



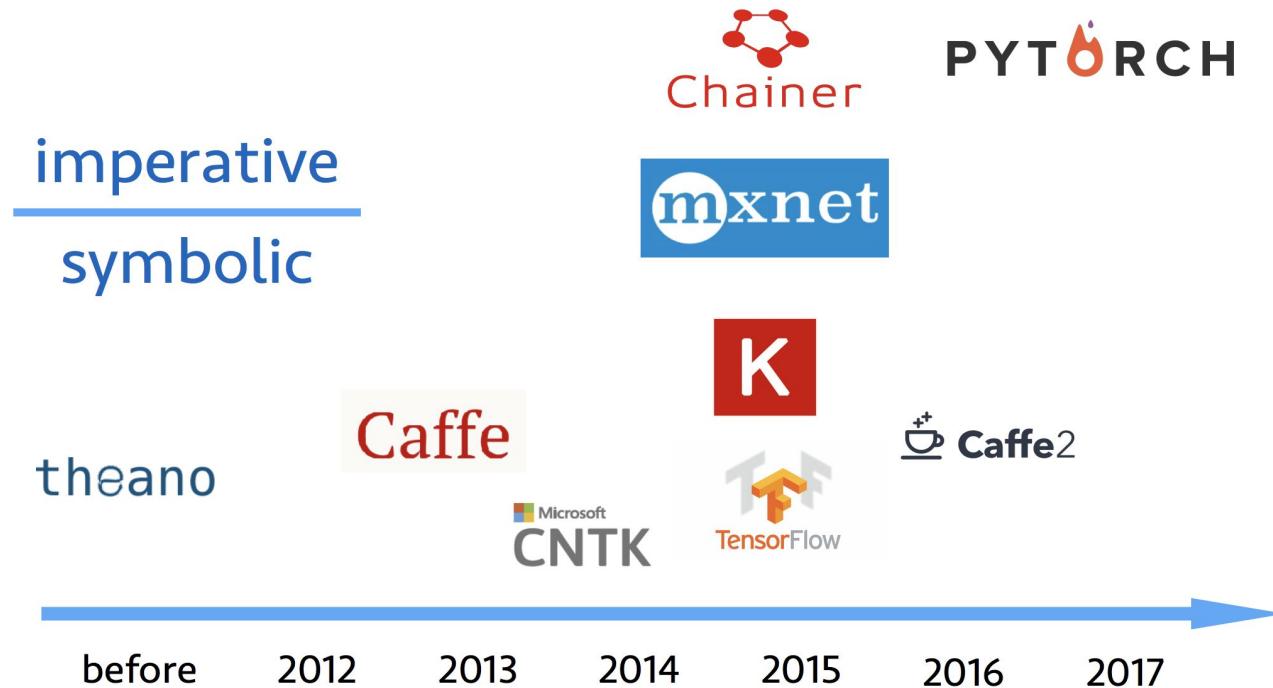
# Pytorch Tutorial

Chongruo Wu

# Agenda

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

# Popular Deep Learning Frameworks



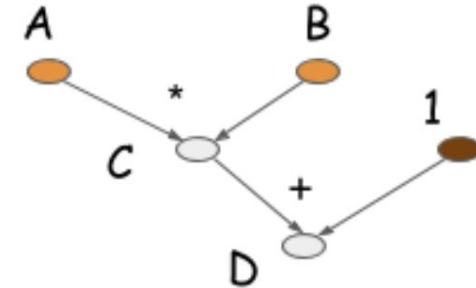
# Popular Deep Learning Frameworks

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

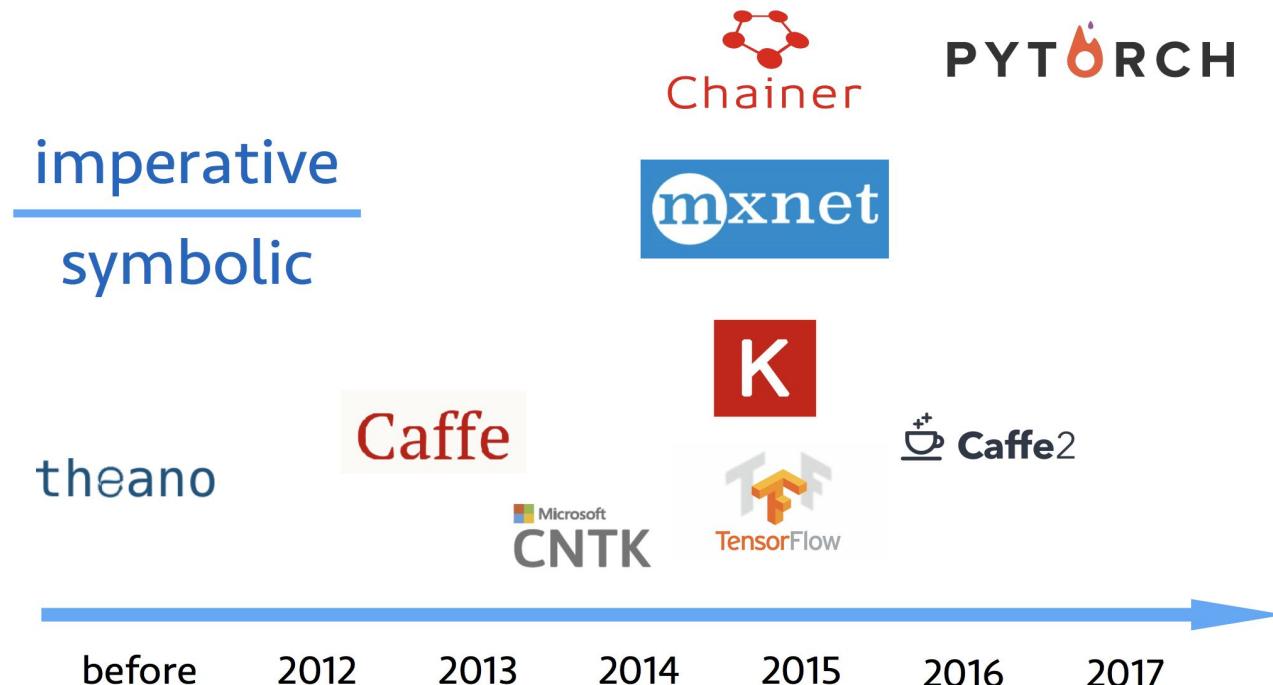
```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

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)
```

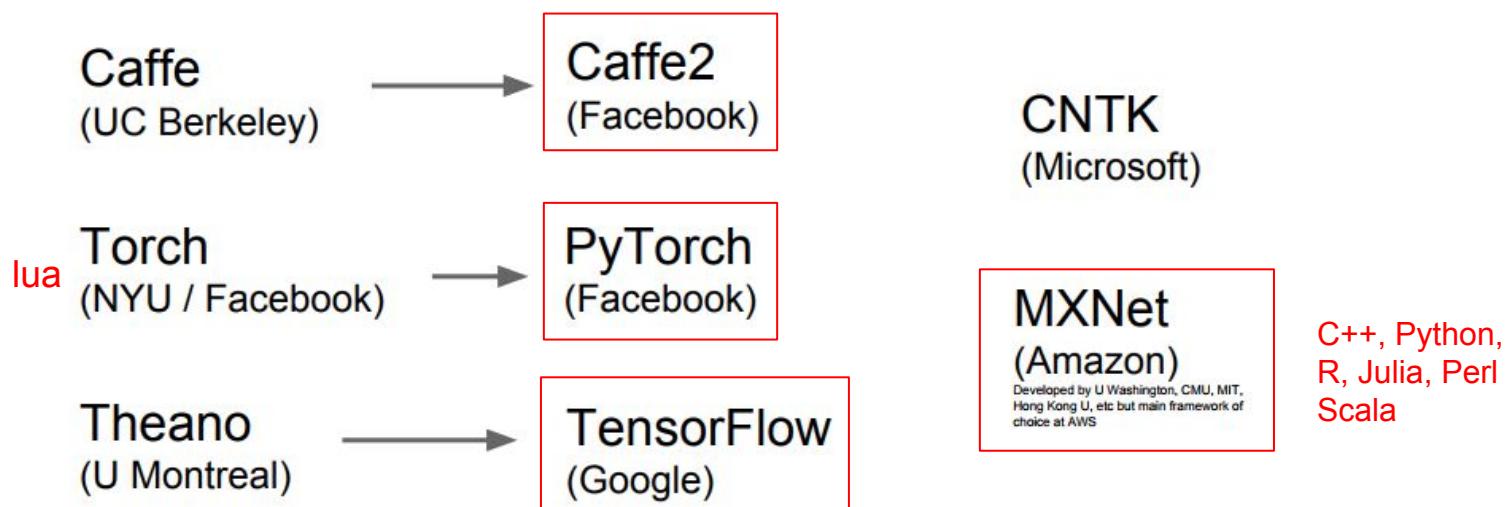


# Popular Deep Learning Frameworks



Gluon: new MXNet interface to accelerate research

# Popular Deep Learning Frameworks



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)

- ◆ Proobuf 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)
```

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

> 300 lines of codes

# 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](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](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

```
graph TD; A[tf.layers] --- B[TF-Slim]; A --- C[tf.contrib.learn]; C --- D[Pretty Tensor]; C --- E[Sonnet]; E --> C
```

# 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_filter=64, kernel_size=(3, 3), stride=(1, 1), pad=(1, 1))
```

## Implement Adam

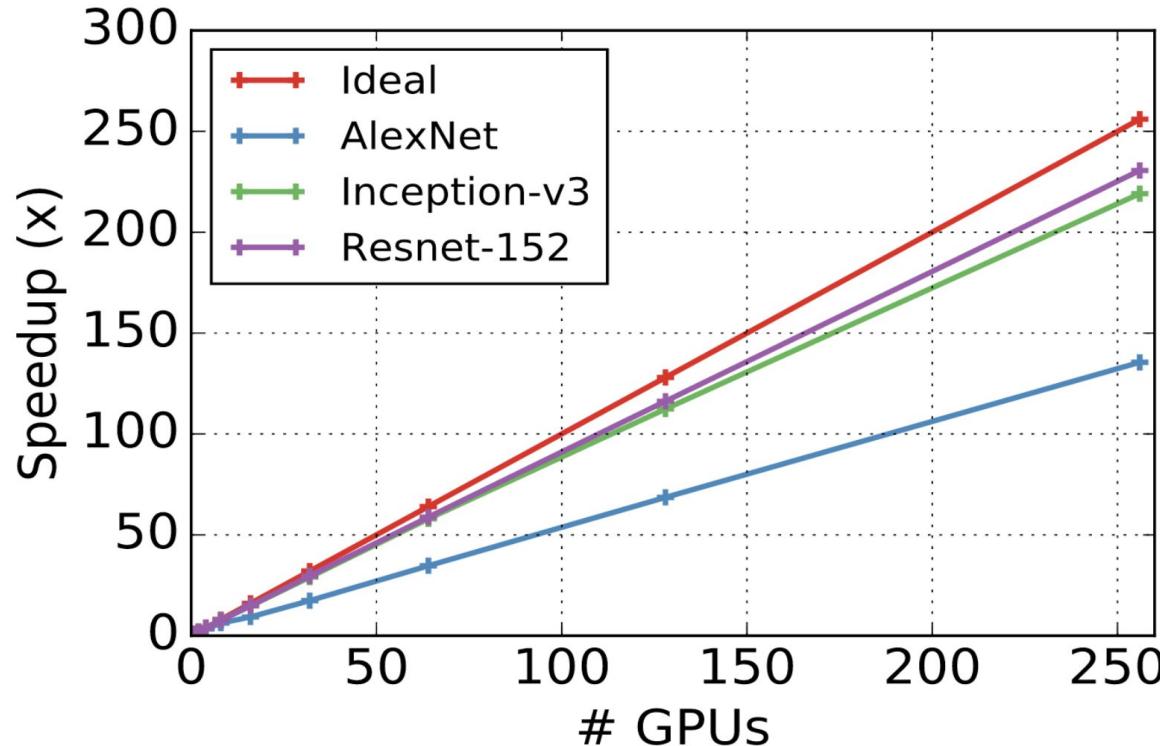
```
coef2 = 1. - self.beta2**t
lr *= math.sqrt(coef2)/coef1

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

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

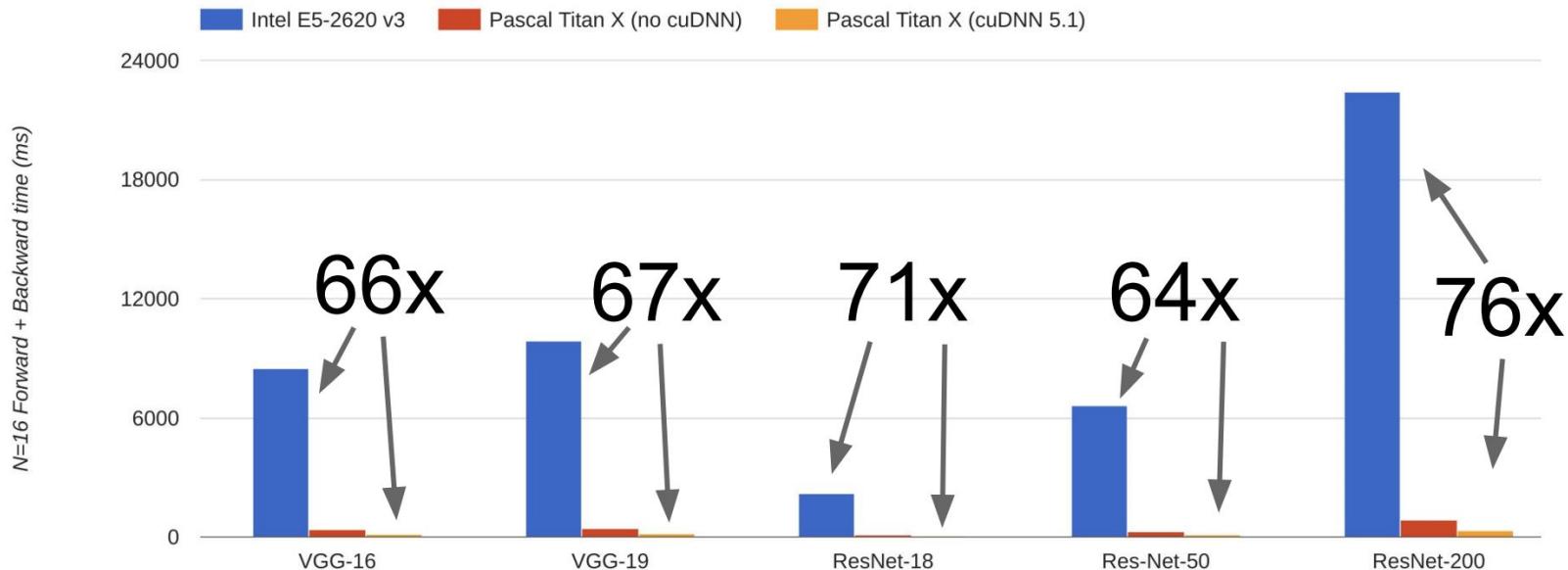
# MXNet

The following figure shows speedup against the number of GPUs used and compares it with ideal speedup.



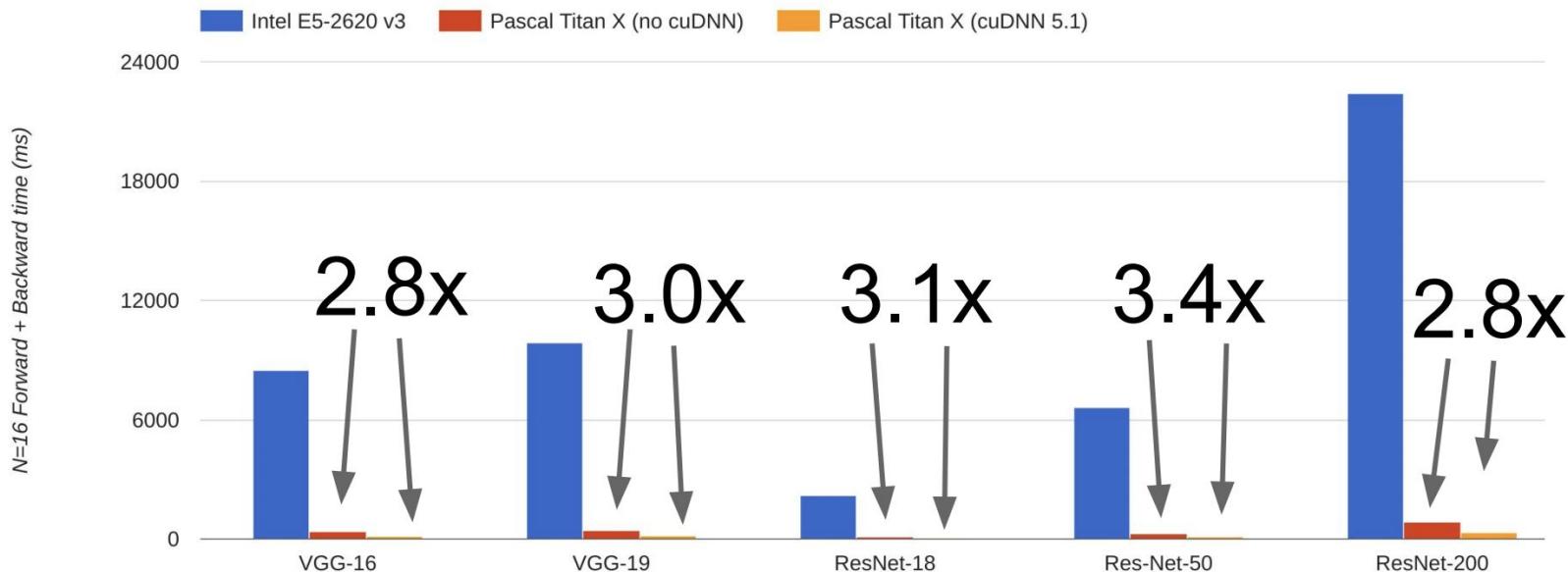
# CPU vs GPU in practice

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



# CPU vs GPU in practice

cuDNN much faster than  
“unoptimized” CUDA



# **Pytorch**

# PyTorch: 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
```

# 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)

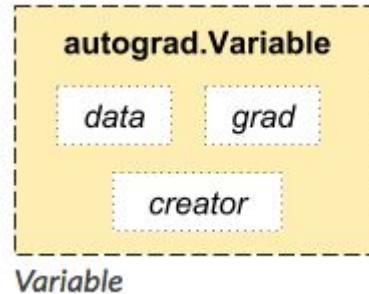
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    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

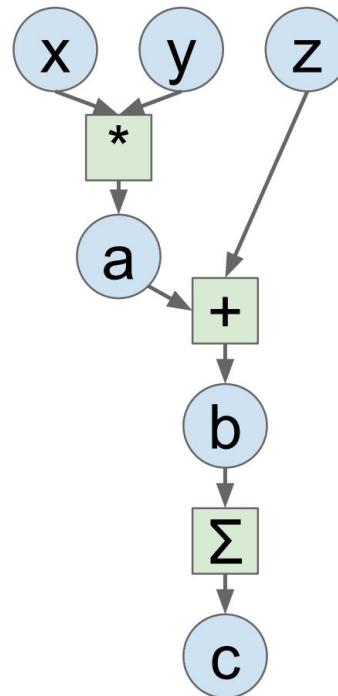
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

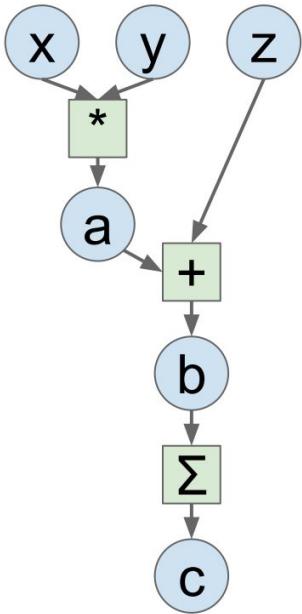
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

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

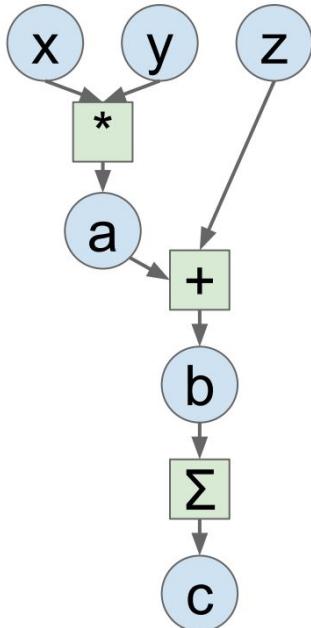
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

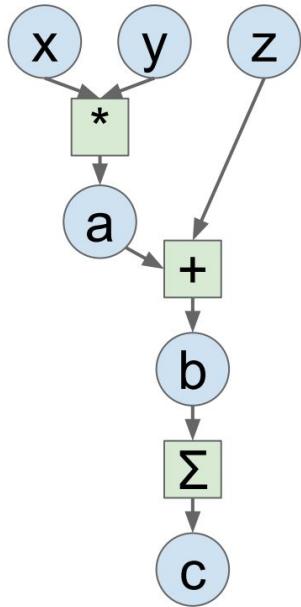
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

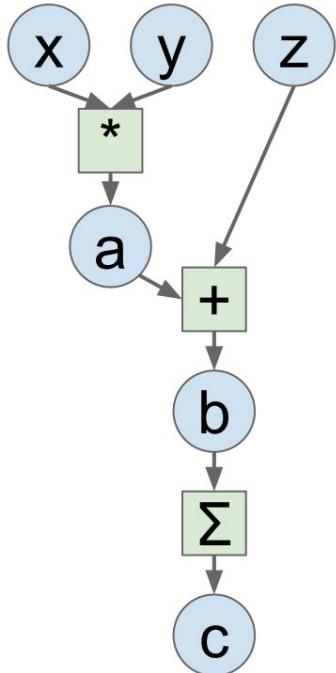
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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) # Line 10
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

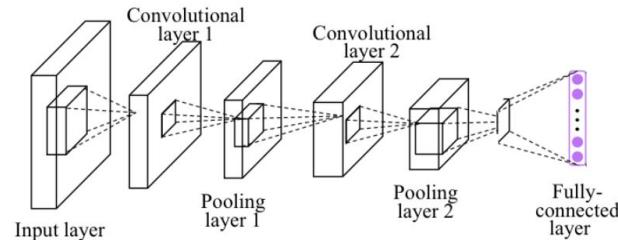
CosineEmbeddingLoss

MultiMarginLoss

TripletMarginLoss

Other layers:  
Dropout, Linear,  
Normalization Layer

## Module, network

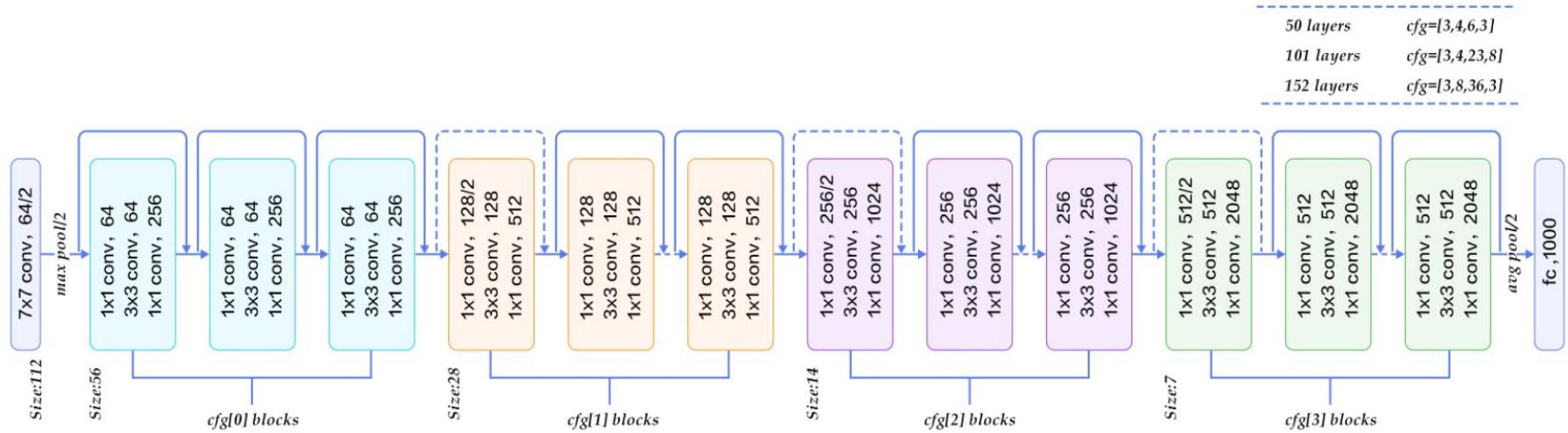


```
class Net(nn.Module):

    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)
    )
)
```

# 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

## **When starting a new project**

1. Data preparation ( processing, format )
2. Model Design ( pretrained, design your own Model)
3. Training Strategy

## Train a simple Network

# PyTorch: nn

Higher-level wrapper for working with neural nets

Similar to Keras and friends ...  
but only one, and it's good =)

```
import torch
from torch.autograd import Variable

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

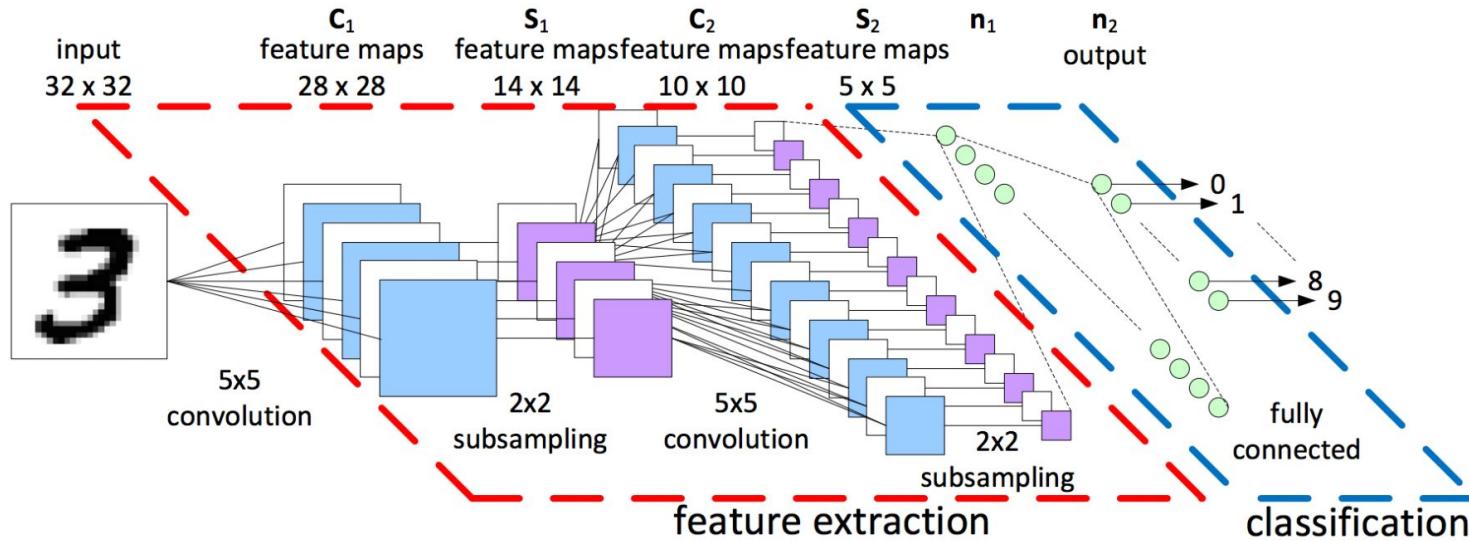
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

## Train a simple Network



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

## Train a simple Network

# PyTorch: nn

Define our model as a sequence of layers

nn also defines common loss functions

```
import torch
from torch.autograd import Variable

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

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

loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

## Train a simple Network

# PyTorch: nn

Forward pass: feed data to model, and prediction to loss function

```
import torch
from torch.autograd import Variable

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

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

## Train a simple Network

# PyTorch: nn

Backward pass:  
compute all 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))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

## Train a simple Network

# PyTorch: nn

```
import torch
from torch.autograd import Variable

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

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Make gradient step on  
each model parameter



## Train a simple Network

# PyTorch: optim

Use an **optimizer** for different update rules

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

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

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

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```



## Train a simple Network

# PyTorch: optim

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

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

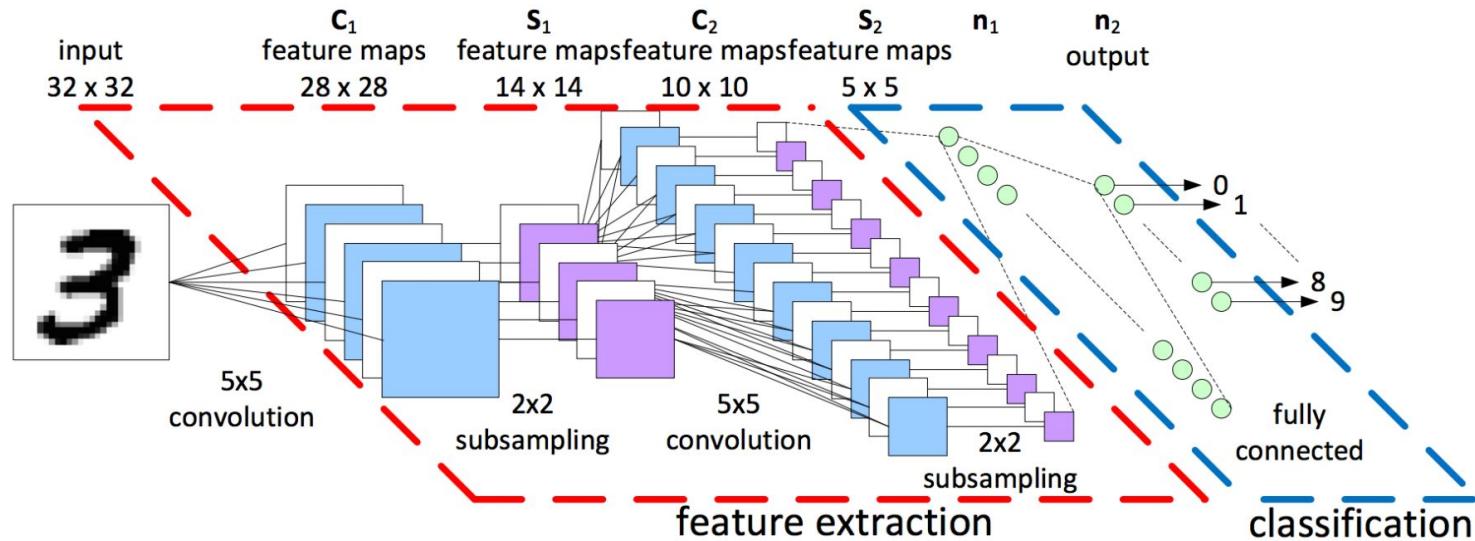
    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

Update all parameters  
after computing gradients



## MNIST example



# 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

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable

# Training settings
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                    help='input batch size for training (default: 64)')
parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                    help='input batch size for testing (default: 1000)')
parser.add_argument('--epochs', type=int, default=10, metavar='N',
                    help='number of epochs to train (default: 10)')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate (default: 0.01)')
parser.add_argument('--momentum', type=float, default=0.5, metavar='M',
                    help='SGD momentum (default: 0.5)')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='how many batches to wait before logging training status')  
...  
def weights_init(m):  
    if isinstance(m, nn.Conv2d):  
        m.weight.data.normal_(0.0, 0.02)  
        if m.bias is not None:  
            m.bias.data.fill_(0)  
    elif isinstance(m, nn.Linear):  
        m.weight.data.normal_(0.0, 0.01)  
        if m.bias is not None:  
            m.bias.data.fill_(0)
```

## MNIST example

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable

# Training settings
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                    help='input batch size for training (default: 64)')
parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                    help='input batch size for testing (default: 1000)')
parser.add_argument('--epochs', type=int, default=10, metavar='N',
                    help='number of epochs to train (default: 10)')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate (default: 0.01)')
parser.add_argument('--momentum', type=float, default=0.5, metavar='M',
                    help='SGD momentum (default: 0.5)')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='how many batches to wait before logging training status')
args = parser.parse_args()
args.cuda = not args.no_cuda and torch.cuda.is_available()

torch.manual_seed(args.seed)
if args.cuda:
    torch.cuda.manual_seed(args.seed)
```

## MNIST example

### Data Loading

```
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)
```

## MNIST example

### Define Network

```
52  class Net(nn.Module):
53      def __init__(self):
54          super(Net, self).__init__()
55          self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
56          self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
57          self.conv2_drop = nn.Dropout2d()
58          self.fc1 = nn.Linear(320, 50)
59          self.fc2 = nn.Linear(50, 10)
60
61      def forward(self, x):
62          x = F.relu(F.max_pool2d(self.conv1(x), 2))
63          x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
64          x = x.view(-1, 320)
65          x = F.relu(self.fc1(x))
66          x = F.dropout(x, training=self.training)
67          x = self.fc2(x)
68          return F.log_softmax(x)
69
70  model = Net()
71  if args.cuda:
72      model.cuda()
```

## MNIST example

Training

```
74     optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum)
75
76     def train(epoch):
77         model.train()
78         for batch_idx, (data, target) in enumerate(train_loader):
79             if args.cuda:
80                 data, target = data.cuda(), target.cuda()
81                 data, target = Variable(data), Variable(target)
82                 optimizer.zero_grad()
83                 output = model(data)
84                 loss = F.nll_loss(output, target)
85                 loss.backward()
86                 optimizer.step()
87             if batch_idx % args.log_interval == 0:
88                 print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
89                     epoch, batch_idx * len(data), len(train_loader.dataset),
90                     100. * batch_idx / len(train_loader), loss.data[0]))
```

## MNIST example

Inference

eval() mode:

\*Dropout Layer  
\*Batchnorm Layer

```
92  def test():
93      model.eval()
94      test_loss = 0
95      correct = 0
96      for data, target in test_loader:
97          if args.cuda:
98              data, target = data.cuda(), target.cuda()
99              data, target = Variable(data, volatile=True), Variable(target)
100             output = model(data)
101             test_loss += F.nll_loss(output, target, size_average=False).data[0] # sum up batch loss
102             pred = output.data.max(1, keepdim=True)[1] # get the index of the max log-probability
103             correct += pred.eq(target.data.view_as(pred)).cpu().sum()
104
105             test_loss /= len(test_loader.dataset)
106             print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
107                 test_loss, correct, len(test_loader.dataset),
108                 100. * correct / len(test_loader.dataset)))
```

## **When starting a new project**

1. Data preparation ( processing, format )
2. Model Design ( pretrained, design your own model)
3. Training Strategy

# **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

# Manual data feed

```
xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
x_data = Variable(torch.from_numpy(xy[:, 0:-1]))
y_data = Variable(torch.from_numpy(xy[:, [-1]]))

...
# Training Loop
for epoch in range(100):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

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

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

## Data Loading

```
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 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):
        return

    def __len__(self):
        return

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

1

download, read data, etc.

2

return one item on the index

3

return the data length

## 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)
```

## **When starting a new project**

1. Data preparation ( processing, format )
2. Model Design ( pretrained, design your own model)
3. Training Strategy ( learning rate )

## Learning Rate Scheduler

# PyTorch: optim

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

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

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

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

Update all parameters  
after computing gradients



## 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.squeezezenet1_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: 0.02763858002902972
```

## 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.

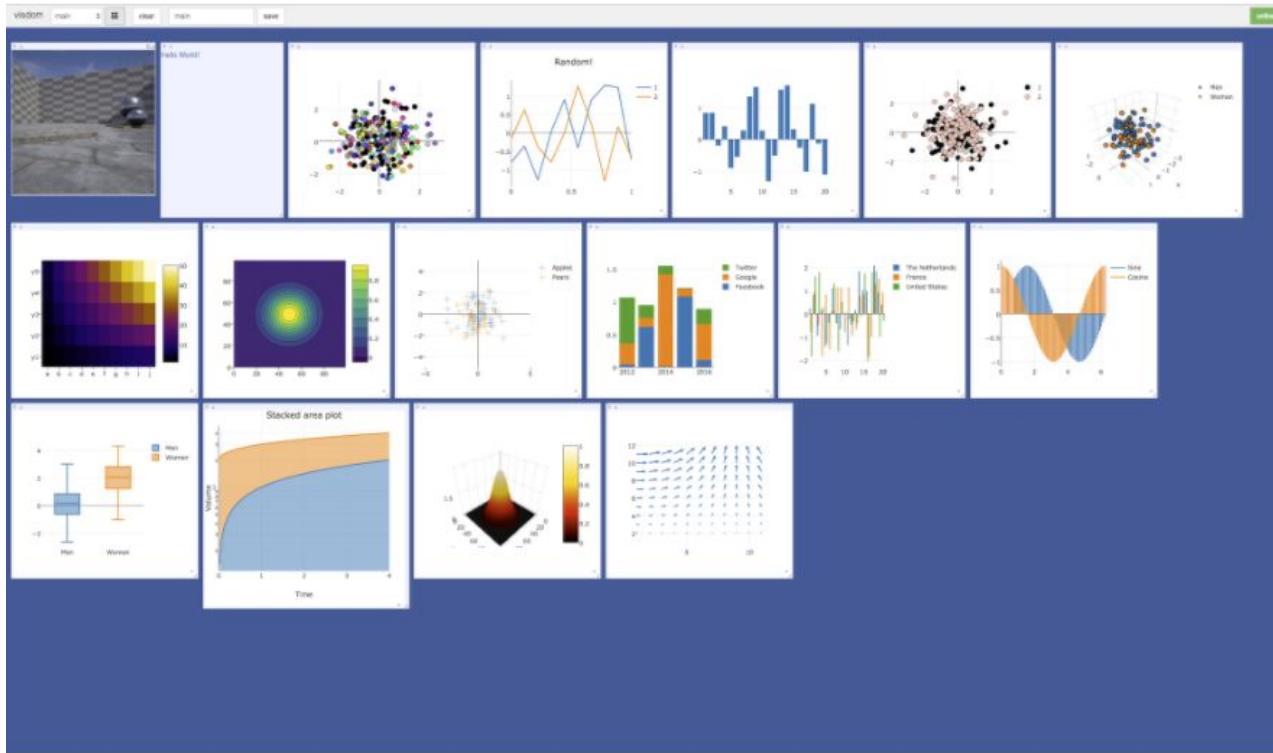
But if your input sizes changes at each iteration, then cudnn will benchmark every time a new size appears, possibly leading to worse runtime performances.

11 Likes



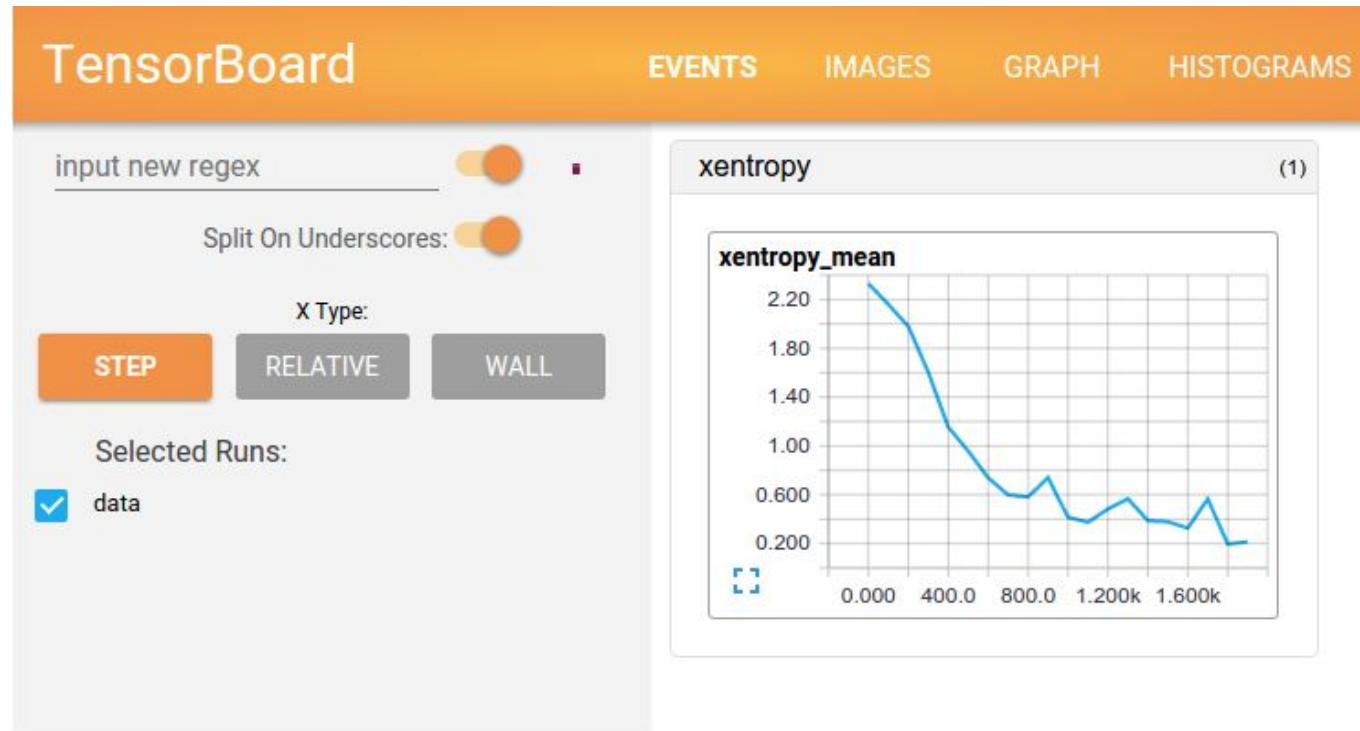
# Visualization, Pytorch Visdom

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



## Visualization, TensorBoard

[https://www.tensorflow.org/get\\_started/summaries\\_and\\_tensorboard](https://www.tensorflow.org/get_started/summaries_and_tensorboard)



# Other Resources

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

 PyTorch Tutorials  
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



[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 PyTorch documentation website. At the top, there is a navigation bar with the PyTorch logo, version 0.3.0, a 'version selector' dropdown set to '0.3.0', a search bar labeled 'Search docs', and links for 'Docs' and 'Edit on GitHub'. Below the navigation bar, the page title is 'PyTorch documentation' with a 'PyTorch' link. A sub-section title 'PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.' follows. The main content area is divided into two sections: 'Notes' and 'Package Reference'. The 'Notes' section contains a bulleted list of topics: Autograd mechanics, Broadcasting semantics, CUDA semantics, Extending PyTorch, Multiprocessing best practices, and Serialization semantics. The 'Package Reference' section lists various PyTorch modules: 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, and torch.legacy.

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 stars and 596 forks. It features tabs for Code, Issues (11), Pull requests (3), Projects (0), Wiki, and Insights. The description states: "Image-to-image translation in PyTorch (e.g. horse2zebra, edges2cats, and more)". Key tags include: 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. A recent commit by `junyanz` was made 5 hours ago. The repository structure includes data, datasets, imgs, and models.

This repository

Search

Pull requests Issues Marketplace Explore

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>

The screenshot shows a YouTube channel page for 'Sung Kim'. The channel banner features a photo of a young man giving a thumbs-up, with a large image of a coastal city skyline in the background. A caption at the bottom right of the banner reads 'ML/DL for everyone (in Korean)'. The channel has 17,457 subscribers. The navigation bar includes links for Home, Videos, Playlists (which is underlined), Community, Channels, and About. Below the navigation bar, there's a thumbnail for a playlist titled 'PyTorchZeroToAll (in English)'. The thumbnail includes a diagram showing a 'Labeled dataset' being processed by 'training' to produce a 'Model'. The description below the thumbnail states: 'Machine Learning needs lots of training' and 'Basic ML/DL lectures using PyTorch in English.' Below the thumbnail are four video thumbnails for individual lectures:

- 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