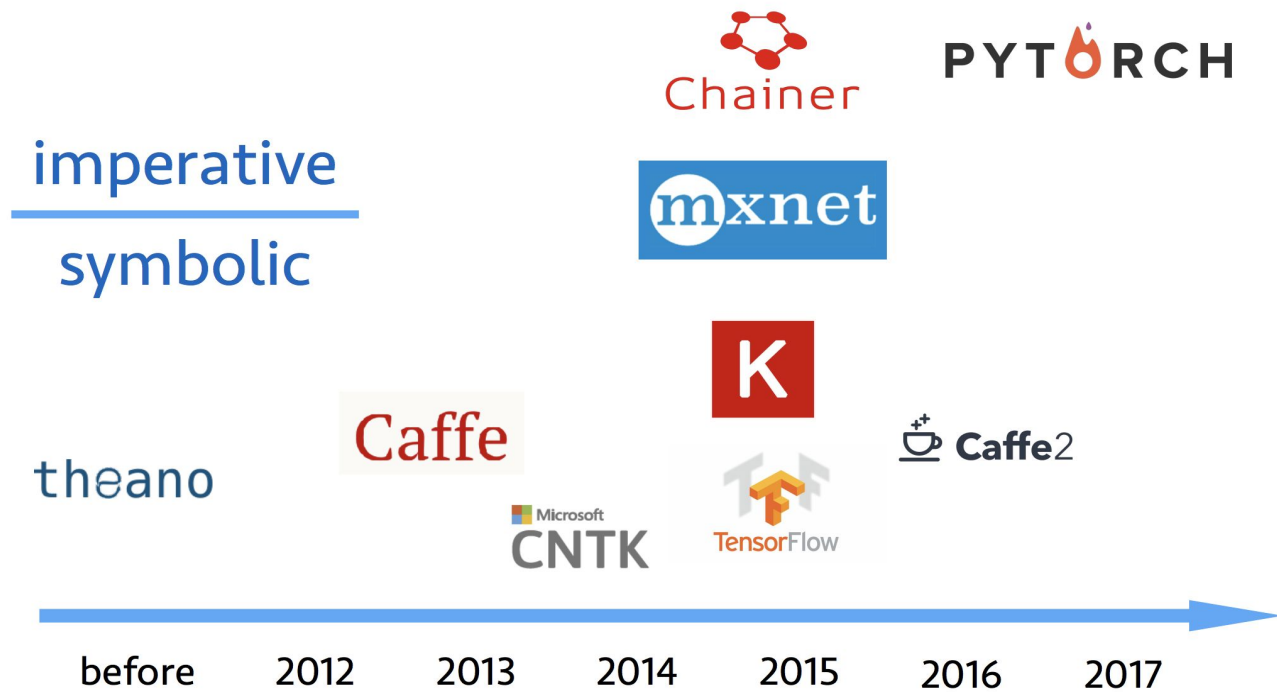
```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

Symbolic: define the function first, then compile them

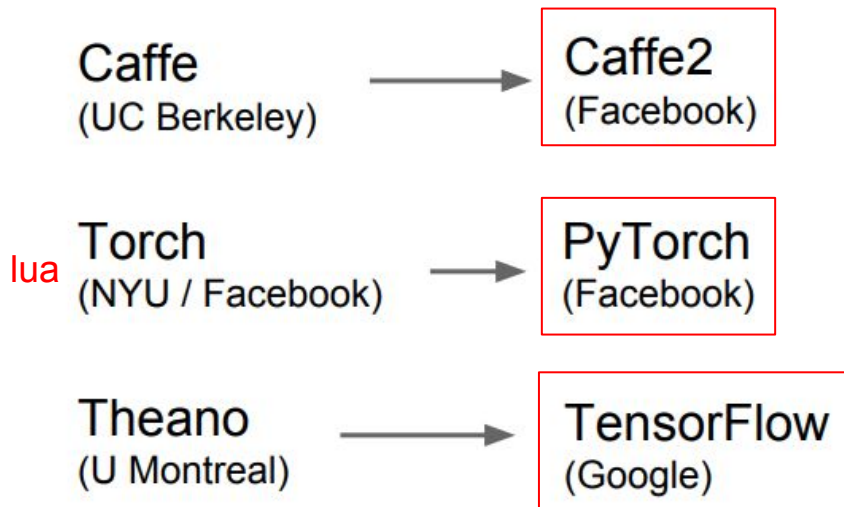
```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```



Popular Deep Learning Frameworks



Popular Deep Learning Frameworks



Paddle
(Baidu)

CNTK
(Microsoft)

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

C++, Python,
R, Julia, Perl
Scala

And others...

Caffe

ResNet-101-deploy.prototxt

```
layer {  
  bottom: "data"  
  top: "conv1"  
  name: "conv1"  
  type: "Convolution"  
  convolution_param {  
    num_output: 64  
    kernel_size: 7  
    pad: 3  
    stride: 2
```

....

(4K lines of codes)

- ◆ Protobuf as the interface
- ◆ Portable
 - ❖ caffe binary + protobuf model
- ◆ Reading and writing protobuf are not straightforward

Tensorflow

Implement Adam

```
# m_t = beta1 * m + (1 - beta1) * g_t
m = self.get_slot(var, "m")
m_scaled_g_values = grad.values * (1 - beta1_t)
m_t = state_ops.assign(m, m * beta1_t,
                       use_locking=self._use_locking)
m_t = state_ops.scatter_add(m_t, grad.indices, m_scaled_g_values,
                           use_locking=self._use_locking)
```

> 300 lines of codes

- ◆ A rich set of operators (~2000)
- ◆ The codes are not very easy to read, e.g. not python-like

Keras

```
model = Sequential()
model.add(Dense(512, activation='relu',
               input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

model.compile(...)
model.fit(...)
```

- ◆ Simple and easy to use
- ◆ Difficult to implement sophisticated algorithms

TensorFlow: Other High-Level Wrappers

Keras (<https://keras.io/>)

TFLearn (<http://tflearn.org/>)

TensorLayer (<http://tensorlayer.readthedocs.io/en/latest/>)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

TF-Slim (<https://github.com/tensorflow/models/tree/master/inception/inception/slim>)

tf.contrib.learn (https://www.tensorflow.org/get_started/tflearn)

Pretty Tensor (<https://github.com/google/prettytensor>)

Sonnet (<https://github.com/deepmind/sonnet>)

Ships with TensorFlow

From Google

From DeepMind

Pytorch & Chainer

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

- ◆ Flexible
- ◆ Complicate programs might be slow to run

MXNet

Implement Resnet

```
bn1 = sym.BatchNorm(data=data, fix_gamma=False)
act1 = sym.Activation(data=bn1, act_type='relu')
conv1 = sym.Convolution(data=act1, num_filters=...
```

Implement Adam

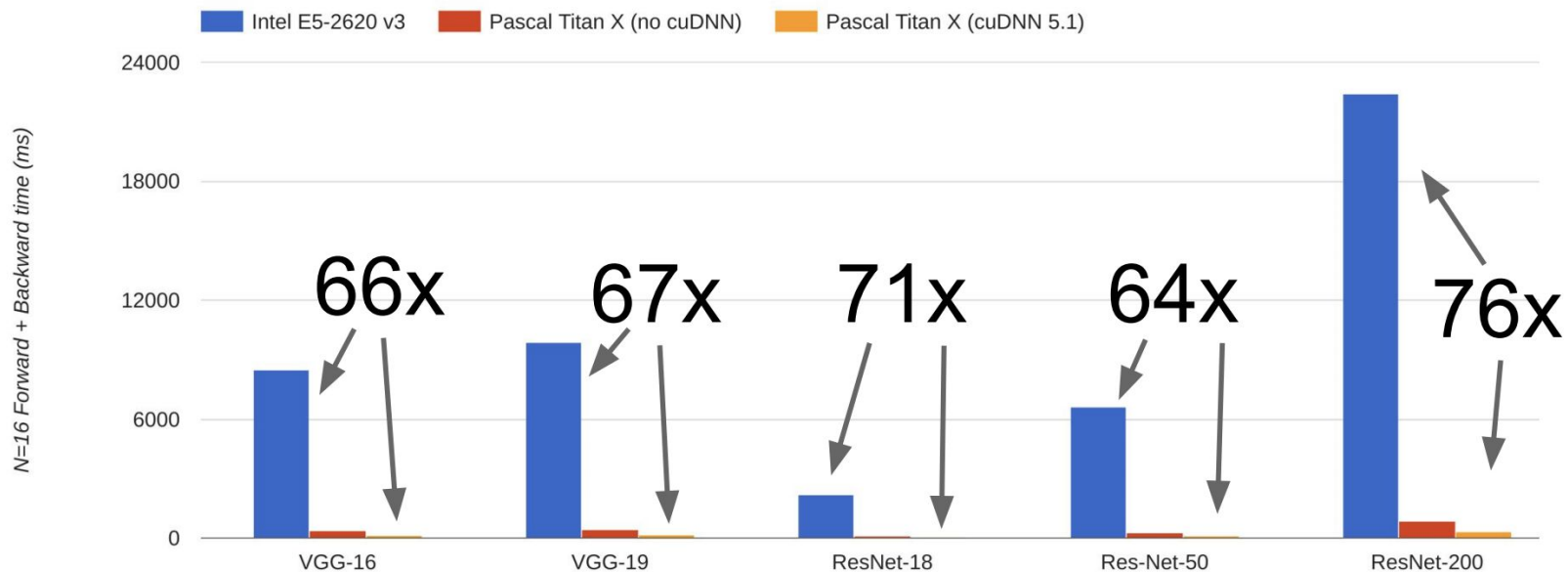
```
coef2 = 1. - self.beta2**t
lr *= math.sqrt(coef2)/coef1

weight -= lr*mean/(sqrt(variance) + self.epsilon)
```

- ◆ Symbolic on network definition
- ◆ Imperative on tensor computation
- ◆ Huh.., not good enough

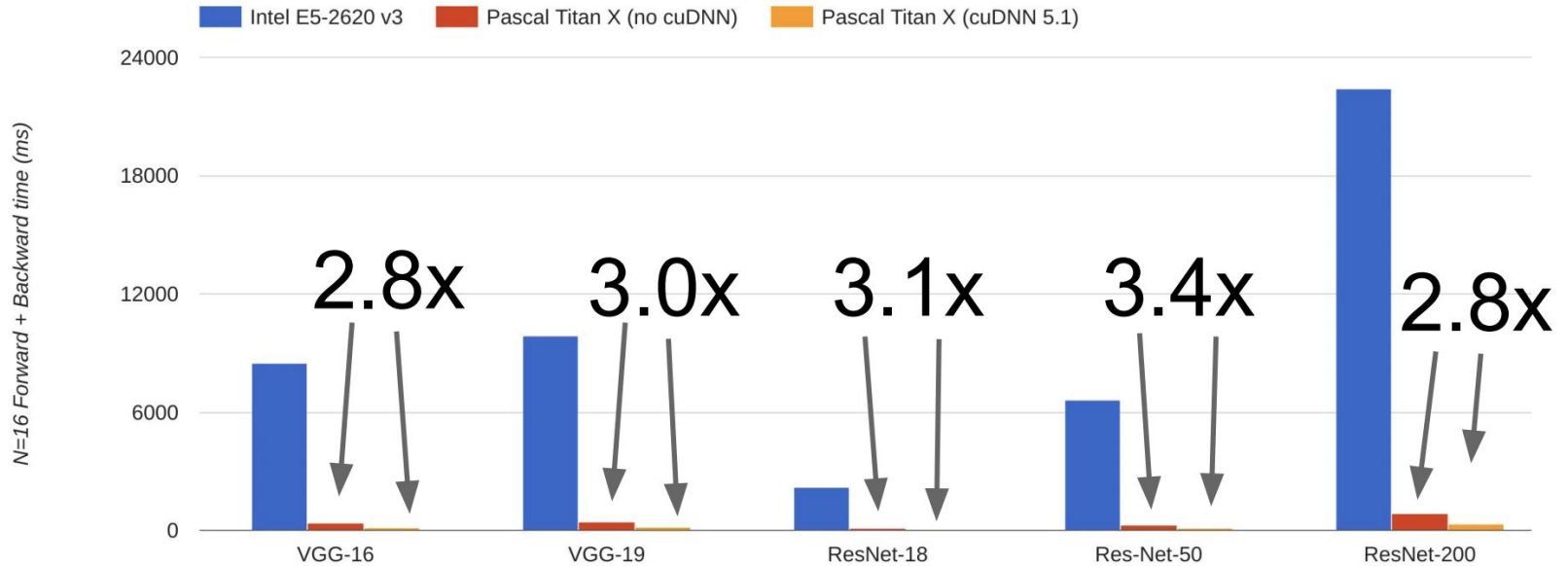
CPU vs GPU in practice

(CPU performance not well-optimized, a little unfair)



CPU vs GPU in practice

cuDNN much faster than
“unoptimized” CUDA



Pytorch

Pytorch

1. Pytorch-0.3.1
2. Pytorch-0.4
Tensor and Variable are merged
3. Pytorch-1.0

PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray,
but runs on GPU

Variable: Node in a
computational graph; stores
data and gradient

Module: A neural network
layer; may store state or
learnable weights

PyTorch: Tensors

PyTorch Tensors are just like numpy arrays, but they can run on GPU.

No built-in notion of computational graph, or gradients, or deep learning.

Here we fit a two-layer net using PyTorch Tensors:

```
import torch

dtype = torch.FloatTensor

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

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

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

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

Pytorch Tensors

```
1 import numpy as np
2 import torch
3
4 # Task: compute matrix multiplication C = AB
5 d = 3000
6
7 # using numpy
8 A = np.random.rand(d, d).astype(np.float32)
9 B = np.random.rand(d, d).astype(np.float32)
10 C = A.dot(B)
11
12 # using torch with gpu
13 A = torch.rand(d, d).cuda()
14 B = torch.rand(d, d).cuda()
15 C = torch.mm(A, B)
```

350 ms

0.1 ms

PyTorch: Tensors

To run on GPU, just cast tensors to a cuda datatype!

```
import torch
```

```
dtype = torch.cuda.FloatTensor
```

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

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    h = x.mm(w1)
```

```
    h_relu = h.clamp(min=0)
```

```
    y_pred = h_relu.mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    grad_y_pred = 2.0 * (y_pred - y)
```

```
    grad_w2 = h_relu.t().mm(grad_y_pred)
```

```
    grad_h_relu = grad_y_pred.mm(w2.t())
```

```
    grad_h = grad_h_relu.clone()
```

```
    grad_h[h < 0] = 0
```

```
    grad_w1 = x.t().mm(grad_h)
```

```
w1 -= learning_rate * grad_w1
```

```
w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Create random tensors
for data and weights



```
import torch

dtype = torch.FloatTensor

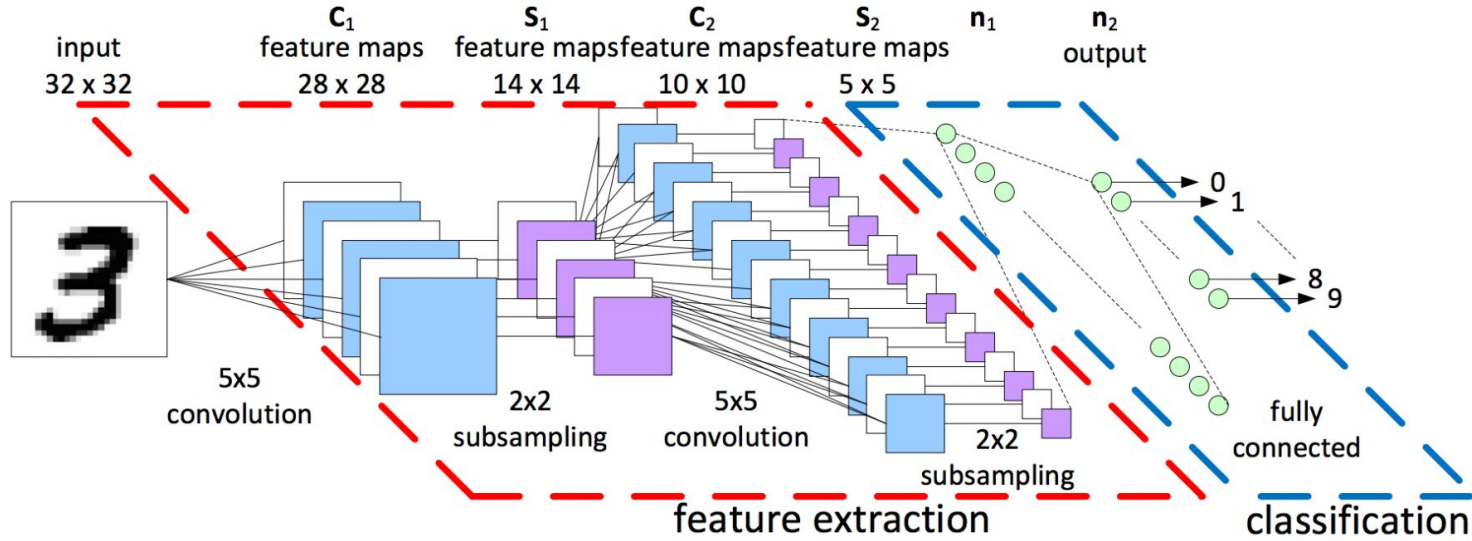
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

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

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

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

Simple Network



1. Forward: compute output of each layer
2. Backward: compute gradient
3. Update: update the parameters with computed gradient

PyTorch: Tensors

Forward pass: compute predictions and loss

```
import torch

dtype = torch.FloatTensor

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


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

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

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


PyTorch: Tensors

Backward pass:
manually compute
gradients



```
import torch

dtype = torch.FloatTensor

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

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

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

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


PyTorch: Tensors

Gradient descent
step on weights

```
import torch

dtype = torch.FloatTensor

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

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

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

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

PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray,
but runs on GPU

Variable: Node in a
computational graph; stores
data and gradient

Module: A neural network
layer; may store state or
learnable weights

PyTorch: Autograd

A PyTorch **Variable** is a node in a computational graph

x.data is a Tensor

x.grad is a Variable of gradients
(same shape as x.data)

x.grad.data is a Tensor of gradients

```
import torch
from torch.autograd import Variable

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

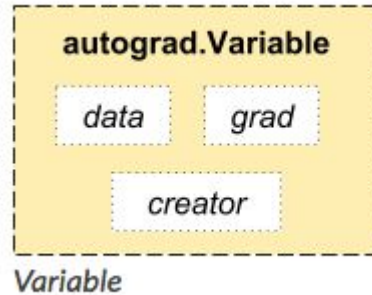
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

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

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

Variable

The `autograd` package provides automatic differentiation for all operations on Tensors.



“ `autograd.Variable` is the central class of the package. It wraps a Tensor, and supports nearly all of operations defined on it.

Once you finish your computation you can call `.backward()` and have all the gradients computed automatically. “

Computational Graphs

Numpy

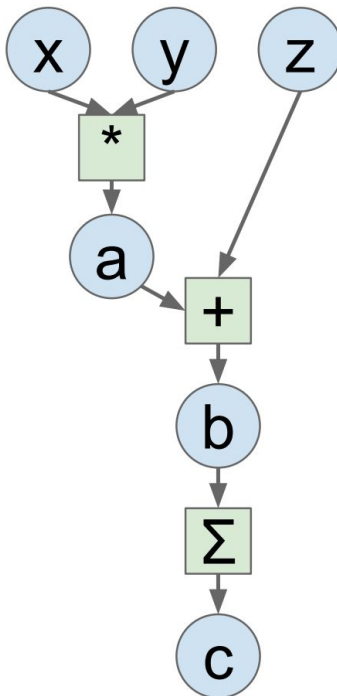
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

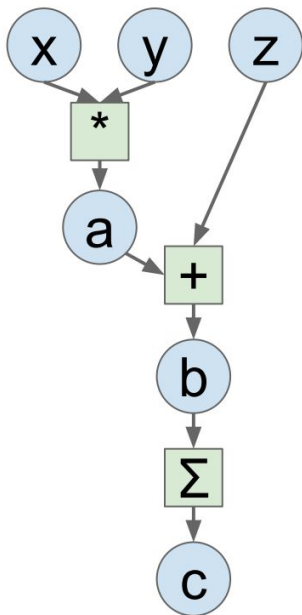
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

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
```



Computational Graphs



Define **Variables** to start building a computational graph

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

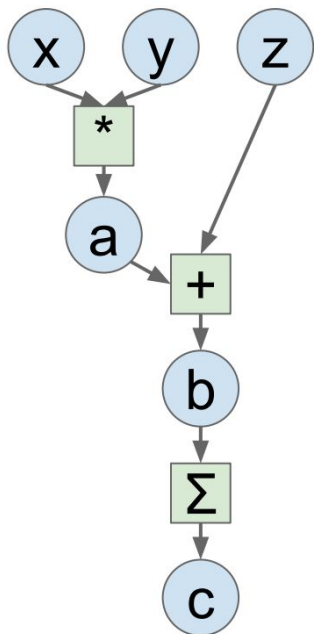
x = Variable(torch.randn(N, D),
              requires_grad=True)
y = Variable(torch.randn(N, D),
              requires_grad=True)
z = Variable(torch.randn(N, D),
              requires_grad=True)

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

c.backward()

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

Computational Graphs



Forward pass
looks just like
numpy

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

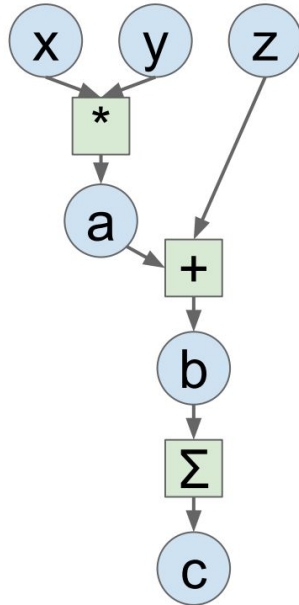
x = Variable(torch.randn(N, D),
              requires_grad=True)
y = Variable(torch.randn(N, D),
              requires_grad=True)
z = Variable(torch.randn(N, D),
              requires_grad=True)

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

c.backward()

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

Computational Graphs



Calling `c.backward()`
computes all
gradients

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

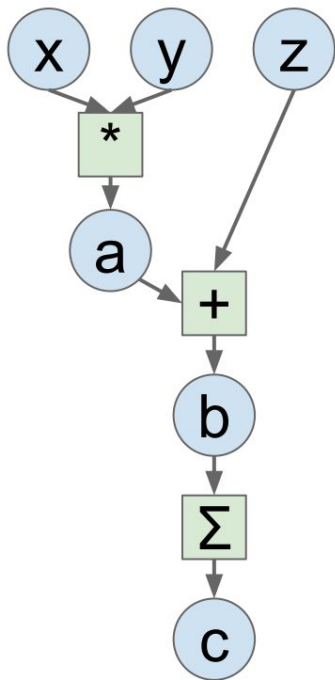
x = Variable(torch.randn(N, D),
              requires_grad=True)
y = Variable(torch.randn(N, D),
              requires_grad=True)
z = Variable(torch.randn(N, D),
              requires_grad=True)

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

c.backward()

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


Computational Graphs



Run on GPU by
casting to `.cuda()`

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

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

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

c.backward()

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

PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray,
but runs on GPU

Variable: Node in a
computational graph; stores
data and gradient

Module: A neural network
layer; may store state or
learnable weights

Module, single layer

torch.nn

Parameters

Containers

Convolution Layers

Conv1d

Conv2d

Conv3d

ConvTranspose1d

ConvTranspose2d

ConvTranspose3d

torch.nn

Parameters

Containers

Convolution Layers

Pooling Layers

MaxPool1d

MaxPool2d

MaxPool3d

MaxUnpool1d

MaxUnpool2d

MaxUnpool3d

AvgPool1d

AvgPool2d

AvgPool3d

FractionalMaxPool2d

LPPool2d

AdaptiveMaxPool1d

AdaptiveMaxPool2d

AdaptiveMaxPool3d

AdaptiveAvgPool1d

AdaptiveAvgPool2d

AdaptiveAvgPool3d

Loss functions

L1Loss

MSELoss

CrossEntropyLoss

NLLLoss

PoissonNLLLoss

KLDivLoss

BCELoss

BCEWithLogitsLoss

MarginRankingLoss

HingeEmbeddingLoss

MultiLabelMarginLoss

SmoothL1Loss

SoftMarginLoss

MultiLabelSoftMarginLoss

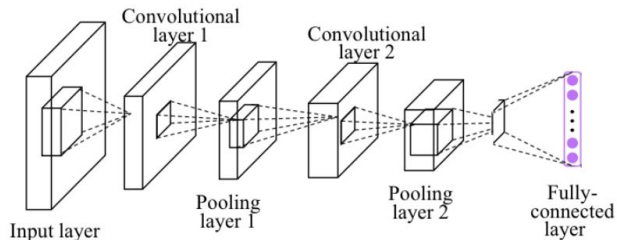
CosineEmbeddingLoss

MultiMarginLoss

TripletMarginLoss

Other layers:
Dropout, Linear,
Normalization Layer

Module, network

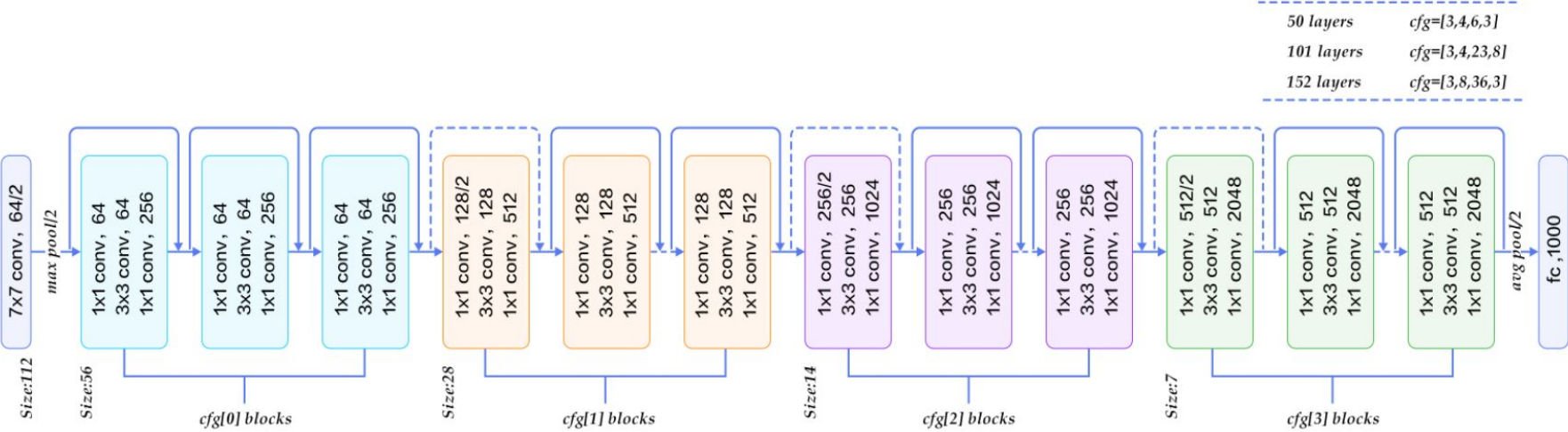


```
class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # 320 -> 10
```

```
    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

Module, sub-network



Module, sub-network

```
# Define a resnet block
class ResnetBlock(nn.Module):
    def __init__(self, dim, padding_type, norm_layer, use_dropout, use_bias):
        super(ResnetBlock, self).__init__()
        self.conv_block = self.build_conv_block(dim, padding_type, norm_layer, use_dropout, use_bias)

    def build_conv_block(self, dim, padding_type, norm_layer, use_dropout, use_bias):
        conv_block = []
        p = 0
        if padding_type == 'reflect':
            conv_block += [nn.ReflectionPad2d(1)]
        elif padding_type == 'replicate':
            conv_block += [nn.ReplicationPad2d(1)]
        elif padding_type == 'zero':
            p = 1
        else:
            raise NotImplementedError('padding [%s] is not implemented' % padding_type)

        conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p, bias=use_bias),
                       norm_layer(dim),
                       nn.ReLU(True)]

        if use_dropout:
            conv_block += [nn.Dropout(0.5)]

        p = 0
        if padding_type == 'reflect':
            conv_block += [nn.ReflectionPad2d(1)]
        elif padding_type == 'replicate':
            conv_block += [nn.ReplicationPad2d(1)]
        elif padding_type == 'zero':
            p = 1
        else:
            raise NotImplementedError('padding [%s] is not implemented' % padding_type)
        conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p, bias=use_bias),
                       norm_layer(dim)]

        return nn.Sequential(*conv_block)

    def forward(self, x):
        out = x + self.conv_block(x)
        return out
```

```
class ResnetGenerator(nn.Module):
    def __init__(self, input_nc, output_nc, ngf=64, norm_layer=nn.BatchNorm2d, use_dropout=False, n_blocks=6, gpu_ids=[], padding_type='ref
assert(n_blocks >= 0)
super(ResnetGenerator, self).__init__()
self.input_nc = input_nc
self.output_nc = output_nc
self.ngf = ngf
self.gpu_ids = gpu_ids
if type(norm_layer) == functools.partial:
    use_bias = norm_layer.func == nn.InstanceNorm2d
else:
    use_bias = norm_layer == nn.InstanceNorm2d

model = [nn.ReflectionPad2d(3),
         nn.Conv2d(input_nc, ngf, kernel_size=7, padding=0,
                   bias=use_bias),
         norm_layer(ngf),
         nn.ReLU(True)]

n_downsampling = 2
for i in range(n_downsampling):
    mult = 2**i
    model += [nn.Conv2d(ngf * mult, ngf * mult * 2, kernel_size=3,
                       stride=2, padding=1, bias=use_bias),
             norm_layer(ngf * mult * 2),
             nn.ReLU(True)]

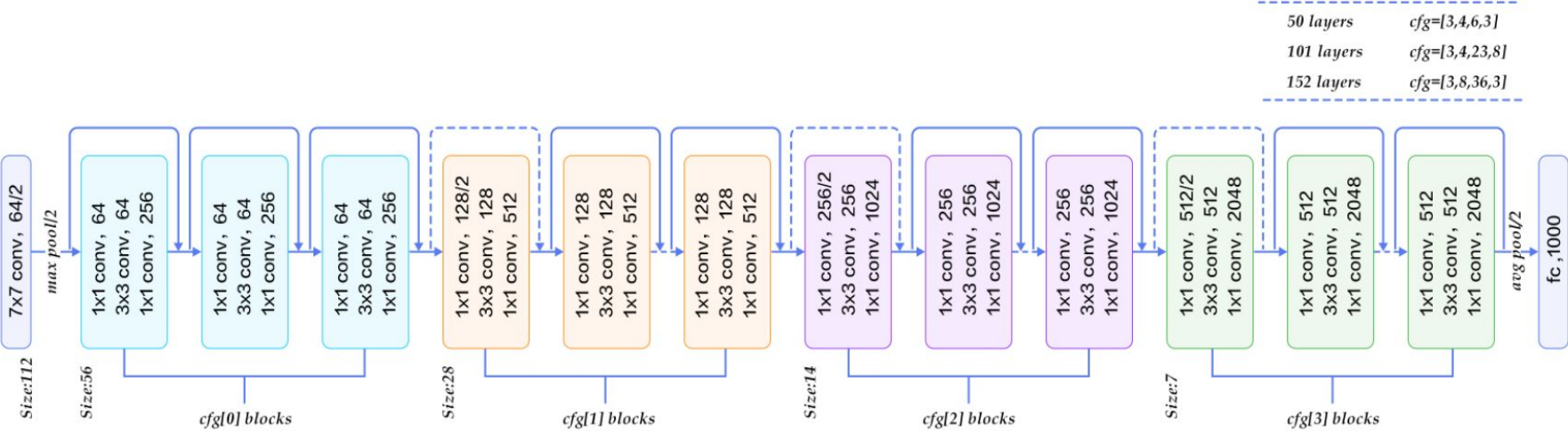
mult = 2**n_downsampling
for i in range(n_blocks):
    model += [ResnetBlock(ngf * mult, padding_type=padding_type, norm_layer=norm_layer, use_dropout=use_dropout, use_bias=use_bias)]

for i in range(n_downsampling):
    mult = 2**(n_downsampling - i)
    model += [nn.ConvTranspose2d(ngf * mult, int(ngf * mult / 2),
                                kernel_size=3, stride=2,
```

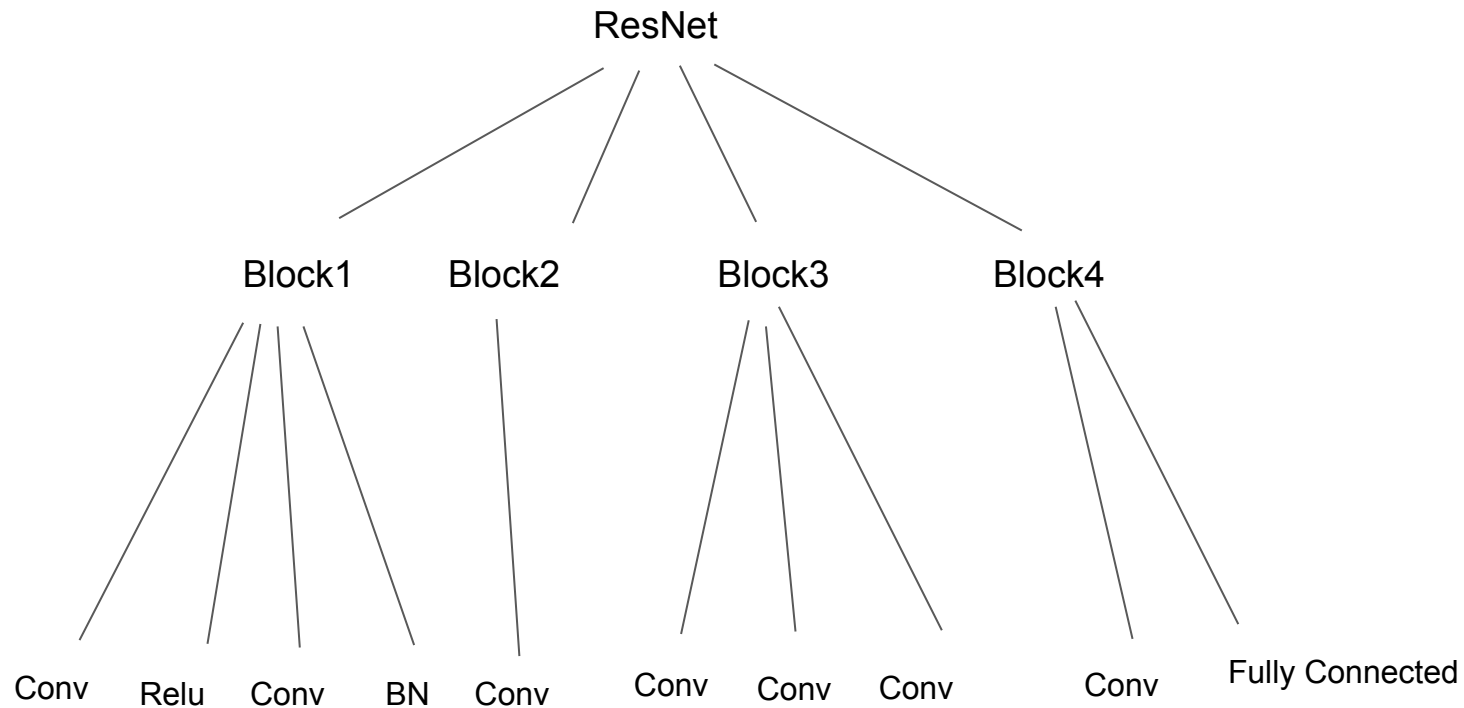
Module

```
VGG (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU (inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU (inplace)
    (4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU (inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU (inplace)
    (9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU (inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU (inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU (inplace)
    (16): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU (inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU (inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU (inplace)
    (23): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU (inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU (inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU (inplace)
    (30): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  )
  (classifier): Sequential (
    (0): Dropout (p = 0.5)
    (1): Linear (25088 -> 4096)
    (2): ReLU (inplace)
    (3): Dropout (p = 0.5)
    (4): Linear (4096 -> 4096)
    (5): ReLU (inplace)
    (6): Linear (4096 -> 1000)
  )
)
```

Module, sub-network



Module



Module, sub-network

```
# Define a resnet block
class ResnetBlock(nn.Module):
    def __init__(self, dim, padding_type, norm_layer, use_dropout, use_bias):
        super(ResnetBlock, self).__init__()
        self.conv_block = self.build_conv_block(dim, padding_type, norm_layer, use_dropout, use_bias)

    def build_conv_block(self, dim, padding_type, norm_layer, use_dropout, use_bias):
        conv_block = []
        p = 0
        if padding_type == 'reflect':
            conv_block += [nn.ReflectionPad2d(1)]
        elif padding_type == 'replicate':
            conv_block += [nn.ReplicationPad2d(1)]
        elif padding_type == 'zero':
            p = 1
        else:
            raise NotImplementedError('padding [%s] is not implemented' % padding_type)

        conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p, bias=use_bias),
                       norm_layer(dim),
                       nn.ReLU(True)]

        if use_dropout:
            conv_block += [nn.Dropout(0.5)]

        p = 0
        if padding_type == 'reflect':
            conv_block += [nn.ReflectionPad2d(1)]
        elif padding_type == 'replicate':
            conv_block += [nn.ReplicationPad2d(1)]
        elif padding_type == 'zero':
            p = 1
        else:
            raise NotImplementedError('padding [%s] is not implemented' % padding_type)
        conv_block += [nn.Conv2d(dim, dim, kernel_size=3, padding=p, bias=use_bias),
                       norm_layer(dim)]

        return nn.Sequential(*conv_block)

    def forward(self, x):
        out = x + self.conv_block(x)
        return out
```

```
class ResnetGenerator(nn.Module):
    def __init__(self, input_nc, output_nc, ngf=64, norm_layer=nn.BatchNorm2d, use_dropout=False, n_blocks=6, gpu_ids=[], padding_type='ref
assert(n_blocks >= 0)
super(ResnetGenerator, self).__init__()
self.input_nc = input_nc
self.output_nc = output_nc
self.ngf = ngf
self.gpu_ids = gpu_ids
if type(norm_layer) == functools.partial:
    use_bias = norm_layer.func == nn.InstanceNorm2d
else:
    use_bias = norm_layer == nn.InstanceNorm2d

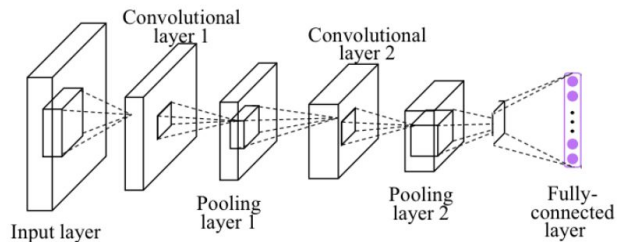
model = [nn.ReflectionPad2d(3),
         nn.Conv2d(input_nc, ngf, kernel_size=7, padding=0,
                   bias=use_bias),
         norm_layer(ngf),
         nn.ReLU(True)]

n_downsampling = 2
for i in range(n_downsampling):
    mult = 2**i
    model += [nn.Conv2d(ngf * mult, ngf * mult * 2, kernel_size=3,
                       stride=2, padding=1, bias=use_bias),
             norm_layer(ngf * mult * 2),
             nn.ReLU(True)]

mult = 2**n_downsampling
for i in range(n_blocks):
    model += [ResnetBlock(ngf * mult, padding_type=padding_type, norm_layer=norm_layer, use_dropout=use_dropout, use_bias=use_bias)]

for i in range(n_downsampling):
    mult = 2**(n_downsampling - i)
    model += [nn.ConvTranspose2d(ngf * mult, int(ngf * mult / 2),
                                kernel_size=3, stride=2,
```

Module, network



```
class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # 320 -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

```
Class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()

        layer_list = [
            nn.Conv2d(1,10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10,20, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU()
        ]

        self.sub_model = nn.Sequential(*layer_list)

        self.fc = nn.Linear(320, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = self.sub_model(x)

        x = x.view(in_size, -1)
        x = self.fc(x)

        return F.log_softmax(x)
```

PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray,
but runs on GPU

Variable: Node in a
computational graph; stores
data and gradient

Module: A neural network
layer; may store state or
learnable weights

Data Format (Tensor)



Batch Channel

(N, C, H, W)

Most of the layers

(6, 3, 32, 32)

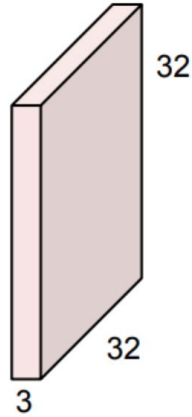
(N, C)

Fully Connected Layer

Data Format (Tensor)

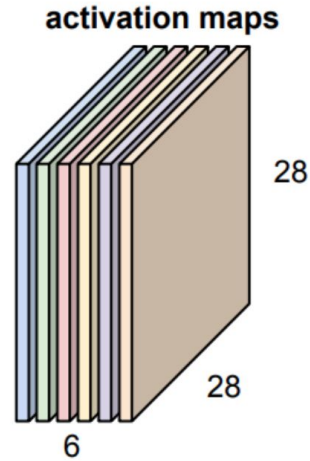
Conv

(1, 3, 32, 32)



Convolution Layer

(1, 6, 28, 28)



In pytorch, `conv2 = nn.Conv2d(3, 6, kernel_size=5)`

Data Format (Tensor)



Batch Channel

(N, C, H, W)

Most of the layers

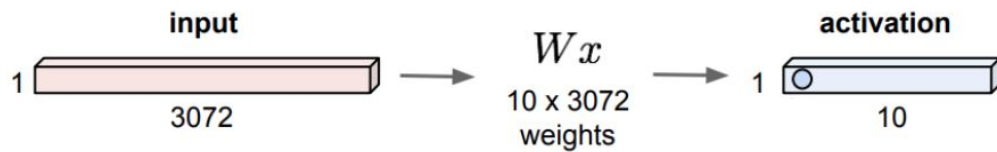
(6, 3, 24, 24)

(N, C)

Fully Connected Layer

Data Format (Tensor)

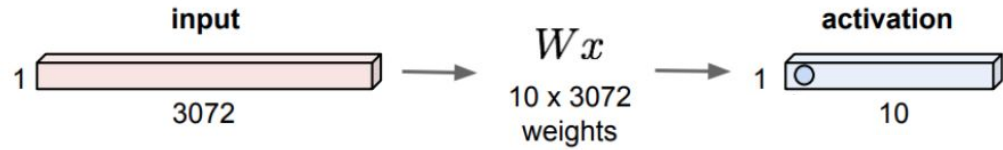
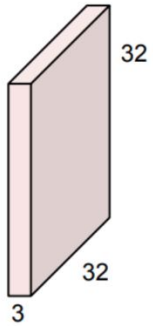
Fully Connected Layer



```
In pytorch, fc = nn.Linear(3072, 10)
```


Data Format (Tensor)

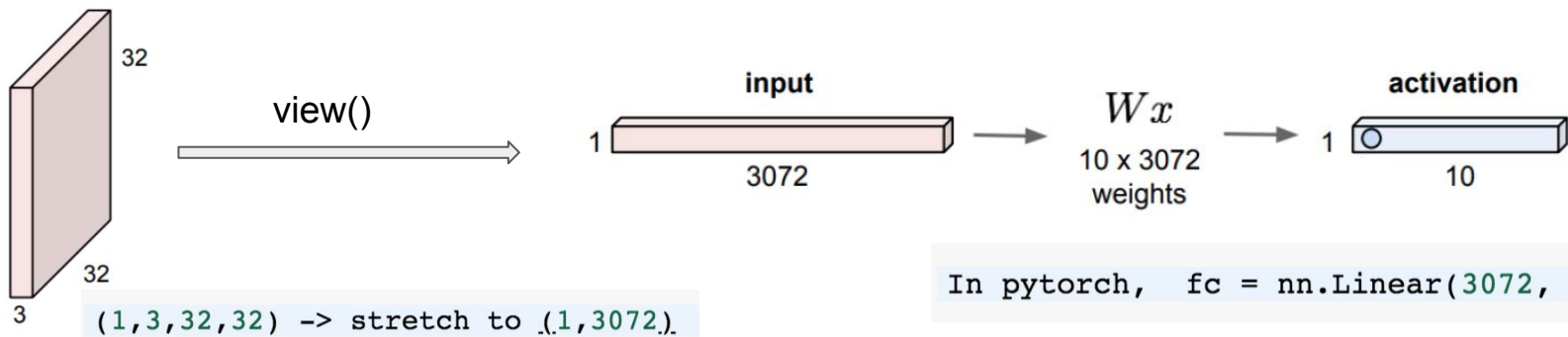
Fully Conv



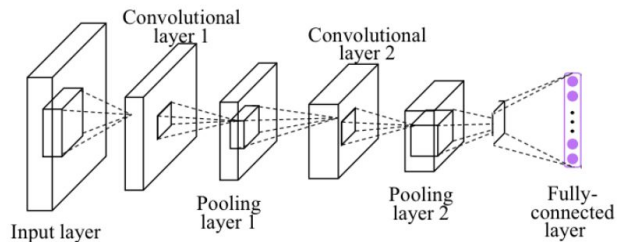
```
In pytorch, fc = nn.Linear(3072, 10)
```

Data Format (Tensor)

Fully Conv



Module, network



```
class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # 320 -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

```
Class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()

        layer_list = [
            nn.Conv2d(1,10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10,20, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU()
        ]

        self.sub_model = nn.Sequential(*layer_list)

        self.fc = nn.Linear(320, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = self.sub_model(x)

        x = x.view(in_size, -1)
        x = self.fc(x)

        return F.log_softmax(x)
```

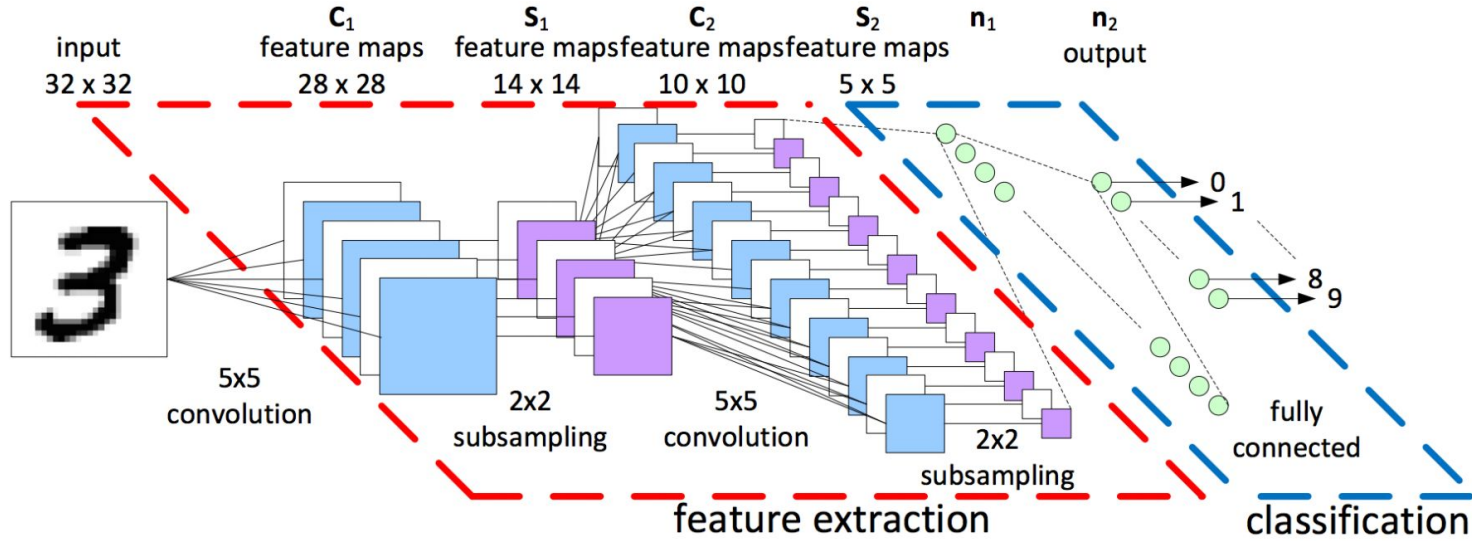
PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray,
but runs on GPU

Variable: Node in a
computational graph; stores
data and gradient

Module: A neural network
layer; may store state or
learnable weights

Train a simple Network



1. Forward: compute output of each layer

L

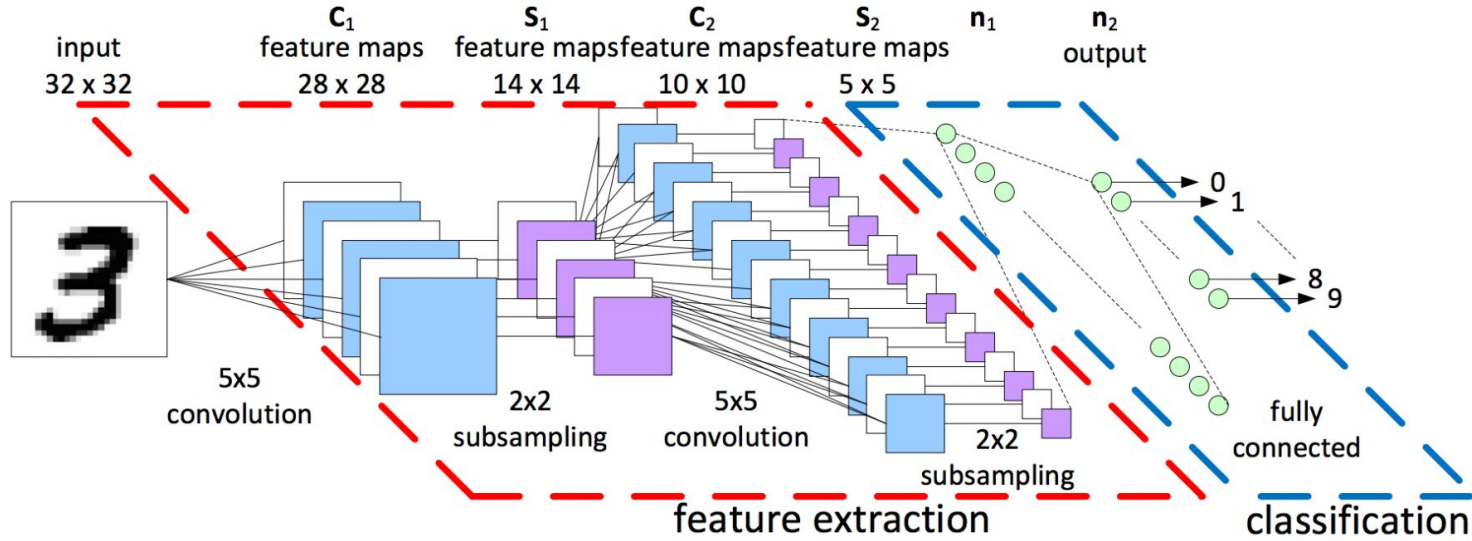
2. Backward: compute gradient

$\frac{\partial L}{\partial w}$

3. Update: update the parameters with computed gradient

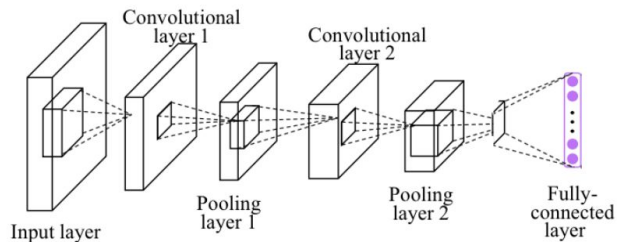
$w = w - \alpha * \frac{\partial L}{\partial w}$

Train a simple Network



1. Forward: compute output of each layer -> Network (Module), Loss L
2. Backward: compute gradient -> Variable (gradient)
3. Update: update the parameters with computed gradient

Module, network



```
class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # 320 -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

```
Class Net(nn.Module):
```

```
    def __init__(self):
        super(Net, self).__init__()

        layer_list = [
            nn.Conv2d(1,10, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(10,20, kernel_size=5),
            nn.MaxPool2d(2),
            nn.ReLU()
        ]

        self.sub_model = nn.Sequential(*layer_list)

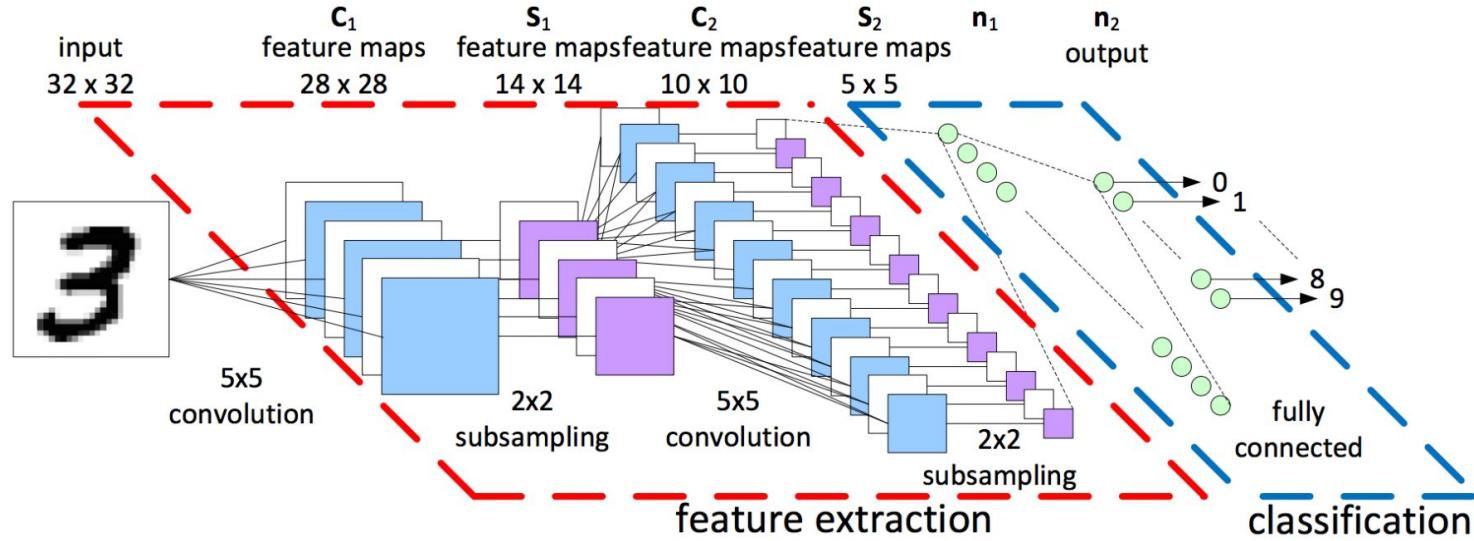
        self.fc = nn.Linear(320, 10)

    def forward(self, x):
        in_size = x.size(0)
        x = self.sub_model(x)

        x = x.view(in_size, -1)
        x = self.fc(x)

        return F.log_softmax(x)
```

Train a simple Network



1. Forward: compute output of each layer -> Network (Module), Loss
2. Backward: compute gradient -> Variable (gradient)
3. Update: update the parameters with computed gradient

 L $\frac{\partial L}{\partial w}$

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

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

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

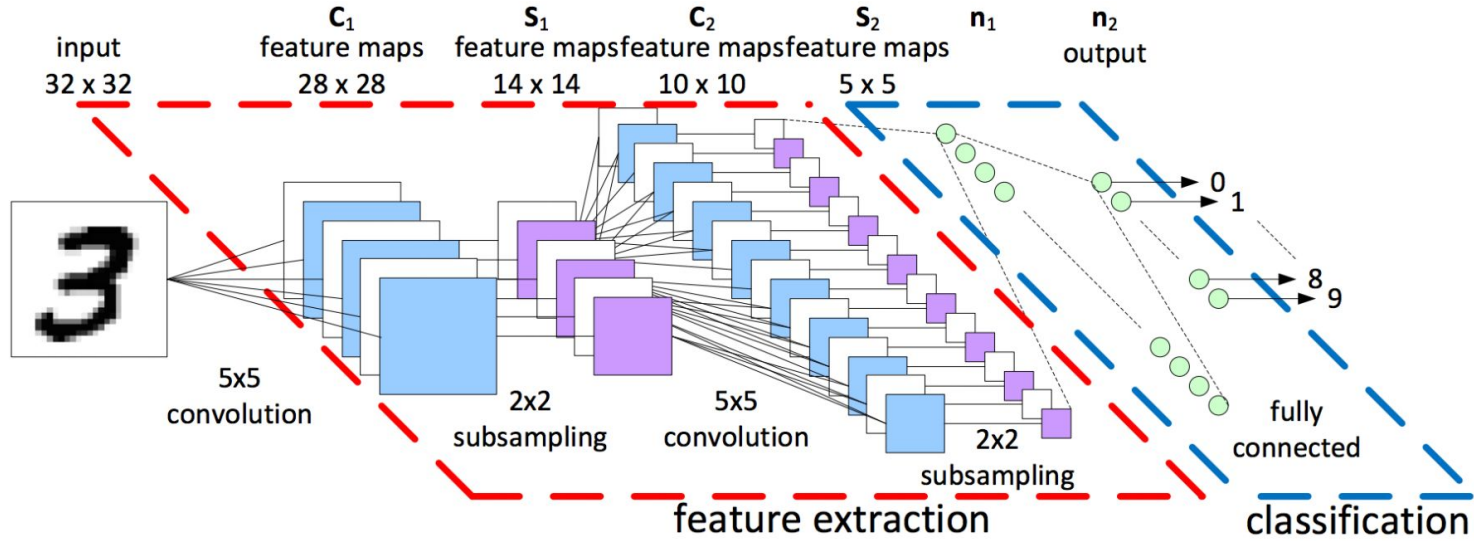
c.backward()

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

Variable

Compute gradient automatically

Train a simple Network



1. Forward: compute output of each layer -> Network (Module), Loss L
2. Backward: compute gradient -> Variable (gradient) $\frac{\partial L}{\partial w}$
3. Update: update the parameters with computed gradient $w = w - \alpha * \frac{\partial L}{\partial w}$

Optimizer

```
optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.momentum).
```

Optimizer

```
net = model()  
optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.momentum)
```

```
### forward
```

```
prob = net(input_data)  
loss = cross_entropy_loss(prob, groundtruth)
```

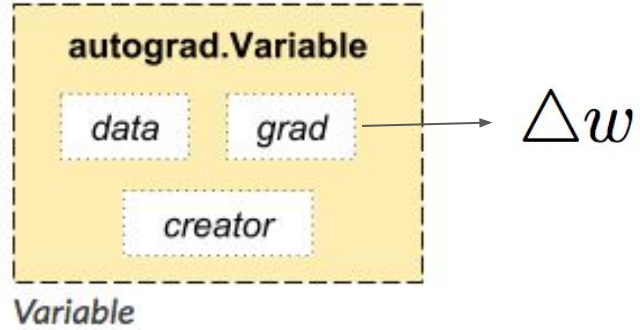
```
### backward
```

```
optimizer.zero_grad()  
loss.backward() →  $\frac{\partial L}{\partial w}$ 
```

```
### update
```

```
optimizer.step() →  $w = w - \alpha * \frac{\partial L}{\partial w}$ 
```

Variable



Optimizer

```
net = model()  
optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.momentum)
```


```
### forward
```

```
prob = net(input_data)  
loss = cross_entropy_loss(prob, groundtruth)
```

```
### backward
```


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

$$\Delta w = 0$$


$$\Delta w \leftarrow \Delta w + \frac{\partial L}{\partial w}$$

```
### update
```

```
optimizer.step()
```


$$w = w - \alpha * \Delta w$$

Optimizer

```
net = model()  
optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.momentum)
```

```
### forward
```

```
prob = net(input_data)  
loss = cross_entropy_loss(prob, groundtruth)
```

```
### backward  $\Delta w \leftarrow \Delta w + \frac{\partial L}{\partial w}$   
loss.backward()
```

```
optimizer.zero_grad(.)  $\Delta w = 0$ 
```

```
### update
```

```
optimizer.step()  $w = w - \alpha * \Delta w$ 
```

Optimizer

```
net = model()  
optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.momentum)
```


```
### forward
```

```
prob = net(input_data)  
loss = cross_entropy_loss(prob, groundtruth)
```

```
### backward
```


```
optimizer.zero_grad()
```

```
loss.backward()
```


$$\Delta w \leftarrow \Delta w + \frac{\partial L}{\partial w}$$

```
### update
```

```
optimizer.step()
```


$$w = w - \alpha * \Delta w$$

Optimizer

```
net = model()  
optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.momentum)
```


```
### forward
```

```
prob = net(input_data)  
loss = cross_entropy_loss(prob, groundtruth)
```

```
### backward
```


```
optimizer.zero_grad()
```

```
loss.backward()
```


$$\Delta w \leftarrow \Delta w + \frac{\partial L}{\partial w}$$

```
### update
```

```
optimizer.step()
```


$$w = w - \alpha * \Delta w$$

Δw

Init

0

Iteration1

$$\frac{\partial L_1}{\partial w}$$

Iteration2

$$\frac{\partial L_1}{\partial w} + \frac{\partial L_2}{\partial w}$$

Optimizer

```
net = model()  
optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.momentum)
```


```
### forward
```

```
prob = net(input_data)  
loss = cross_entropy_loss(prob, groundtruth)
```

```
### backward
```


```
optimizer.zero_grad()
```

```
loss.backward()
```


$$\Delta w \leftarrow \Delta w + \frac{\partial L}{\partial w}$$

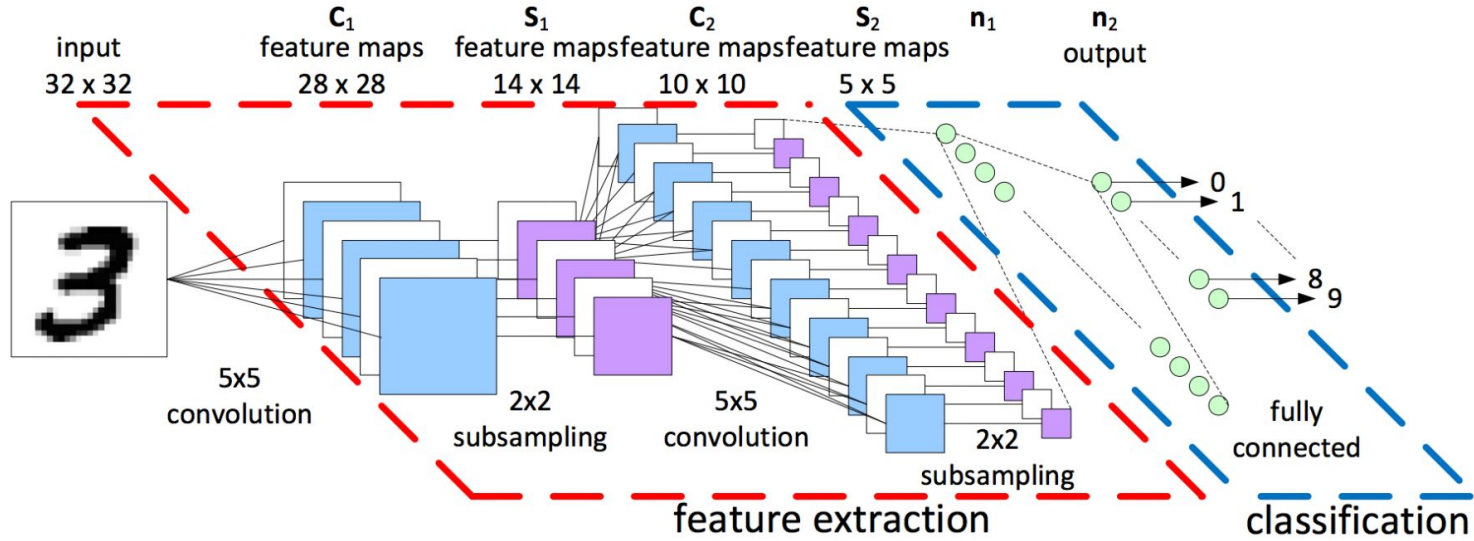
```
### update
```

```
optimizer.step()
```


$$w = w - \alpha * \Delta w$$

	Init	Iteration1	Iteration2
Δw	0	$\frac{\partial L_1}{\partial w}$	$\frac{\partial L_1}{\partial w} + \frac{\partial L_2}{\partial w}$
Δw	0	$\frac{\partial L_1}{\partial w}$	$\frac{\partial L_2}{\partial w}$

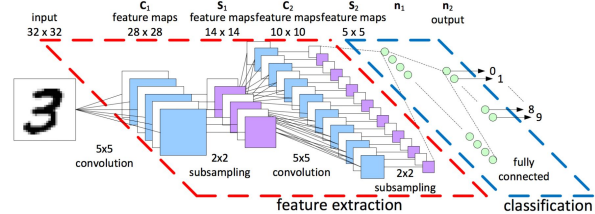
Train a simple Network



1. Forward: compute output of each layer -> Network (Module), Loss L
2. Backward: compute gradient -> Variable (gradient) $\frac{\partial L}{\partial w}$
3. Update: update the parameters with computed gradient -> Optimizer $w = w - \alpha * \frac{\partial L}{\partial w}$

Data Loading

Data Loading



Batch (batch size)

```
# Training cycle
for epoch in range(training_epochs):
    # Loop over all batches
    for i in range(total_batch):
```



In the neural network terminology:

288



- one **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- number of **iterations** = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Data Loading

Custom DataLoader

```
class DiabetesDataset(Dataset):  
    """ Diabetes dataset. """  
  
    # Initialize your data, download, etc.  
    def __init__(self):  
        1 download, read data, etc.  
  
    def __getitem__(self, index):  
        2 return one item on the index  
        return  
  
    def __len__(self):  
        3 return the data length  
        return  
  
dataset = DiabetesDataset()  
train_loader = DataLoader(dataset=dataset,  
                           batch_size=32,  
                           shuffle=True,  
                           num_workers=2)
```

Data Loading

Custom DataLoader

```
class DiabetesDataset(Dataset):
    """ Diabetes dataset. """

    # Initialize your data, download, etc.
    def __init__(self):
        xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, 0:-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

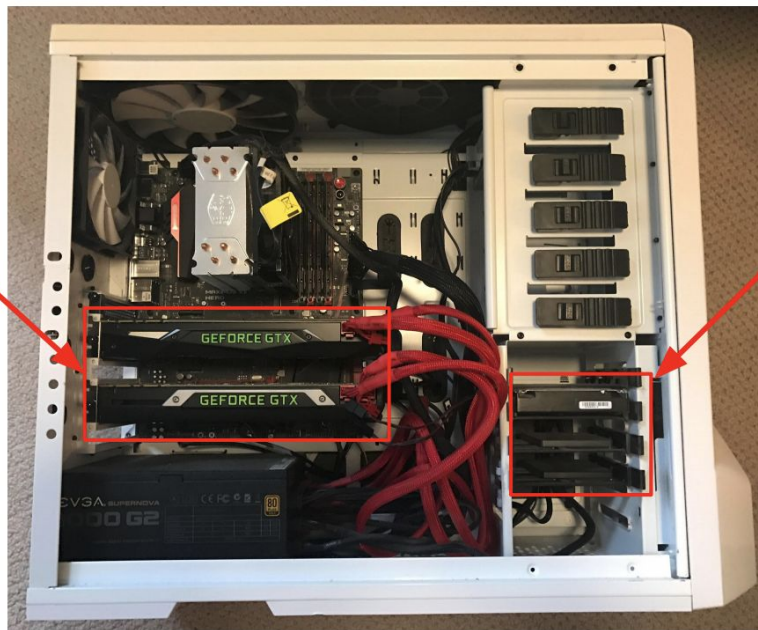
    def __len__(self):
        return self.len

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

Data Loading

CPU / GPU Communication

Model
is here



Data is here

If you aren't careful, training can bottleneck on reading data and transferring to GPU!

Solutions:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data

Data Loading

Using DataLoader

```
dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset, batch_size=32, shuffle=True, num_workers=2)

# Training Loop
for epoch in range(2):
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = Variable(inputs), Variable(labels)

        # Forward pass: Compute predicted y by passing x to the model
        y_pred = model(inputs)

        # Compute and print Loss
        loss = criterion(y_pred, labels)
        print(epoch, i, loss.data[0])

        # Zero gradients, perform a backward pass, and update the weights.
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Data Processing

```
37 train_loader = torch.utils.data.DataLoader(  
38     datasets.MNIST('../data', train=True, download=True,  
39         transform=transforms.Compose([  
40             transforms.ToTensor(),  
41             transforms.Normalize((0.1307,), (0.3081,))  
42         ])),  
43     batch_size=args.batch_size, shuffle=True, **kwargs)  
44 test_loader = torch.utils.data.DataLoader(  
45     datasets.MNIST('../data', train=False, transform=transforms.Compose([  
46         transforms.ToTensor(),  
47         transforms.Normalize((0.1307,), (0.3081,))  
48     ])),  
49     batch_size=args.test_batch_size, shuffle=True, **kwargs)
```

Data Processing

Pix2pix Code

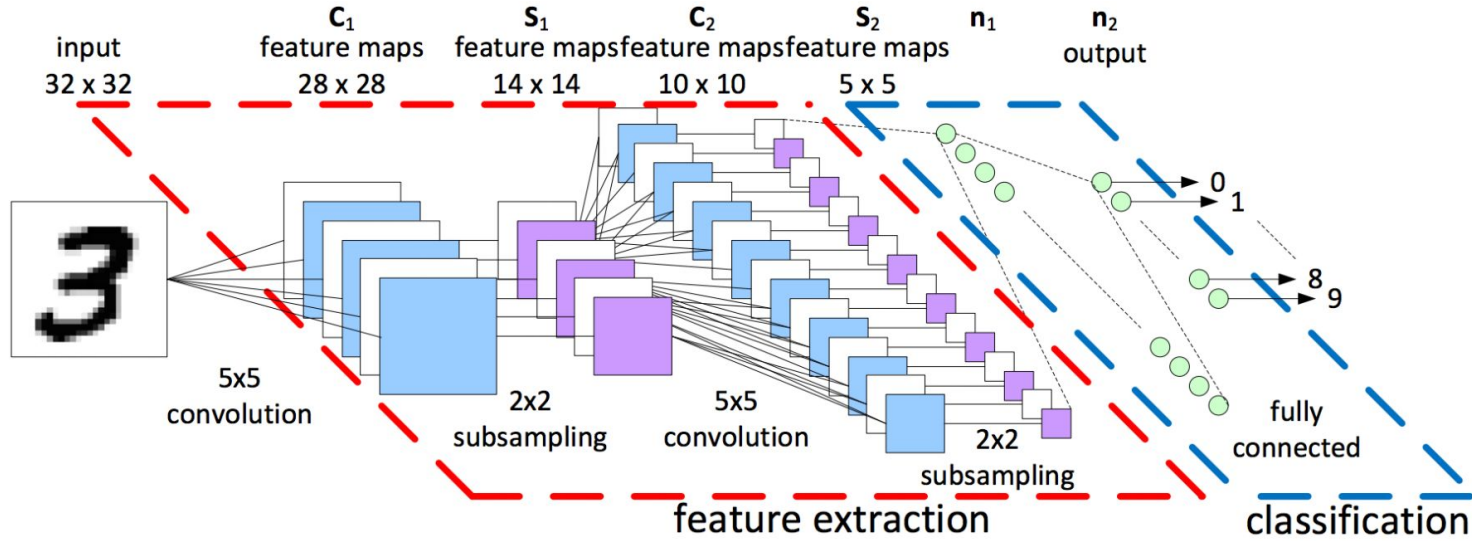
```
def get_transform(opt):
    transform_list = []
    if opt.resize_or_crop == 'resize_and_crop':
        osize = [opt.loadSize, opt.loadSize]
        transform_list.append(transforms.Scale(osize, Image.BICUBIC))
        transform_list.append(transforms.RandomCrop(opt.fineSize))
    elif opt.resize_or_crop == 'crop':
        transform_list.append(transforms.RandomCrop(opt.fineSize))
    elif opt.resize_or_crop == 'scale_width':
        transform_list.append(transforms.Lambda(
            lambda img: __scale_width(img, opt.fineSize)))
    elif opt.resize_or_crop == 'scale_width_and_crop':
        transform_list.append(transforms.Lambda(
            lambda img: __scale_width(img, opt.loadSize)))
        transform_list.append(transforms.RandomCrop(opt.fineSize))

    if opt.isTrain and not opt.no_flip:
        transform_list.append(transforms.RandomHorizontalFlip())

    transform_list += [transforms.ToTensor(),
                      transforms.Normalize((0.5, 0.5, 0.5),
                                           (0.5, 0.5, 0.5))]
    return transforms.Compose(transform_list)

self.transform = get_transform(opt)
```

Train a simple Network



1. Forward: compute output of each layer -> Network (Module), Loss L
2. Backward: compute gradient -> Variable (gradient) $\frac{\partial L}{\partial w}$
3. Update: update the parameters with computed gradient -> Optimizer $w = w - \alpha * \frac{\partial L}{\partial w}$

MNIST example

pytorch / examples

Watch 177 Star 3,073 Fork 1,315

Code Issues 43 Pull requests 18 Projects 0 Wiki Insights

Branch: master examples / mnist / main.py Find file Copy path

rdinse Change test DataLoader to use the test batch size 930ae27 on Sep 5, 2017

9 contributors

114 lines (99 sloc) | 4.41 KB Raw Blame History

```
1 from __future__ import print_function
2 import argparse
3 import torch
4 import torch.nn as nn
5 import torch.nn.functional as F
6 import torch.optim as optim
7 from torchvision import datasets, transforms
8 from torch.autograd import Variable
9
10 # Training settings
11 parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
12 parser.add_argument('--batch-size', type=int, default=64, metavar='N',
13                     help='input batch size for training (default: 64)')
14 parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
15                     help='input batch size for testing (default: 1000)')
16 parser.add_argument('--epochs', type=int, default=10, metavar='N',
17                     help='number of epochs to train (default: 10)')
18 parser.add_argument('--seed', type=int, default=1, metavar='N',
19                     help='random seed (default: 1)')
```

Others

For training:

```
net.train()
```

For testing:

```
net.eval()
```

eval() mode:

*Dropout Layer

*Batchnorm Layer

Difference between frameworks

The hyper-parameters for Caffe may not be good for Pytorch.

Difference between versions

1. Pytorch-0.3.1
2. Pytorch-0.4
Tensor and Variable are merged
3. Pytorch-1.0


```
import torch
from torch.autograd import Variable

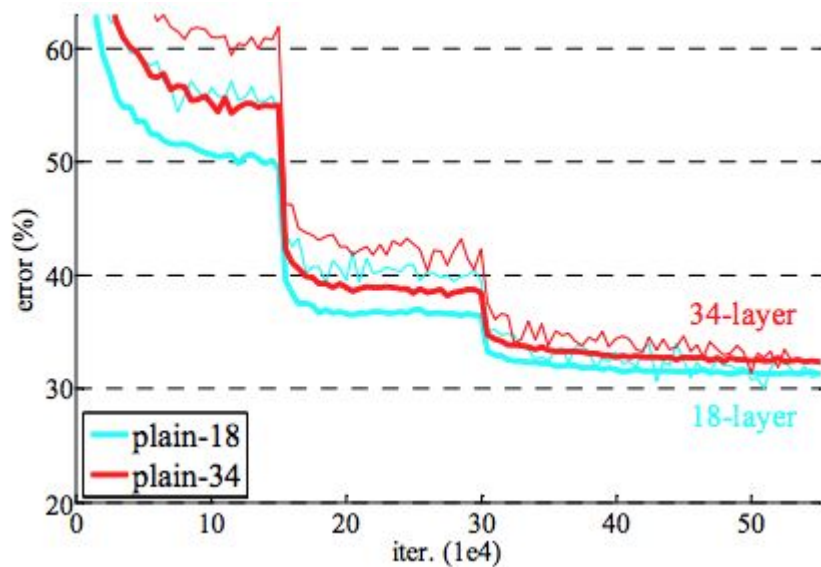
N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

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

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

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


Learning Rate Scheduler



Learning Rate Scheduler

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
scheduler = ReduceLROnPlateau(optimizer, 'min') # set up scheduler
for epoch in range(10):
    train(...)
    val_acc, val_loss = validate(...)
    scheduler.step(val_loss, epoch) # update lr if needed
```

torch.optim.lr_scheduler

- StepLR: LR is delayed by gamma every step_size epochs
- MultiStepLR: LR is delayed by gamma once the number of epoch reaches milestones.
- ExponentialLR
- CosineAnnealingLR
- ReduceLROnPlateau

Pretrained Model

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
squeezenet = models.squeezenet1_0(pretrained=True)
vgg16 = models.vgg16(pretrained=True)
densenet = models.densenet161(pretrained=True)
inception = models.inception_v3(pretrained=True)
```


- Documentation
 - Available models
 - AlexNet
 - BNInception
 - DenseNet121
 - DenseNet161
 - DenseNet169
 - DenseNet201
 - FBResNet152
 - InceptionResNetV2
 - InceptionV3
 - InceptionV4
 - NASNet-A-Large
 - ResNeXt101_32x4d
 - ResNeXt101_64x4d
 - ResNet101
 - ResNet152
 - ResNet18
 - ResNet34
 - ResNet50
 - SqueezeNet1_0
 - SqueezeNet1_1
 - VGG11
 - VGG13
 - VGG16
 - VGG19
 - VGG11_BN
 - VGG13_BN
 - VGG16_BN
 - VGG19_BN

Load Model

args.resume is the path to the trained model

```
if os.path.isfile(args.resume):  
    print("=> loading checkpoint '{}'.format(args.resume))  
    checkpoint = torch.load(args.resume)  
    args.start_epoch = checkpoint['epoch']  
    best_prec1 = checkpoint['best_prec1']  
    model.load_state_dict(checkpoint['state_dict'])  
    print("=> loaded checkpoint '{}' (epoch {})"  
          .format(args.evaluate, checkpoint['epoch']))
```

Define the model
before loading parameters



Weights Initialization

```
from torch.nn import init

def weights_init_normal(m):
    classname = m.__class__.__name__
    # print(classname)
    if classname.find('Conv') != -1:
        init.normal(m.weight.data, 0.0, 0.02)
    elif classname.find('Linear') != -1:
        init.normal(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm2d') != -1:
        init.normal(m.weight.data, 1.0, 0.02)
        init.constant(m.bias.data, 0.0)

net.apply(weights_init_normal)
```

Weights Initialization

```
def weights_init_xavier(m):
    classname = m.__class__.__name__
    # print(classname)
    if classname.find('Conv') != -1:
        init.xavier_normal(m.weight.data, gain=0.02)
    elif classname.find('Linear') != -1:
        init.xavier_normal(m.weight.data, gain=0.02)
    elif classname.find('BatchNorm2d') != -1:
        init.normal(m.weight.data, 1.0, 0.02)
        init.constant(m.bias.data, 0.0)
```

```
def weights_init_orthogonal(m):
    classname = m.__class__.__name__
    print(classname)
    if classname.find('Conv') != -1:
        init.orthogonal(m.weight.data, gain=1)
    elif classname.find('Linear') != -1:
        init.orthogonal(m.weight.data, gain=1)
    elif classname.find('BatchNorm2d') != -1:
        init.normal(m.weight.data, 1.0, 0.02)
        init.constant(m.bias.data, 0.0)
```

```
def weights_init_kaiming(m):
    classname = m.__class__.__name__
    # print(classname)
    if classname.find('Conv') != -1:
        init.kaiming_normal(m.weight.data, a=0, mode='fan_in')
    elif classname.find('Linear') != -1:
        init.kaiming_normal(m.weight.data, a=0, mode='fan_in')
    elif classname.find('BatchNorm2d') != -1:
        init.normal(m.weight.data, 1.0, 0.02)
        init.constant(m.bias.data, 0.0)
```


Hooks

It is used to inspect or modify the output and grad of a layer

Need to write register function

Hooks (Forward)

```
def printnorm(self, input, output):  
    # input is a tuple of packed inputs  
    # output is a Variable. output.data is the Tensor we are interested  
    print('Inside ' + self.__class__.__name__ + ' forward')  
    print('')  
    print('input: ', type(input))  
    print('input[0]: ', type(input[0]))  
    print('output: ', type(output))  
    print('')  
    print('input size:', input[0].size())  
    print('output size:', output.data.size())  
    print('output norm:', output.data.norm())  
  
net.conv2.register_forward_hook(printnorm)  
  
out = net(input)
```

Out:

```
Inside Conv2d forward  
  
input: <class 'tuple'>  
input[0]: <class 'torch.autograd.variable.Variable'>  
output: <class 'torch.autograd.variable.Variable'>  
  
input size: torch.Size([1, 10, 12, 12])  
output size: torch.Size([1, 20, 8, 8])  
output norm: 13.015911962493094
```

Hooks (Backward)

```
def printgradnorm(self, grad_input, grad_output):
    print('Inside ' + self.__class__.__name__ + ' backward')
    print('Inside class:' + self.__class__.__name__)
    print('')
    print('grad_input: ', type(grad_input))
    print('grad_input[0]: ', type(grad_input[0]))
    print('grad_output: ', type(grad_output))
    print('grad_output[0]: ', type(grad_output[0]))
    print('')
    print('grad_input size:', grad_input[0].size())
    print('grad_output size:', grad_output[0].size())
    print('grad_input norm:', grad_input[0].data.norm())

net.conv2.register_backward_hook(printgradnorm)

out = net(input)
err = loss_fn(out, target)
err.backward()
```

Out:

```
Inside Conv2d forward
input: <class 'tuple'>
input[0]: <class 'torch.autograd.variable.Variable'>
output: <class 'torch.autograd.variable.Variable'>

input size: torch.Size([1, 10, 12, 12])
output size: torch.Size([1, 20, 8, 8])
output norm: 13.015911962493094
Inside Conv2d backward
Inside class:Conv2d

grad_input: <class 'tuple'>
grad_input[0]: <class 'torch.autograd.variable.Variable'>
grad_output: <class 'tuple'>
grad_output[0]: <class 'torch.autograd.variable.Variable'>

grad_input size: torch.Size([1, 10, 12, 12])
grad_output size: torch.Size([1, 20, 8, 8])
grad_input norm: 0.027638580029029722
```

Cudnn.benchmark flag

```
46 print("Random Seed: ", opt.manualSeed)
47 random.seed(opt.manualSeed)
48 torch.manual_seed(opt.manualSeed)
49 if opt.cuda:
50     torch.cuda.manual_seed_all(opt.manualSeed)
51
52 cudnn.benchmark = True
53
54 if torch.cuda.is_available() and not opt.cuda:
55     print("WARNING: You have a CUDA device, so you should probably run with --cuda")
56
57 if opt.dataset in ['imagenet', 'folder', 'lfw']:
58     # folder dataset
```



fmassa  Francisco Massa

Aug '17

It enables benchmark mode in cudnn.

benchmark mode is good whenever your input sizes for your network do not vary. This way, cudnn will look for the optimal set of algorithms for that particular configuration (which takes some time). This usually leads to faster runtime.

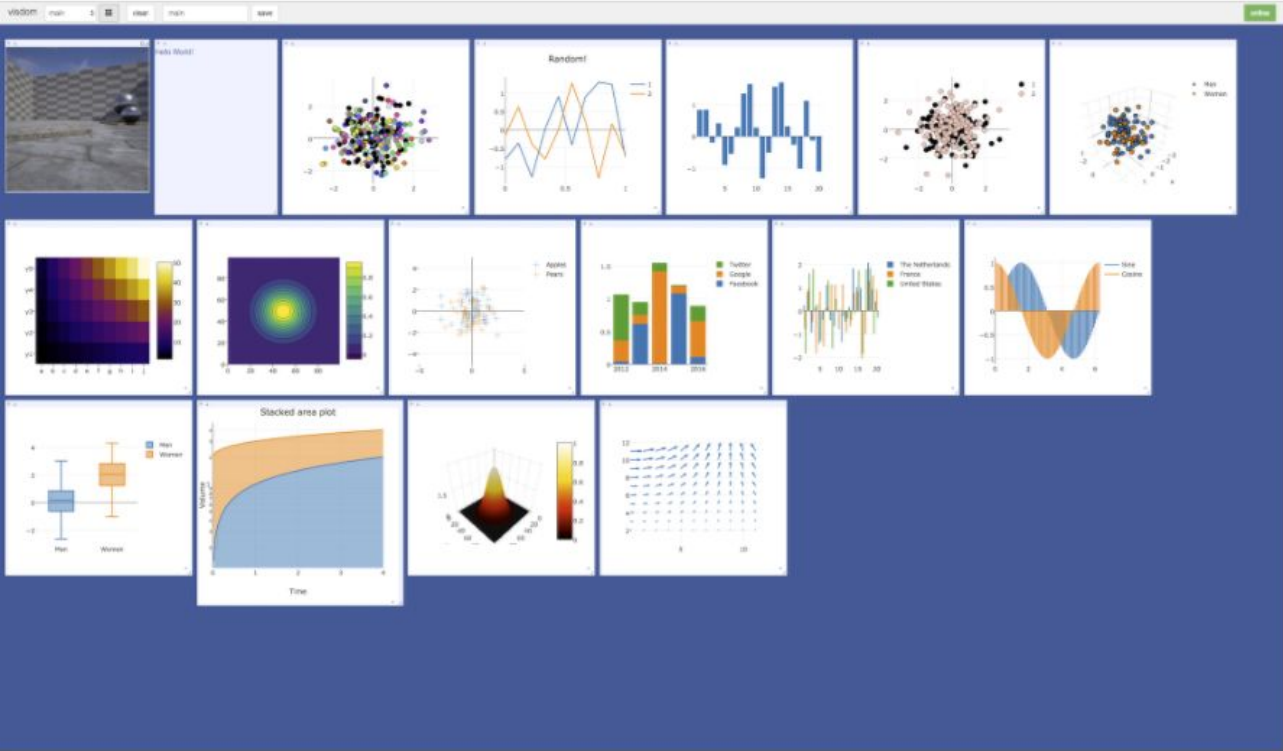
But if your input sizes changes at each iteration, then cudnn will benchmark every time a new size appears, possibly leading to worse runtime performances.

11 Likes



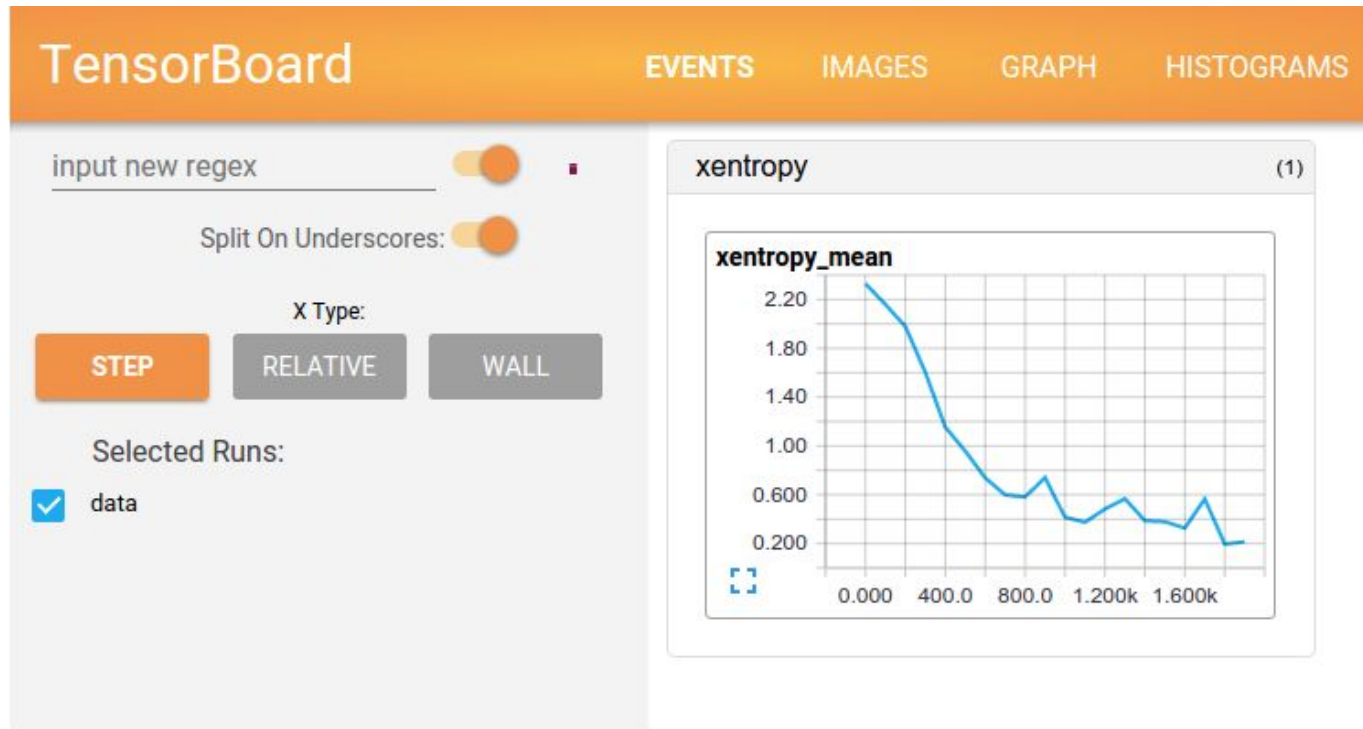
Visualization, Pytorch Visdom

<https://github.com/facebookresearch/visdom>



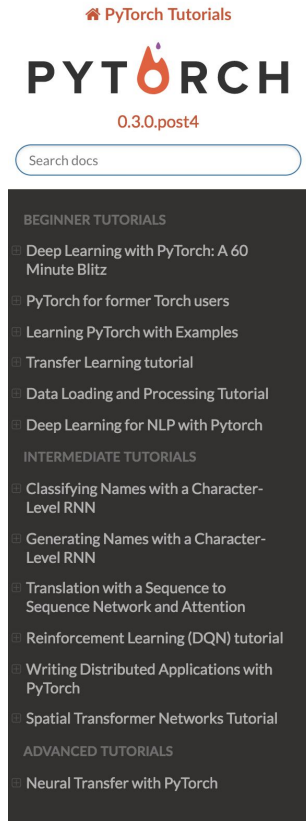
Visualization, TensorBoard

https://www.tensorflow.org/get_started/summaries_and_tensorboard



Other Resources

Official examples. <https://goo.gl/Q6Z2k8>



PyTorch Tutorials

PYTORCH

0.3.0.post4

- BEGINNER TUTORIALS
 - Deep Learning with PyTorch: A 60 Minute Blitz
 - PyTorch for former Torch users
 - Learning PyTorch with Examples
 - Transfer Learning tutorial
 - Data Loading and Processing Tutorial
 - Deep Learning for NLP with Pytorch
- INTERMEDIATE TUTORIALS
 - Classifying Names with a Character-Level RNN
 - Generating Names with a Character-Level RNN
 - Translation with a Sequence to Sequence Network and Attention
 - Reinforcement Learning (DQN) tutorial
 - Writing Distributed Applications with PyTorch
 - Spatial Transformer Networks Tutorial
- ADVANCED TUTORIALS
 - Neural Transfer with PyTorch

Docs » Welcome to PyTorch Tutorials

[View page source](#)

Welcome to PyTorch Tutorials

To get started with learning PyTorch, start with our Beginner Tutorials. The **60-minute blitz** is the most common starting point, and gives you a quick introduction to PyTorch. If you like learning by examples, you will like the tutorial [Learning PyTorch with Examples](#)

If you would like to do the tutorials interactively via IPython / Jupyter, each tutorial has a download link for a Jupyter Notebook and Python source code.

We also provide a lot of high-quality examples covering image classification, unsupervised learning, reinforcement learning, machine translation and many other applications at <https://github.com/pytorch/examples/>

You can find reference documentation for PyTorch's API and layers at <http://docs.pytorch.org> or via inline help. If you would like the tutorials section improved, please open a github issue here with your feedback: <https://github.com/pytorch/tutorials>

Beginner Tutorials

PYTORCH



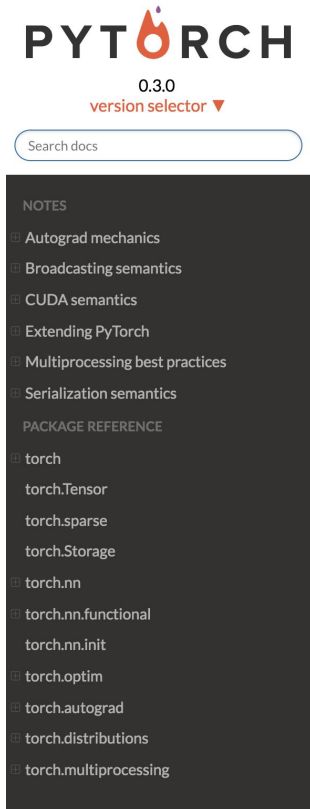
[Deep Learning with PyTorch: A 60 Minute Blitz](#)

[PyTorch for former Torch users](#)

[Learning PyTorch with Examples](#)

Other Resources

Official documents. <https://goo.gl/gecKC4>



The screenshot shows the top part of the PyTorch documentation website. At the top left is the PyTorch logo, which consists of the word "PYTORCH" in a bold, sans-serif font with a stylized orange flame above the letter "O". Below the logo, the version "0.3.0" is displayed, followed by a "version selector" dropdown menu. To the right of the version selector is a search bar with the placeholder text "Search docs". Below these elements is a dark grey sidebar containing a list of navigation links. The links are organized into sections: "NOTES" (with links to Autograd mechanics, Broadcasting semantics, CUDA semantics, Extending PyTorch, Multiprocessing best practices, and Serialization semantics), "PACKAGE REFERENCE" (with a link to torch), and a list of sub-packages under torch (torch.Tensor, torch.sparse, torch.Storage, torch.nn, torch.nn.functional, torch.nn.init, torch.optim, torch.autograd, torch.distributions, and torch.multiprocessing).

[Docs](#) » PyTorch documentation

[Edit on GitHub](#)

PyTorch documentation

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.

Notes

- [Autograd mechanics](#)
- [Broadcasting semantics](#)
- [CUDA semantics](#)
- [Extending PyTorch](#)
- [Multiprocessing best practices](#)
- [Serialization semantics](#)

Package Reference

- [torch](#)
- [torch.Tensor](#)
- [torch.sparse](#)
- [torch.Storage](#)
- [torch.nn](#)
- [torch.nn.functional](#)
- [torch.nn.init](#)
- [torch.optim](#)
- [torch.autograd](#)
- [torch.distributions](#)
- [torch.multiprocessing](#)
- [torch.distributed](#)
- [torch.legacy](#)

Other Resources

Pix2pix code <https://github.com/phillipi/pix2pix>

The screenshot shows the GitHub repository page for 'junyanz / pytorch-CycleGAN-and-pix2pix'. The repository has 122 watches, 2,995 stars, and 596 forks. The 'Code' tab is selected, showing 11 issues, 3 pull requests, 0 projects, a Wiki, and Insights. The repository description is 'Image-to-image translation in PyTorch (e.g. horse2zebra, edges2cats, and more)'. It includes tags for 'pytorch', 'gan', 'cyclegan', 'pix2pix', 'deep-learning', 'computer-vision', 'computer-graphics', 'image-manipulation', 'image-generation', and 'generative-adversarial-network'. The repository statistics show 164 commits, 1 branch, 0 releases, and 21 contributors. The 'Clone or download' button is visible. The commit history shows a merge pull request #187 from SsnL/resize_ and four recent commits: 'update aligned_dataset' (6 hours ago), 'add 'aspect_ratio' in the test code' (6 hours ago), 'add edges2cats demo' (9 months ago), and 'fix resize_ issue #170' (a day ago).

This repository Search Pull requests Issues Marketplace Explore

junyanz / pytorch-CycleGAN-and-pix2pix Watch 122 Star 2,995 Fork 596

Code Issues 11 Pull requests 3 Projects 0 Wiki Insights

Image-to-image translation in PyTorch (e.g. horse2zebra, edges2cats, and more)

pytorch gan cyclegan pix2pix deep-learning computer-vision computer-graphics image-manipulation image-generation generative-adversarial-network

164 commits 1 branch 0 releases 21 contributors


Branch: master New pull request Create new file Upload files Find file Clone or download

junyanz Merge pull request #187 from SsnL/resize_ Latest commit 7ed6fac 5 hours ago

data	update aligned_dataset	6 hours ago
datasets	add 'aspect_ratio' in the test code	6 hours ago
imgs	add edges2cats demo	9 months ago
models	fix resize_ issue #170	a day ago

Other Resources

Pytorch, Zero to All (HKUST) <https://goo.gl/S3vEUN>



ML/DL for everyone (in Korean)


Sung Kim Subscribe 17,457

Home Videos **Playlists** Community Channels About

PyTorchZeroToAll (in English)
Sung Kim • 14 videos • 11,715 views • Last updated on Dec 31, 2017
Basic ML/DL lectures using PyTorch in English.

[▶ Play all](#) [◀ Share](#) [+ Save](#)

Machine Learning
Machine needs lots of training



1 **PyTorch Lecture 01: Overview**
by Sung Kim 10:19

2 **PyTorch Lecture 02: Linear Model**
by Sung Kim 12:52

3 **PyTorch Lecture 03: Gradient Descent**
by Sung Kim 8:24

4 **PyTorch Lecture 04: Back-propagation and Autograd**
by Sung Kim 15:26

Thank You