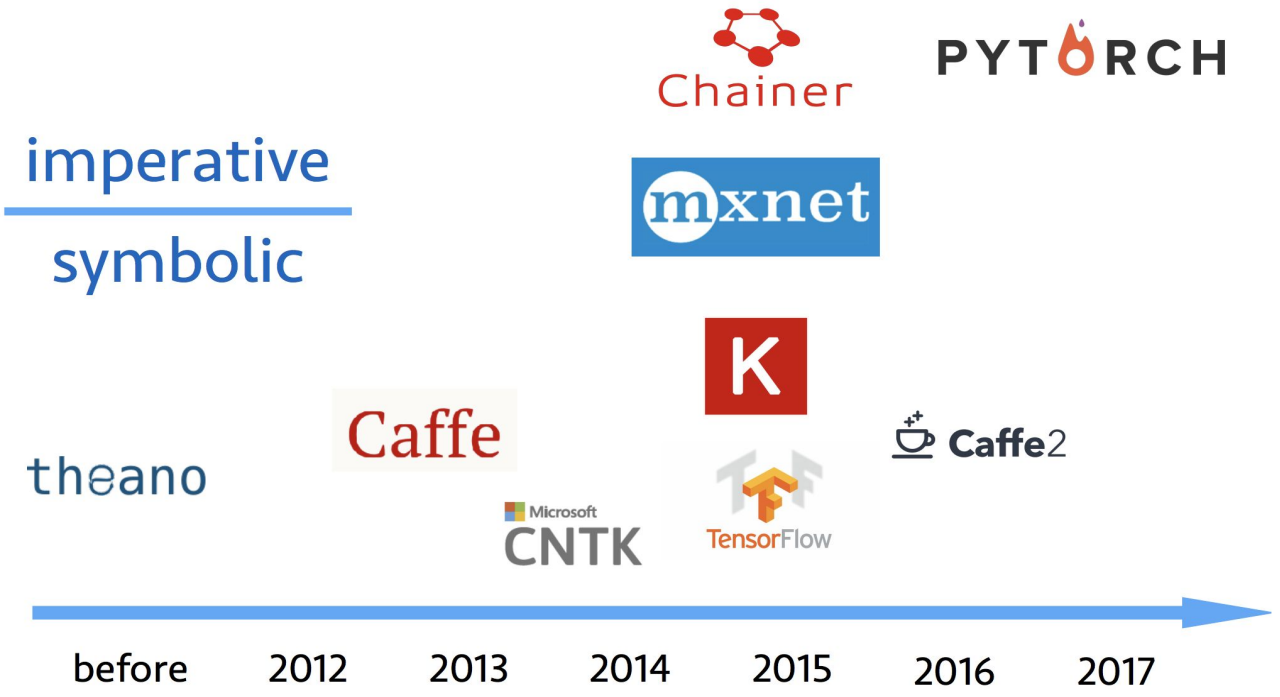
Pytorch Tutorial

Chongruo Wu

Agenda

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

Popular Deep Learning Frameworks



Gluon: new MXNet interface to accelerate research

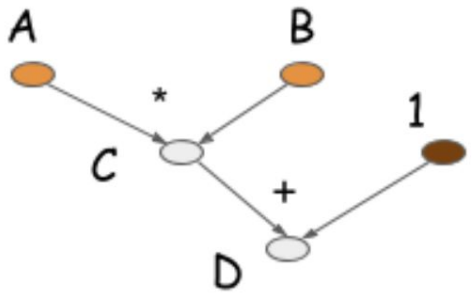
Popular Deep Learning Frameworks

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
```

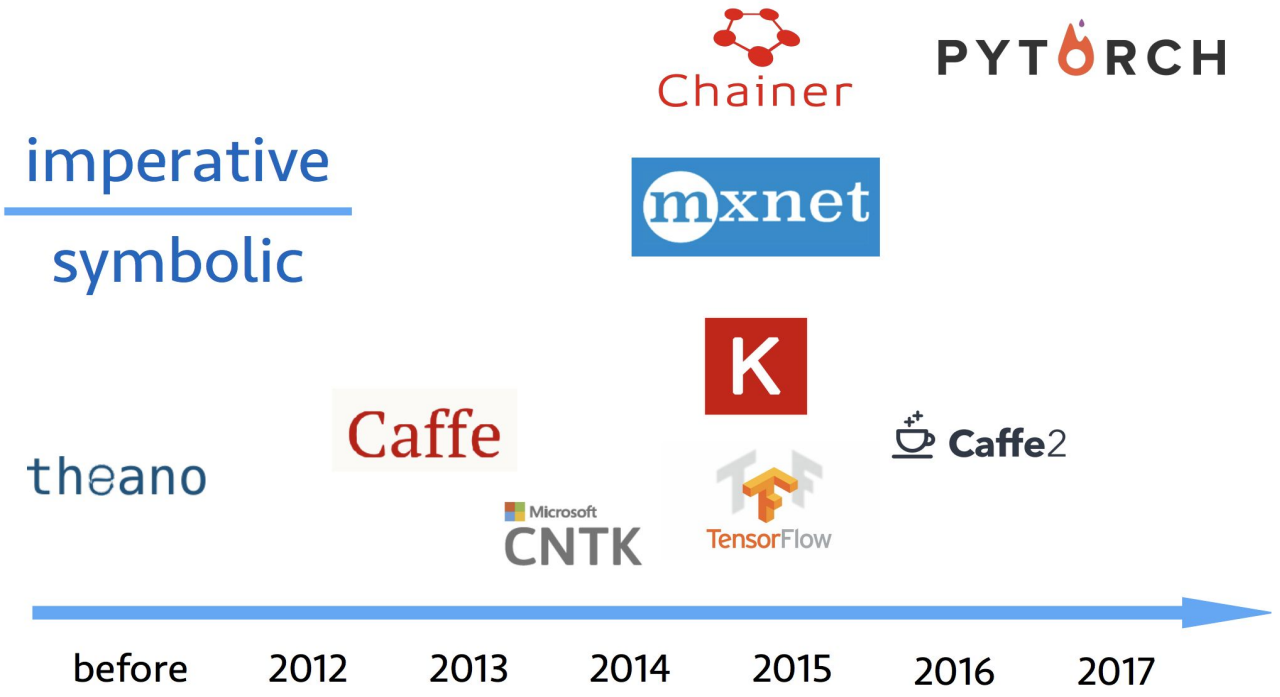
Symbolic: define the function first, then compile them

```
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)
```



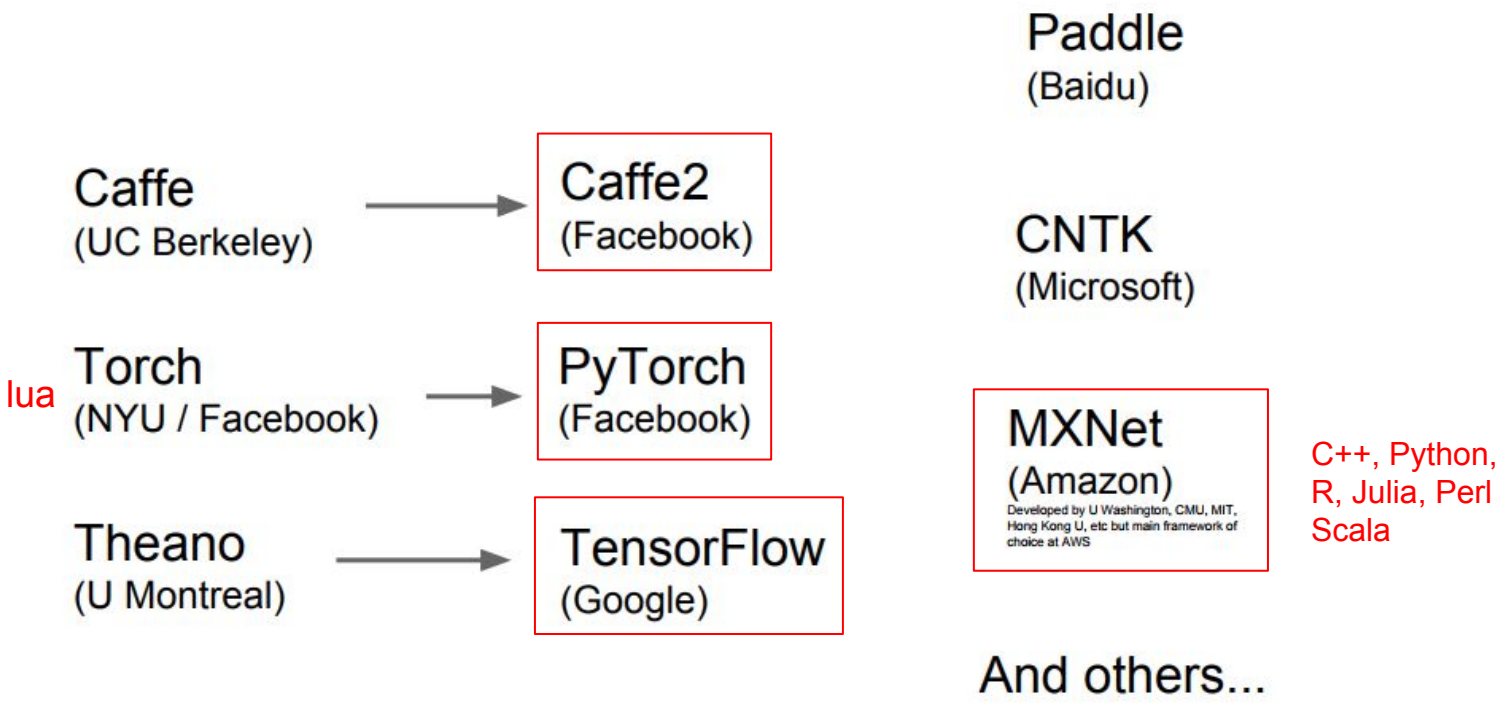
Gluon: new MXNet interface to accelerate research

Popular Deep Learning Frameworks



Gluon: new MXNet interface to accelerate research

Popular Deep Learning Frameworks



Stanford cs231n.

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

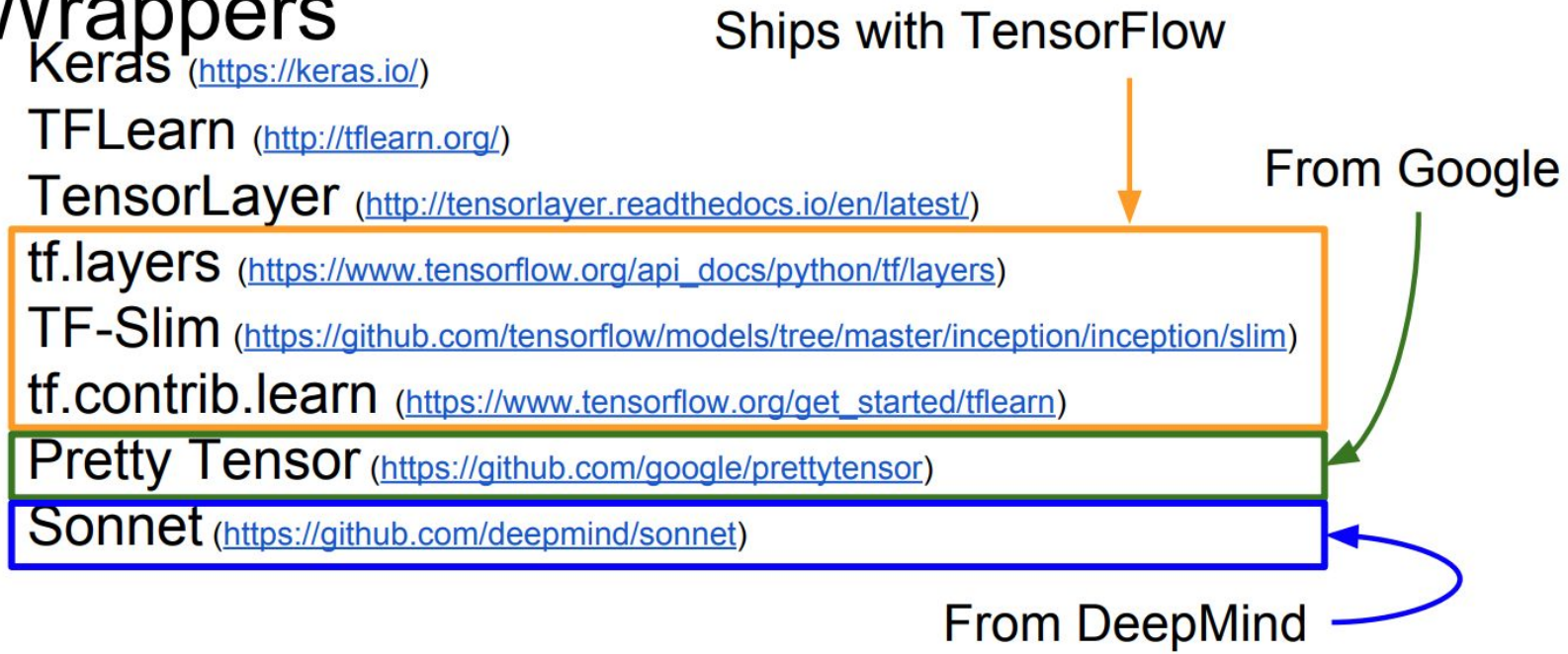
- A rich set of operators (~2000)
- The codes are not very easy to read, e.g. not python-like

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



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=Fal
act1 = sym.Activation(data=bn1, act_type='re
conv1 = sym.Convolution(data=act1, num_filte
```

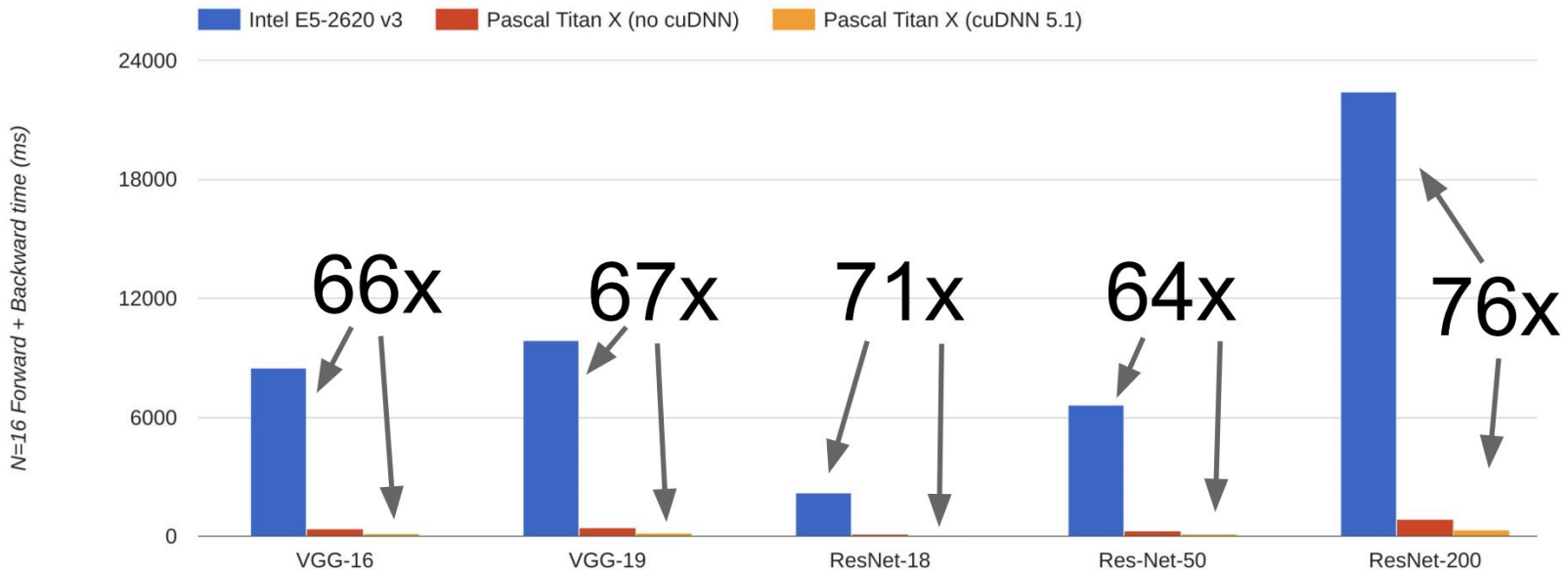
Implement Adam

```
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

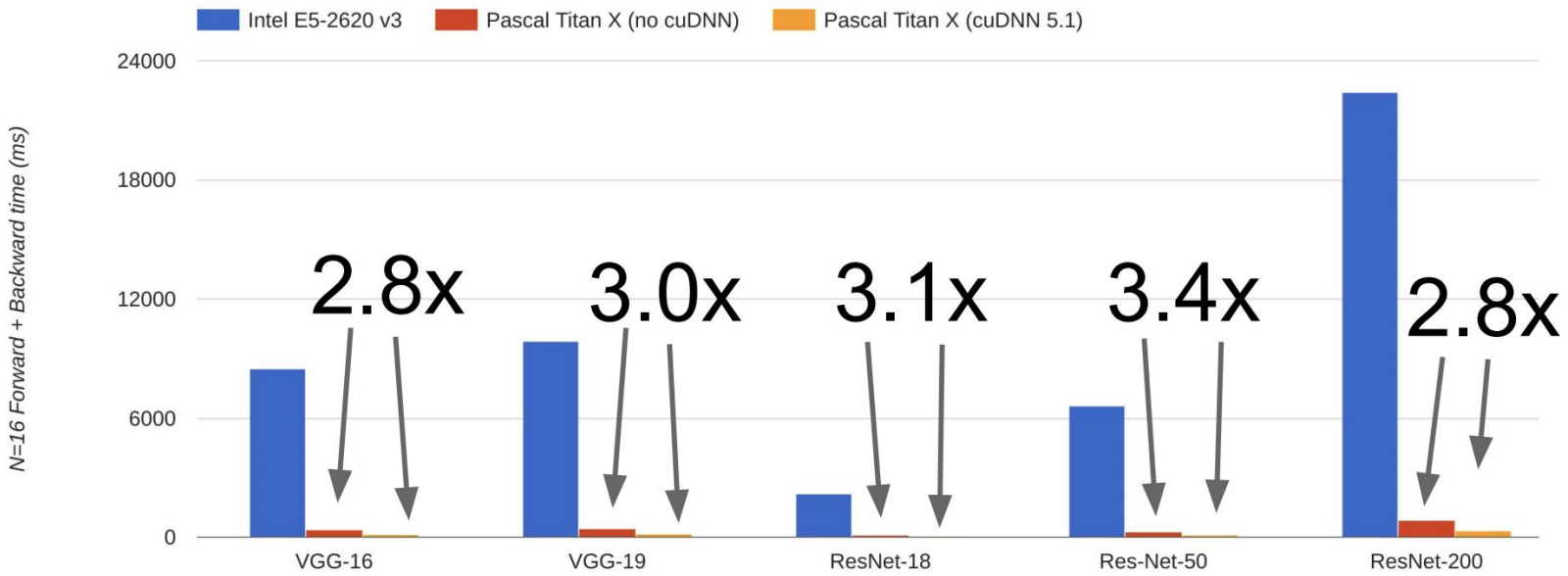
(CPU performance not well-optimized, a little unfair)



Stanford cs231n.

CPU vs GPU in practice

cuDNN much faster than
“unoptimized” CUDA



Stanford cs231n.

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

Stanford cs231n.

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
```

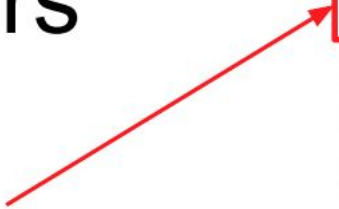
Stanford cs231n.

Pytorch Tensors

https://transfer.d2.mpi-inf.mpg.de/rshetty/hlcv/Pytorch_tutorial.pdf

PyTorch: Tensors

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



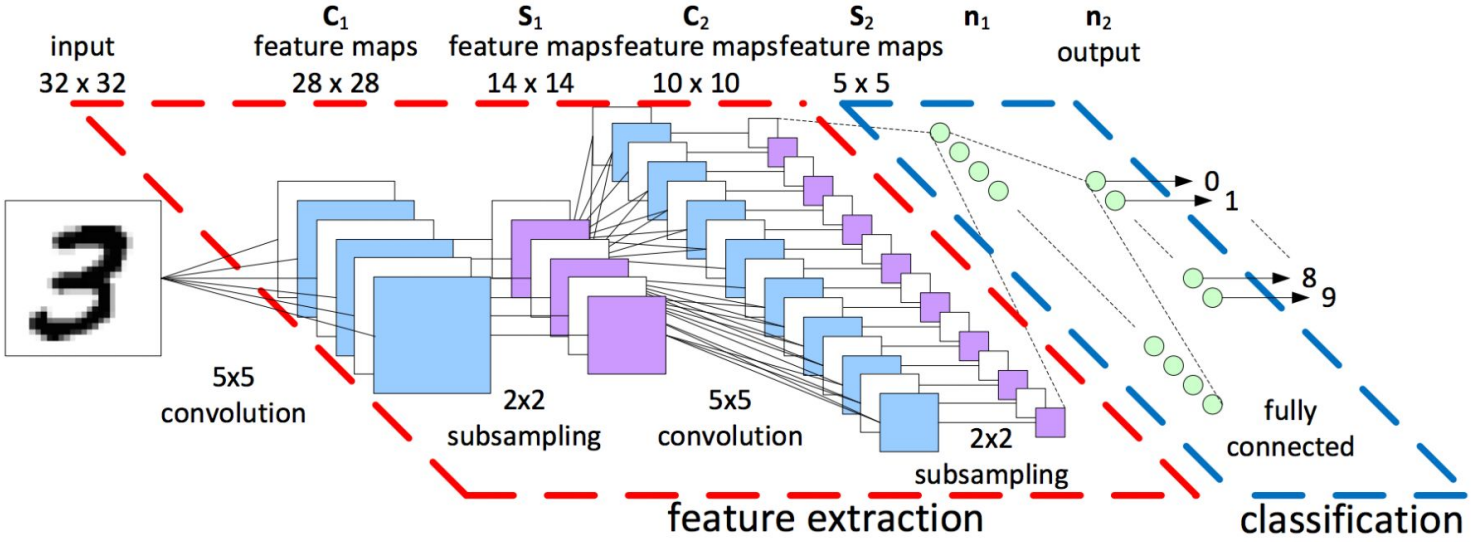
```
import torch

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
```

Stanford cs231n.

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
```

Gradient descent step on weights

Stanford cs231n.

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)

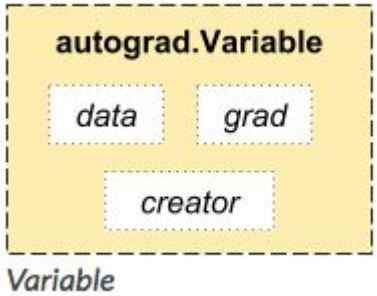
x.grad.data is a Tensor of gradients

```
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
```

Stanford cs231n.

Variable

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



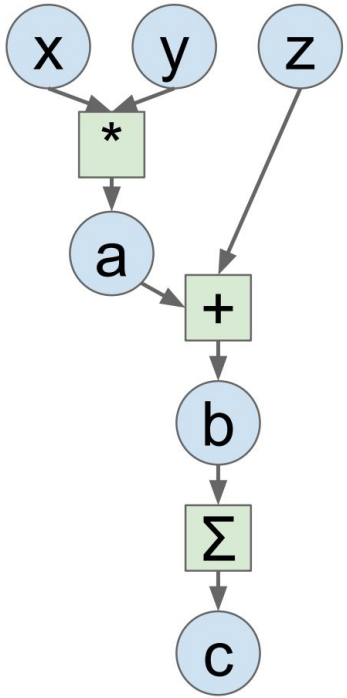
“ autograd.Variable is the central class of the package. It wraps a Tensor, and supports nearly all of operations defined on it.

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

Computational Graphs

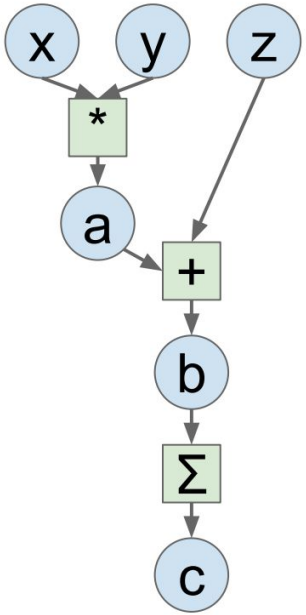
Numpy

```
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
```



Stanford cs231n.

Computational Graphs



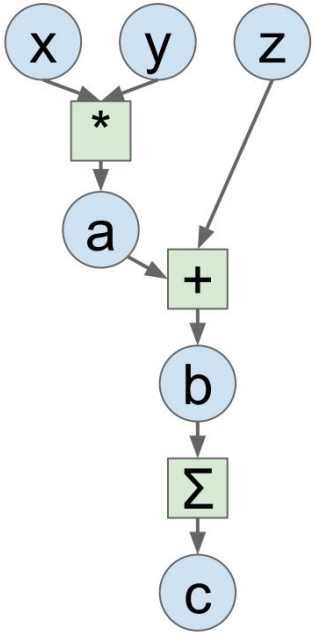
Define **Variables** to start building a computational graph

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)
```

Stanford cs231n.

Computational Graphs



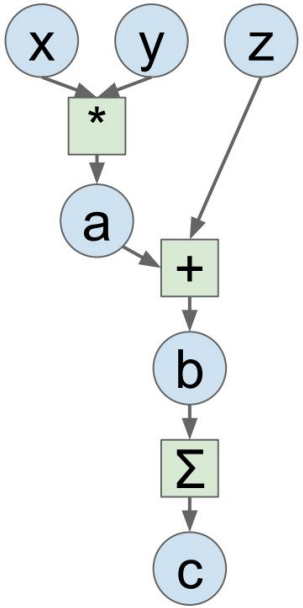
Forward pass looks just like numpy

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)
```

Stanford cs231n.

Computational Graphs



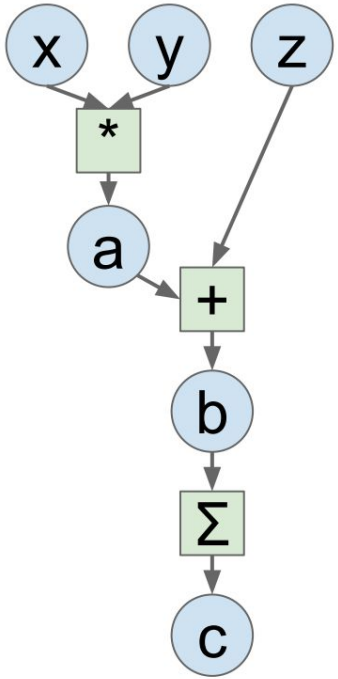
Calling c.backward() computes all gradients

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)
```

Stanford cs231n.

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)
```

Stanford cs231n.

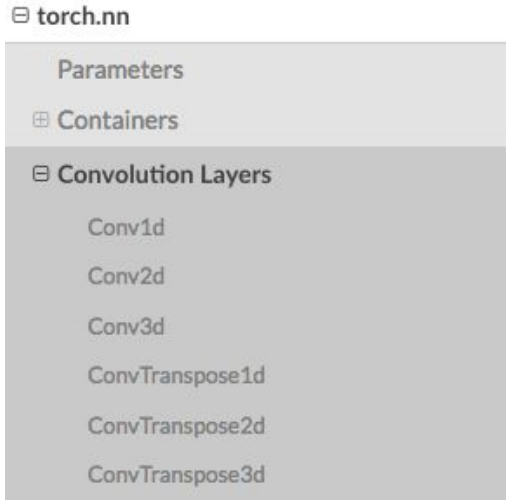
PyTorch: Three Levels of Abstraction

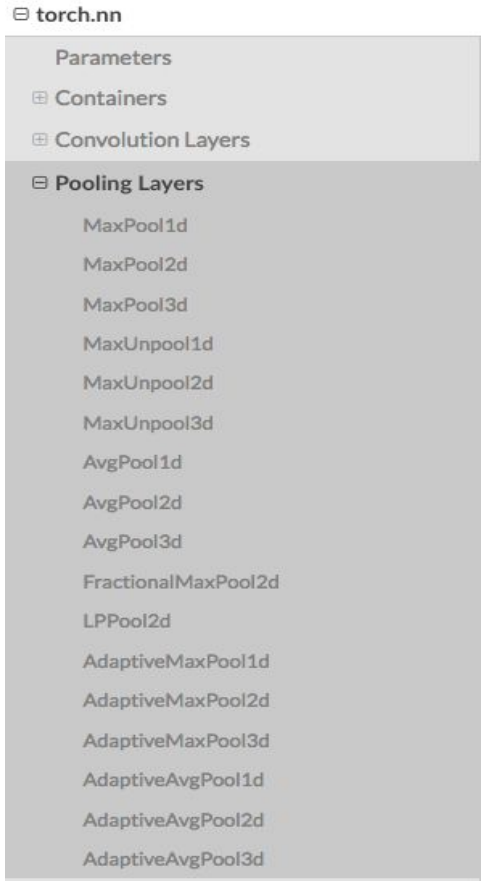
Tensor: Imperative ndarray, but runs on GPU

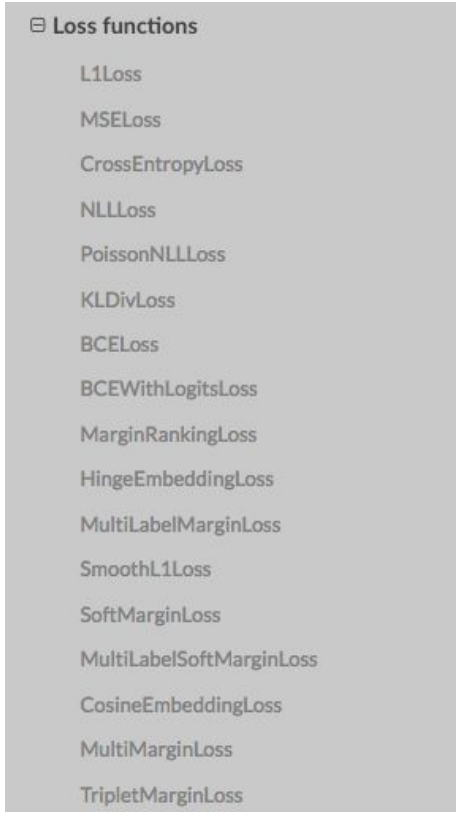
Variable: Node in a computational graph; stores data and gradient

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

Module, single layer

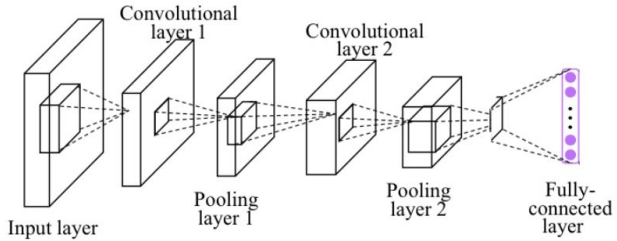






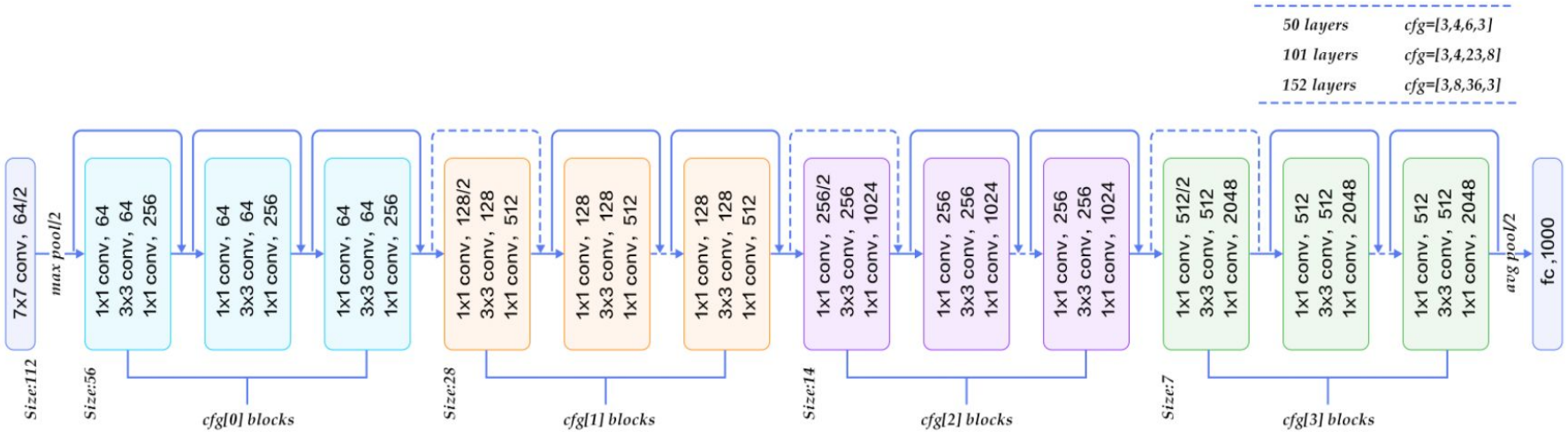
Other layers:
Dropout, Linear,
Normalization Layer

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



http://book.paddlepaddle.org/03.image_classification/

Module, sub-network

```python
# 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
```

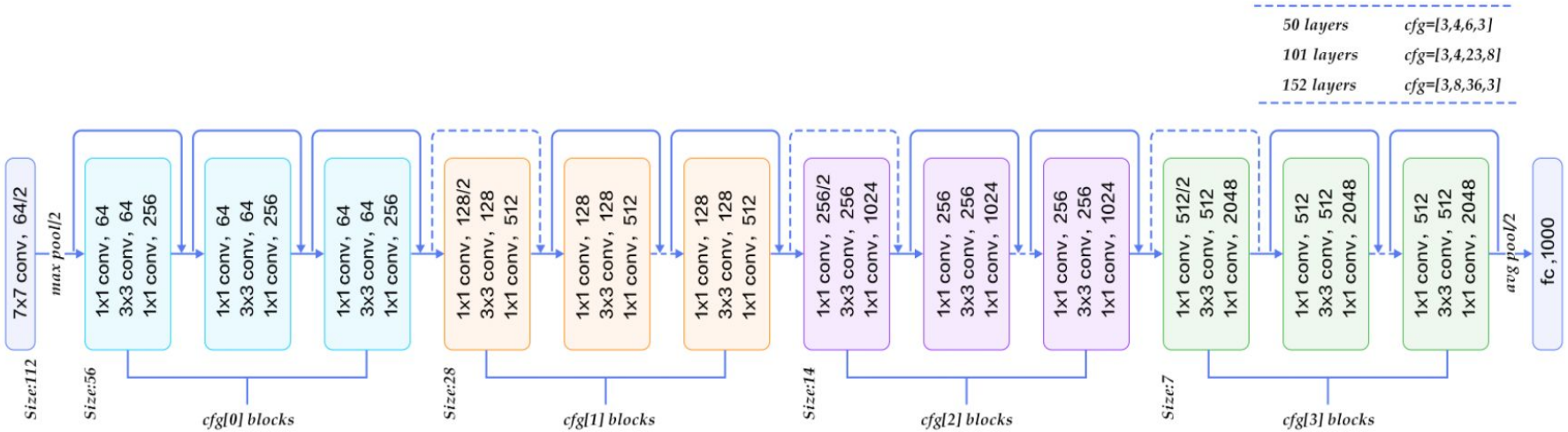
```python
class ResnetGenerator(nn.Module):
    def __init__(self, input_nc, output_nc, ngf=64, norm_layer=nn.BatchNorm2d, use_dropout=False, n_blocks=6, gpu_ids=[], padding_type='ref
        assert(n_blocks >= 0)
        super(ResnetGenerator, self).__init__()
        self.input_nc = input_nc
        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,
```

https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

Module

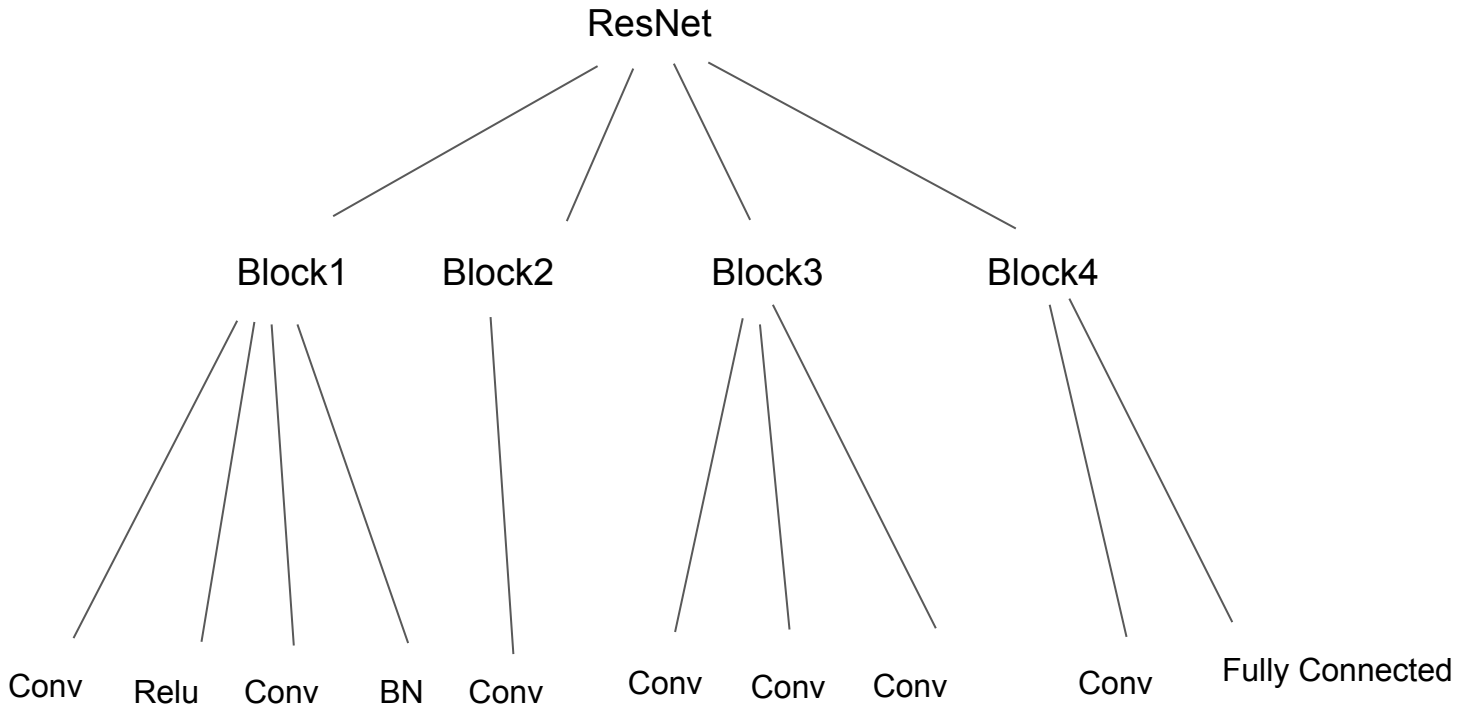
```
VGG (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU (inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU (inplace)
    (4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (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



http://book.paddlepaddle.org/03.image_classification/

Module



http://book.paddlepaddle.org/03.image_classification/

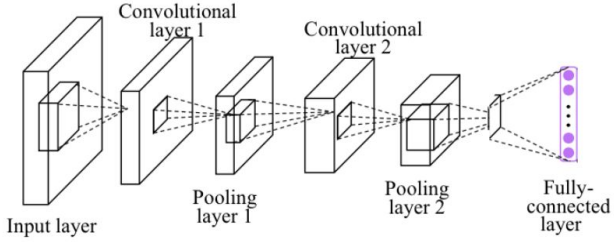
Module, sub-network

```python
# 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
```

```python
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,
```

https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

Module, network




```python
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)
```

```python
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 = size.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)

(6, 3, 32, 32)

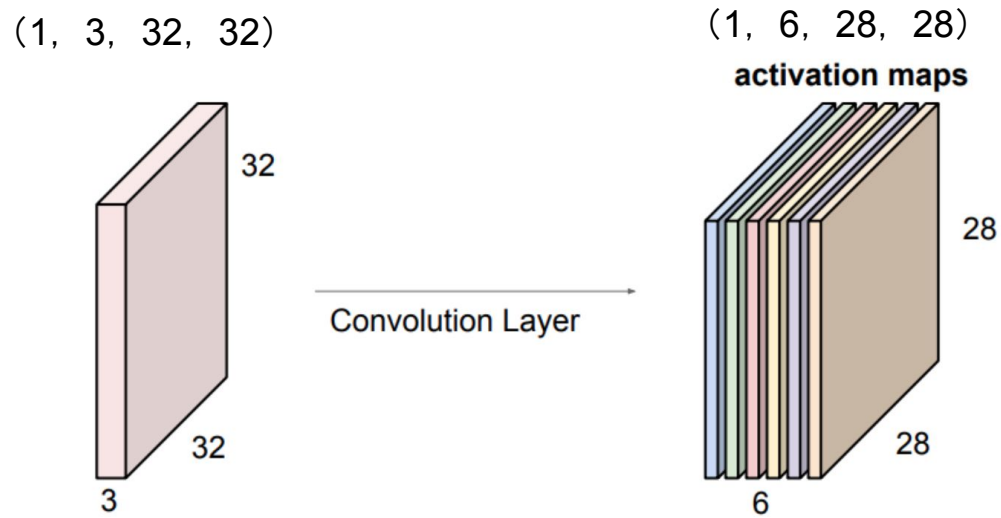
Most of the layers

(N, C)

Fully Connected Layer

Data Format (Tensor)

Conv



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

Data Format (Tensor)

Batch Channel

(N, C, H, W)

(6, 3, 24, 24)

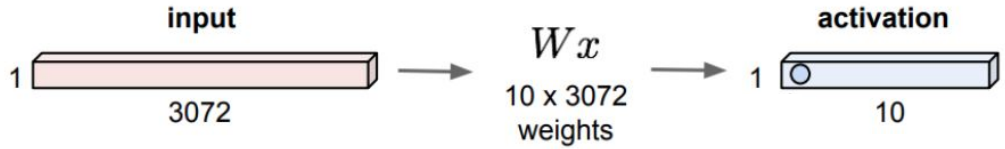
Most of the layers

(N, C)

Fully Connected Layer

Data Format (Tensor)

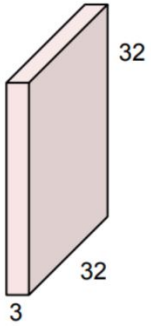
Fully Connected Layer

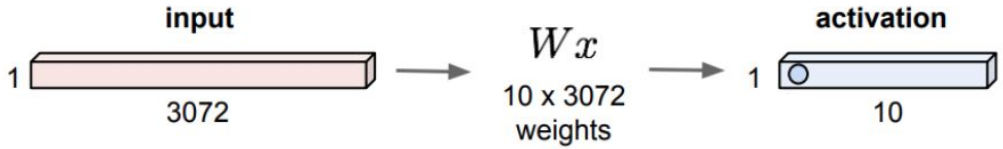



```
In pytorch,  fc = nn.Linear(3072, 10)
```

Data Format (Tensor)

Fully Conv

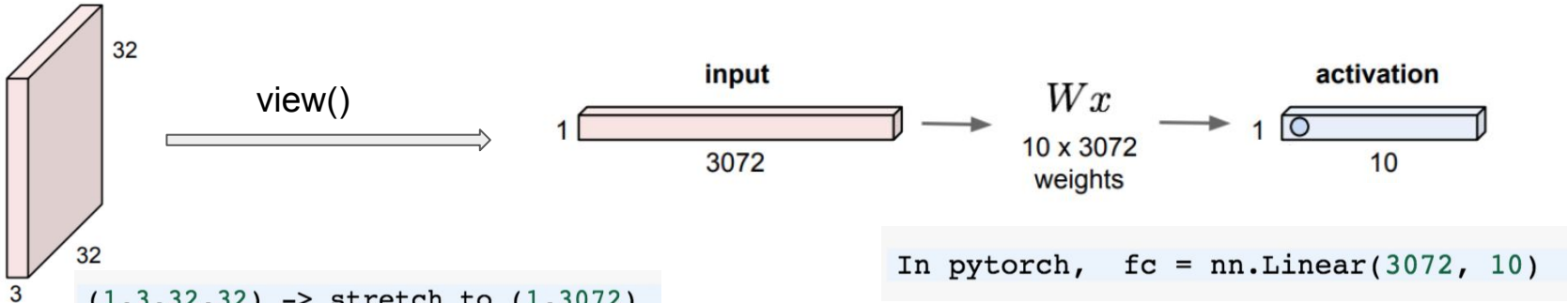





```
In pytorch,  fc = nn.Linear(3072, 10)
```

Data Format (Tensor)

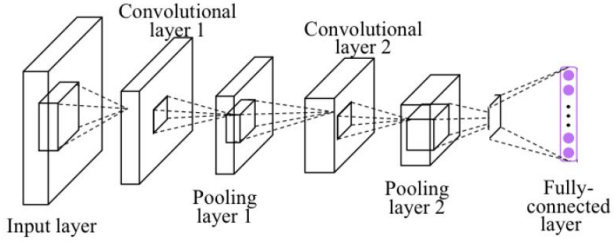
Fully Conv



`(1,3,32,32) -> stretch to (1,3072)`

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

Module, network




```python
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)
```

```python
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 = size.fc(x)

        return F.log_softmax(x)
```

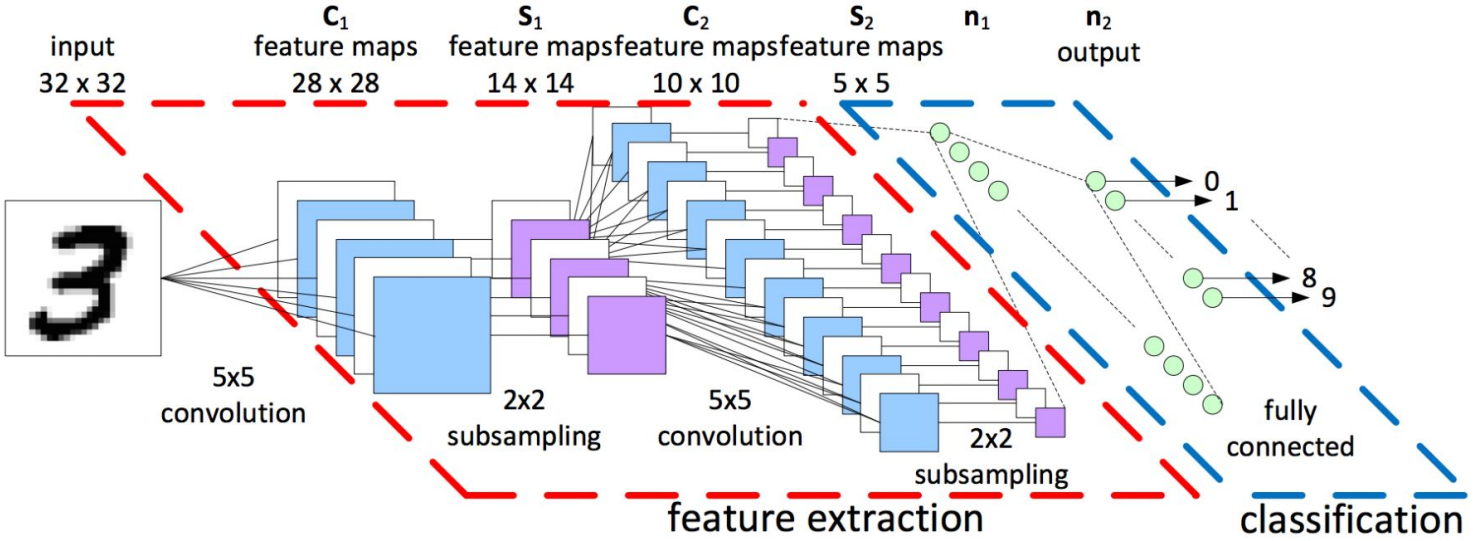
PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray, but runs on GPU

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

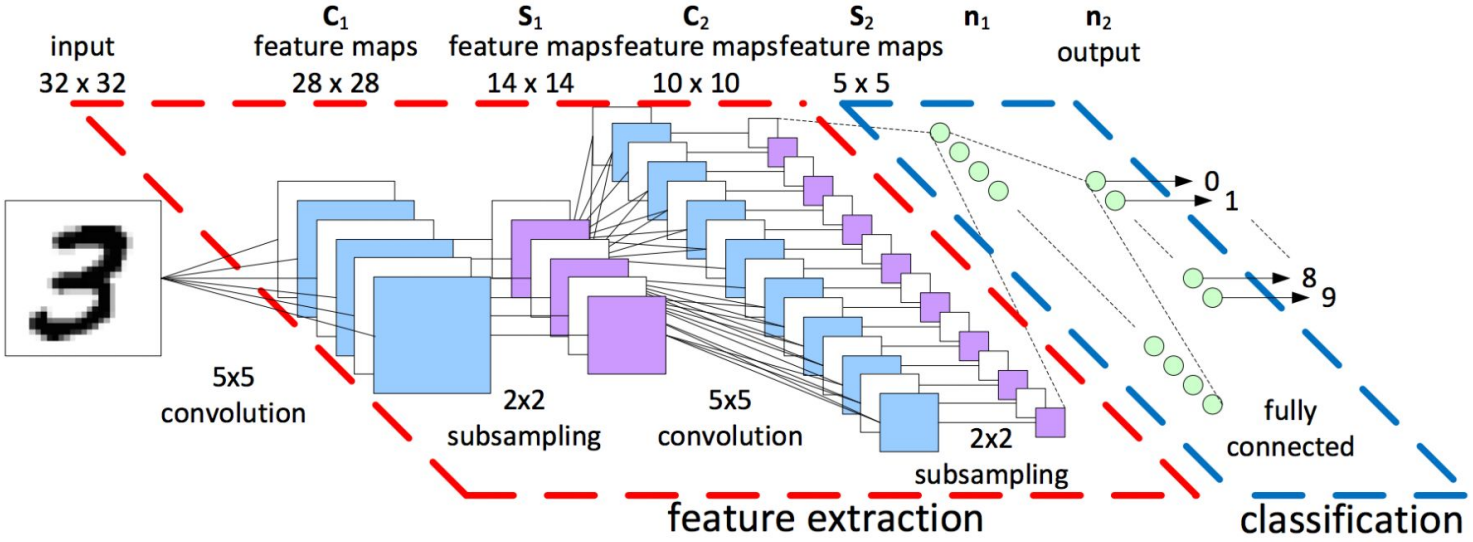
Module: A neural network layer; may store state or learnable weights

Train a simple Network



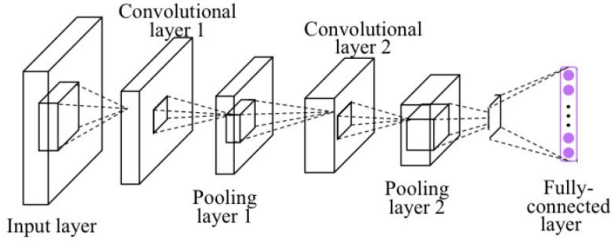
1. Forward: compute output of each layer L
2. Backward: compute gradient $\frac{\partial L}{\partial w}$
3. Update: update the parameters with computed gradient $w = w - \alpha * \frac{\partial L}{\partial w}$

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

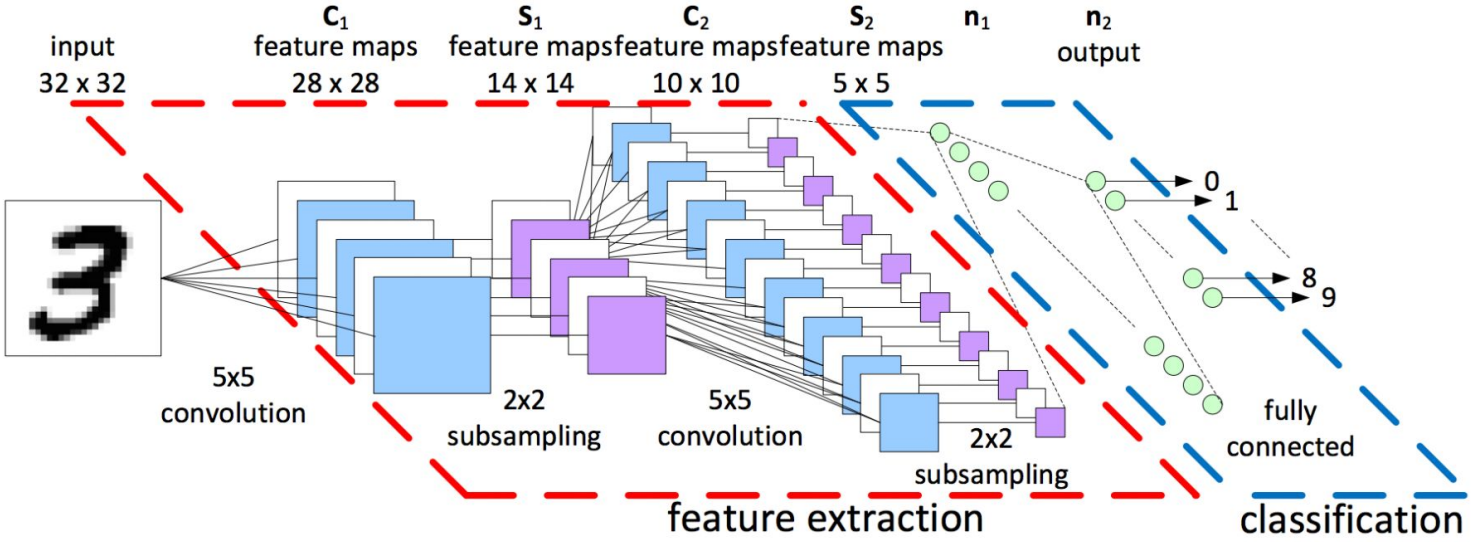
Module, network




```python
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)
```

```python
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 = size.fc(x)

        return F.log_softmax(x)
```

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

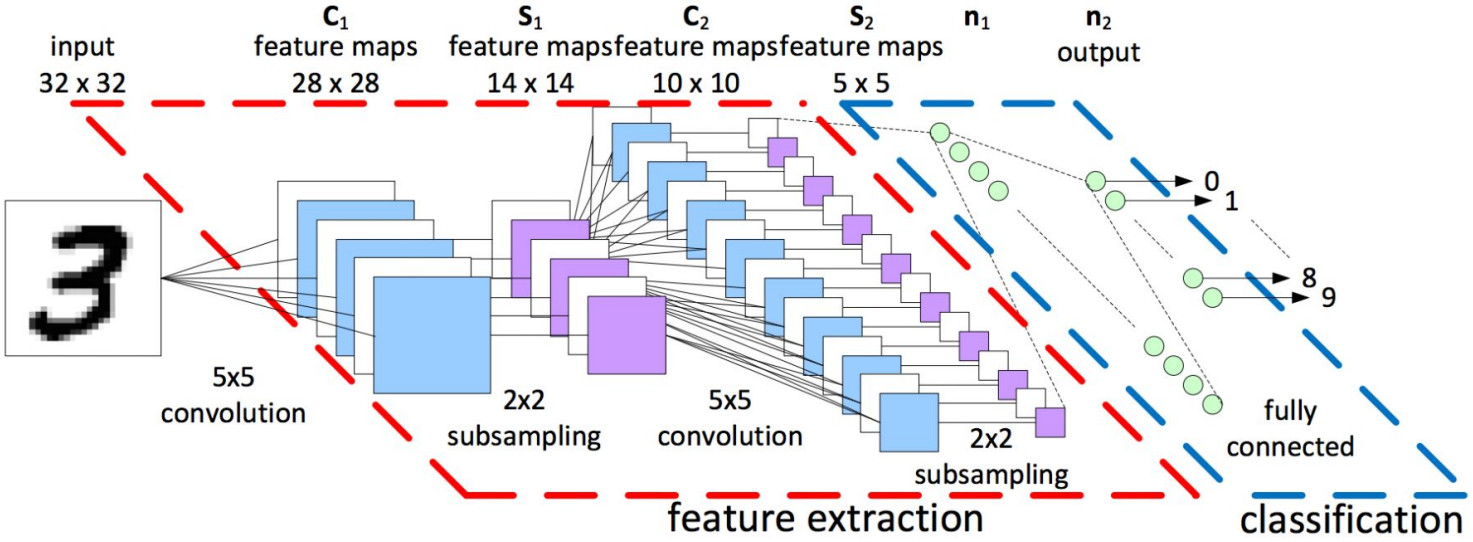
PyTorch

```
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)
```

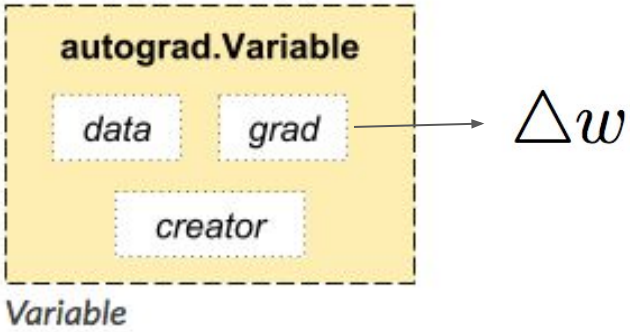
Optimizer

```
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()

### update
optimizer.step()
```

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

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

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()

### update
optimizer.step()
```

$\triangle w = 0$

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

$w = w - \alpha * \triangle w$

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
loss.backward()
optimizer.zero_grad()

### update
optimizer.step()
```

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

$$\triangle w = 0$$

$$w = w - \alpha * \triangle w$$

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()

### update
optimizer.step()
```

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

$$w = w - \alpha * \triangle w$$

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()

### update
optimizer.step()
```

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

$$w = w - \alpha * \triangle w$$

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

Optimizer

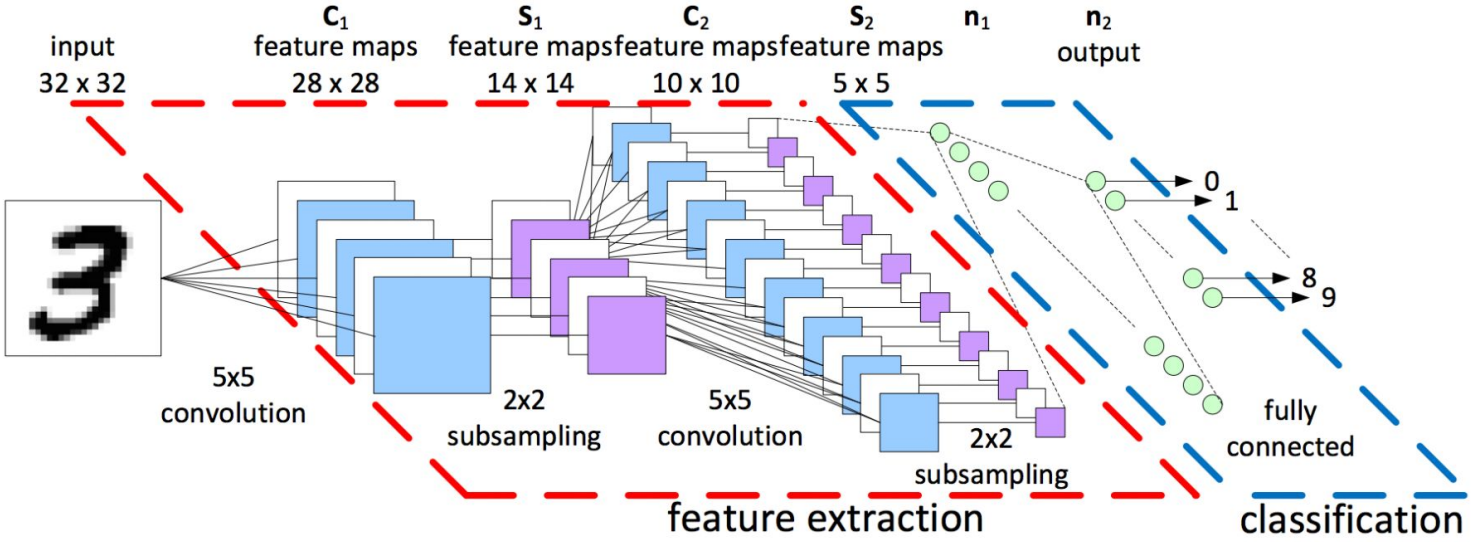
```
net = model()
optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=args.momentum)

### forward
prob = net(input_data)
loss = cross_entropy_loss(prob, groundtruth)

### backward
optimizer.zero_grad()
loss.backward()

### update
optimizer.step()
```

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

$$w = w - \alpha * \triangle w$$

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

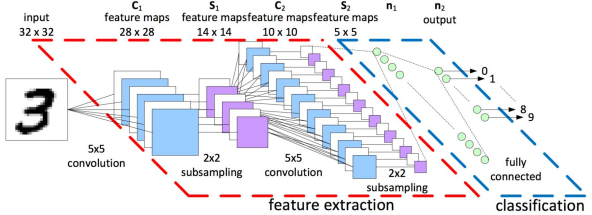
Train a simple Network



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):
```

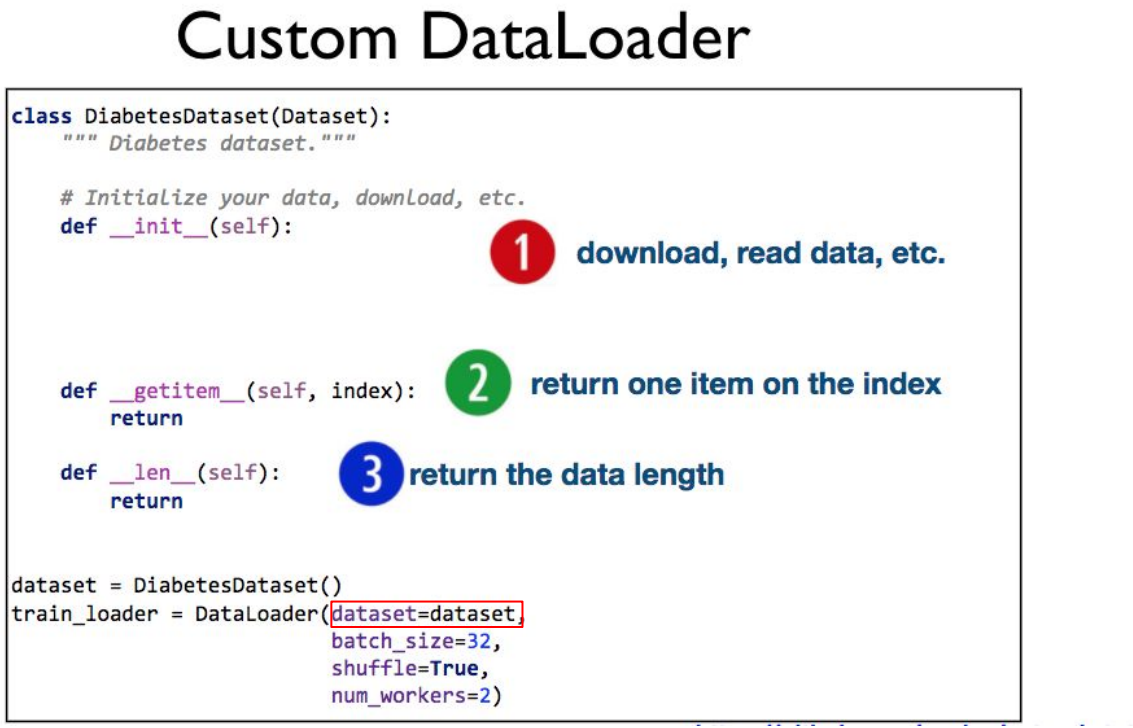
288

In the neural network terminology:

- 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

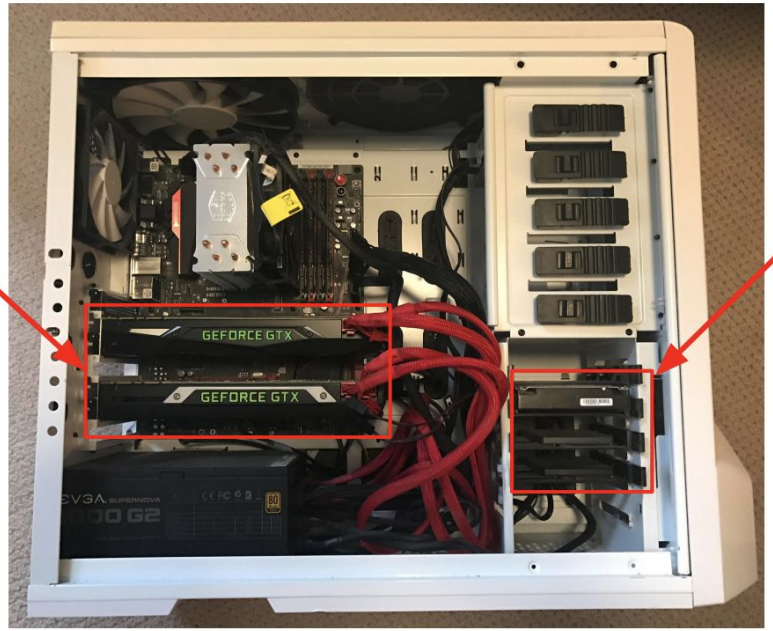


Data Loading

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)
```

CPU / GPU Communication



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

Solutions:

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

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)
```

https://goo.gl/mQEw15

Data Processing

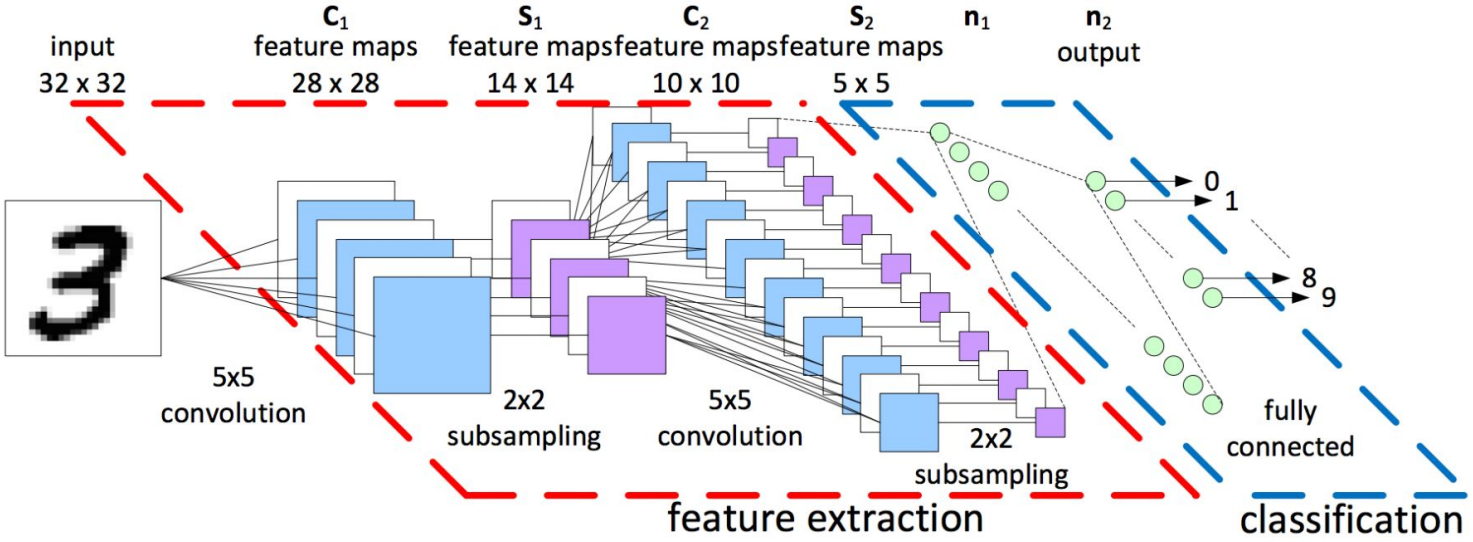
Pix2pix Code

```
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)
```

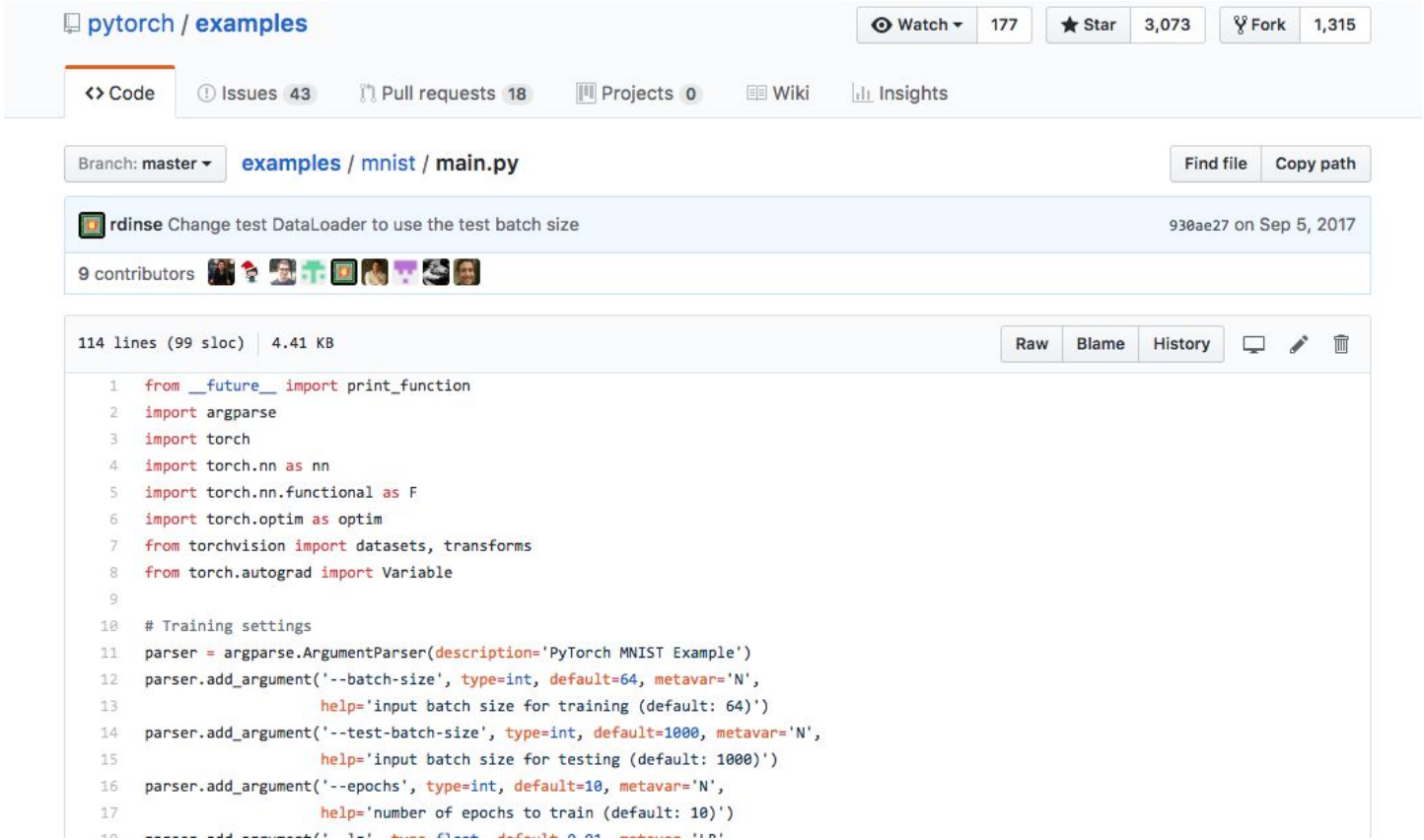
https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

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



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

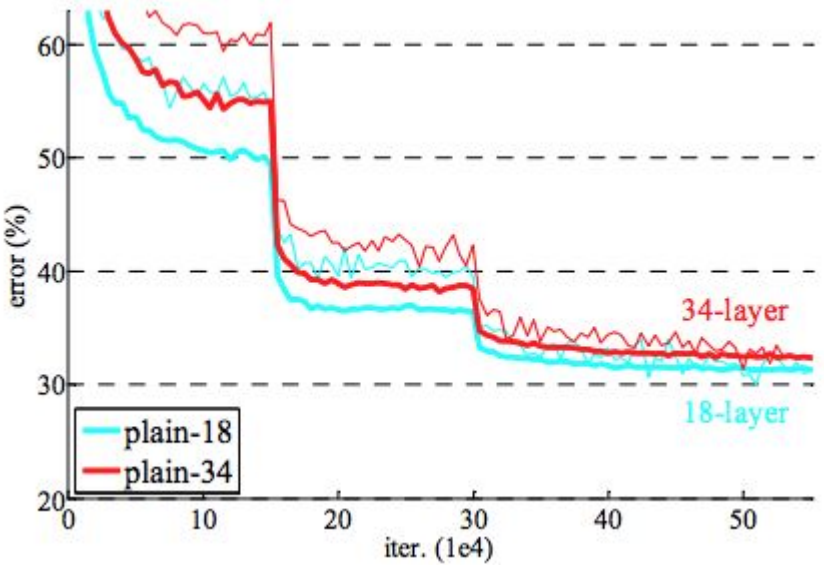
1. Pytorch-0.3.1

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



https://github.com/Jiaming-Liu/pytorch-lr-scheduler
http://pytorch.org/docs/master/optim.html#how-to-adjust-learning-rate

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_scheuler

- 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

https://github.com/Jiaming-Liu/pytorch-lr-scheduler
http://pytorch.org/docs/master/optim.html#how-to-adjust-learning-rate

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

https://gist.github.com/panovr/2977d9f26866b05583b0c40d88a315bf

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)

https://goo.gl/bqeW1K

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_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)
```

```
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)
```

https://goo.gl/bqeW1K

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
```

http://pytorch.org/tutorials/beginner/former_torchies/nn_tutorial.html

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
```

http://pytorch.org/tutorials/beginner/former_torchies/nn_tutorial.html

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

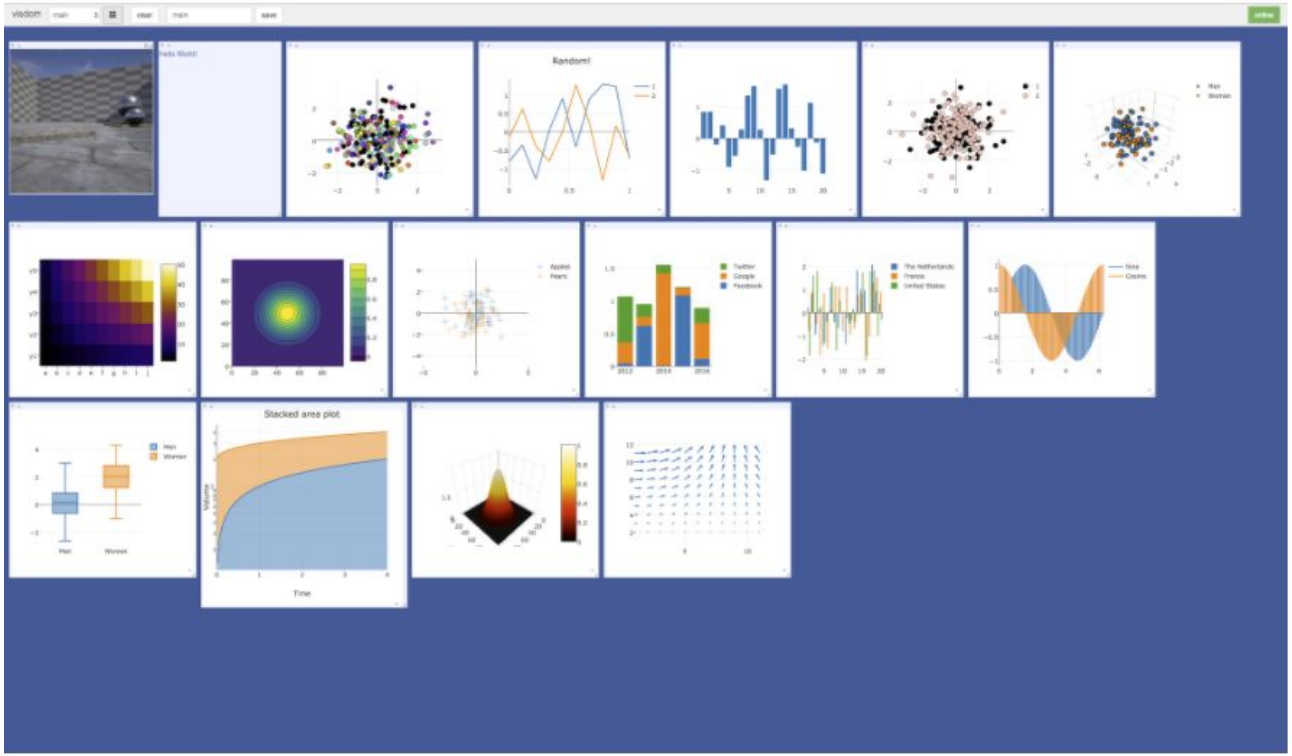
It enables benchmark mode in cudnn.
benchmark mode is good whenever your input sizes for your network do not vary. This way, cudnn will look for the optimal set of algorithms for that particular configuration (which takes some time). This usually leads to faster runtime.
But if your input sizes changes at each iteration, then cudnn will benchmark every time a new size appears, possibly leading to worse runtime performances.

11 Likes

https://goo.gl/5gzj8F

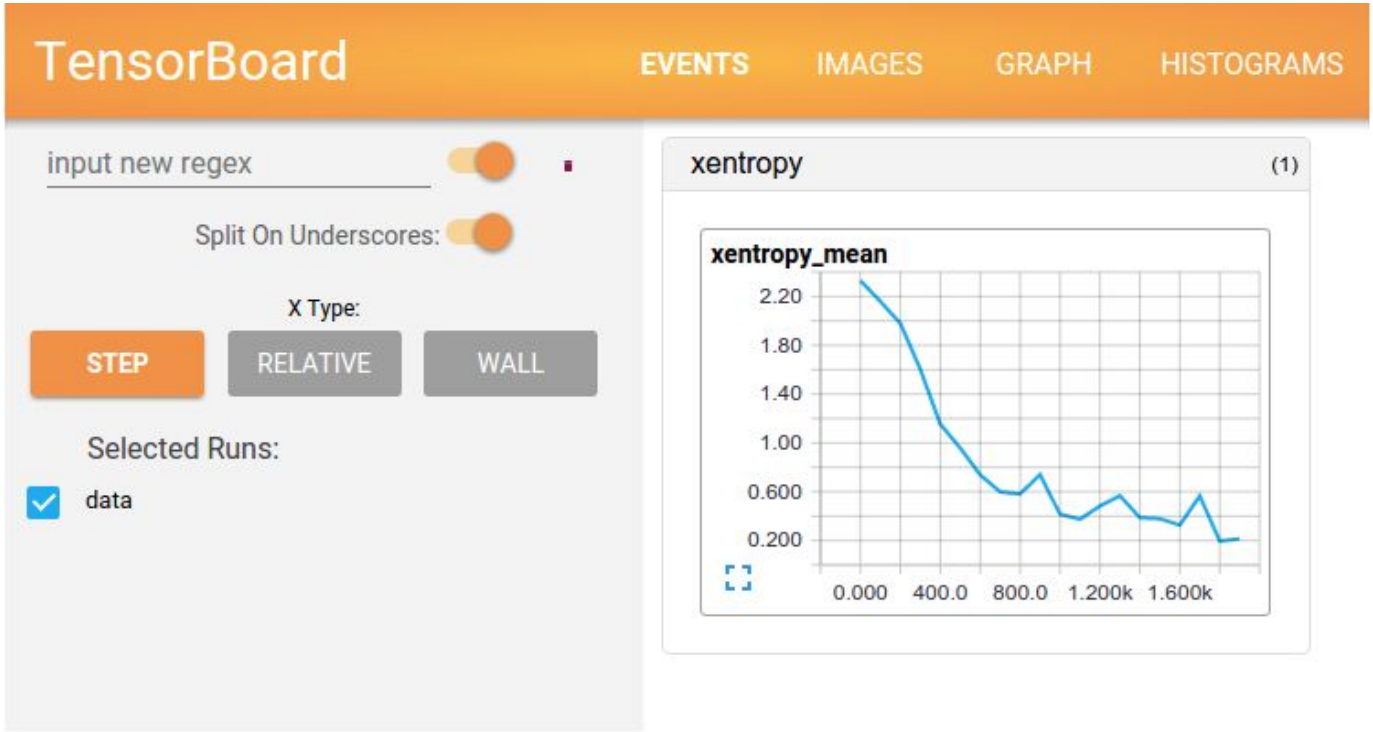
Visualization, Pytorch Visdom

https://github.com/facebookresearch/visdom



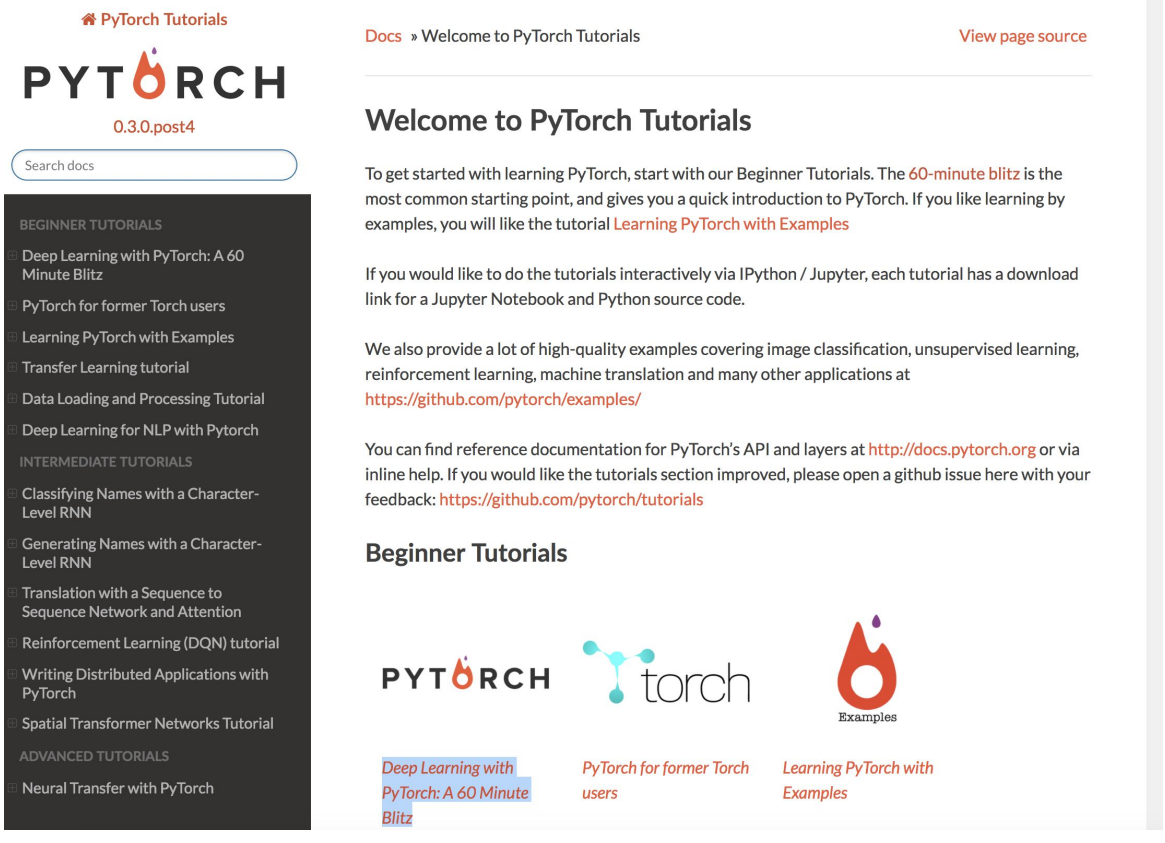
Visualization, TensorBoard

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



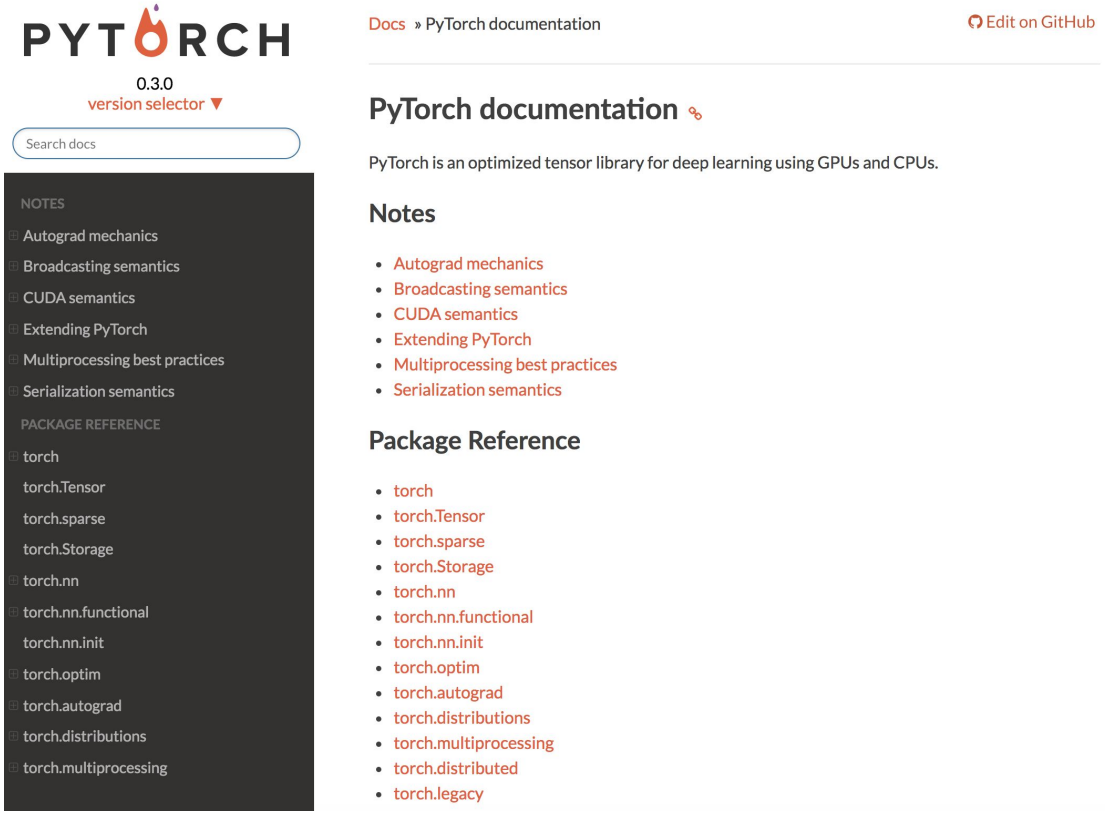
Other Resources

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



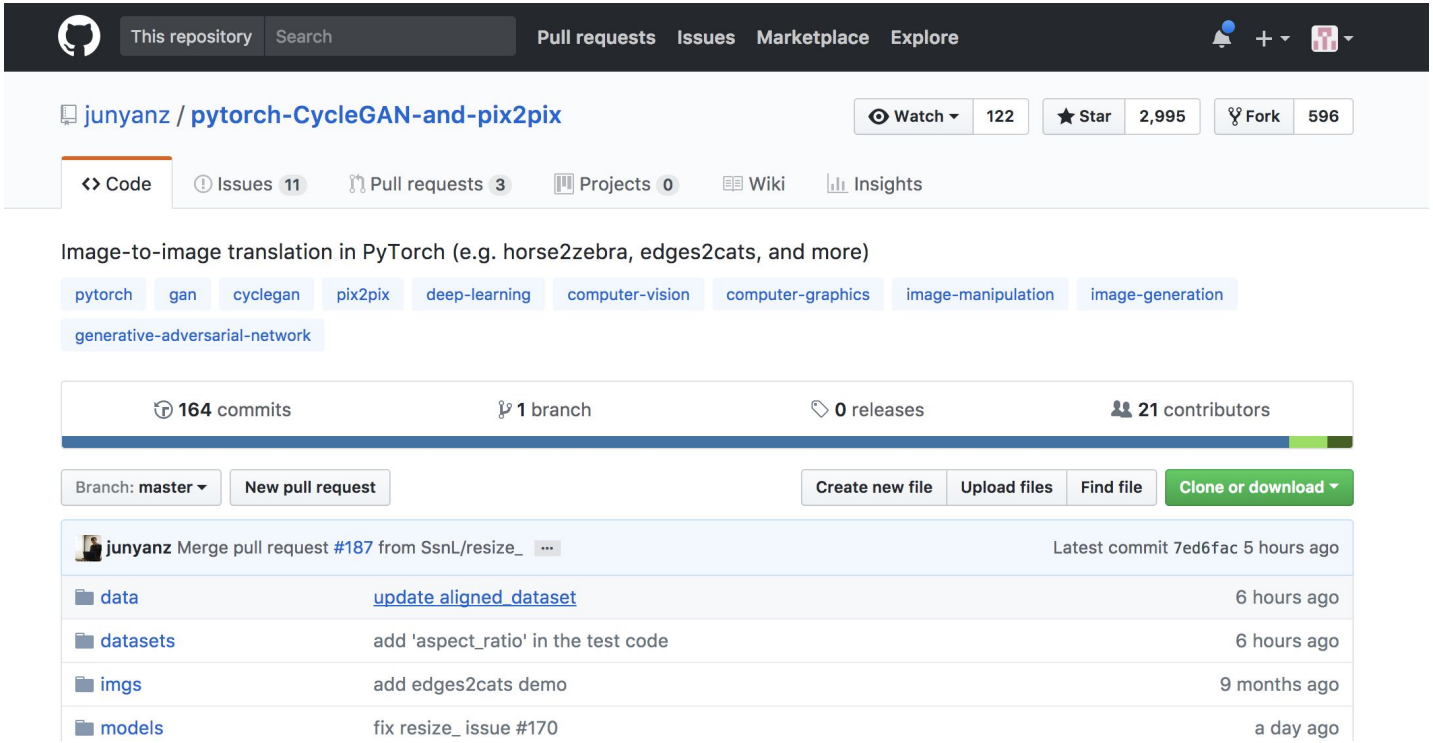
Other Resources

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



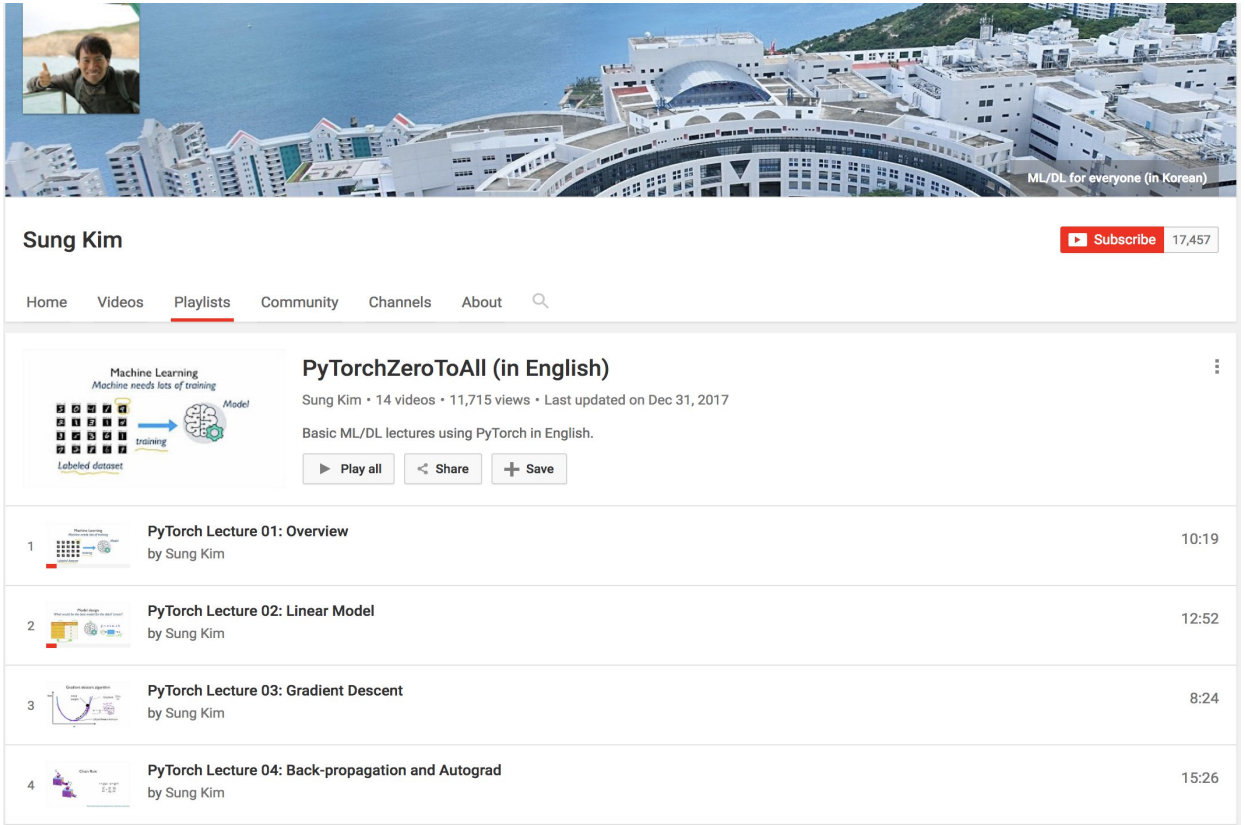
Other Resources

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



Other Resources

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



Thank You