



PyTorch

2019.04.08

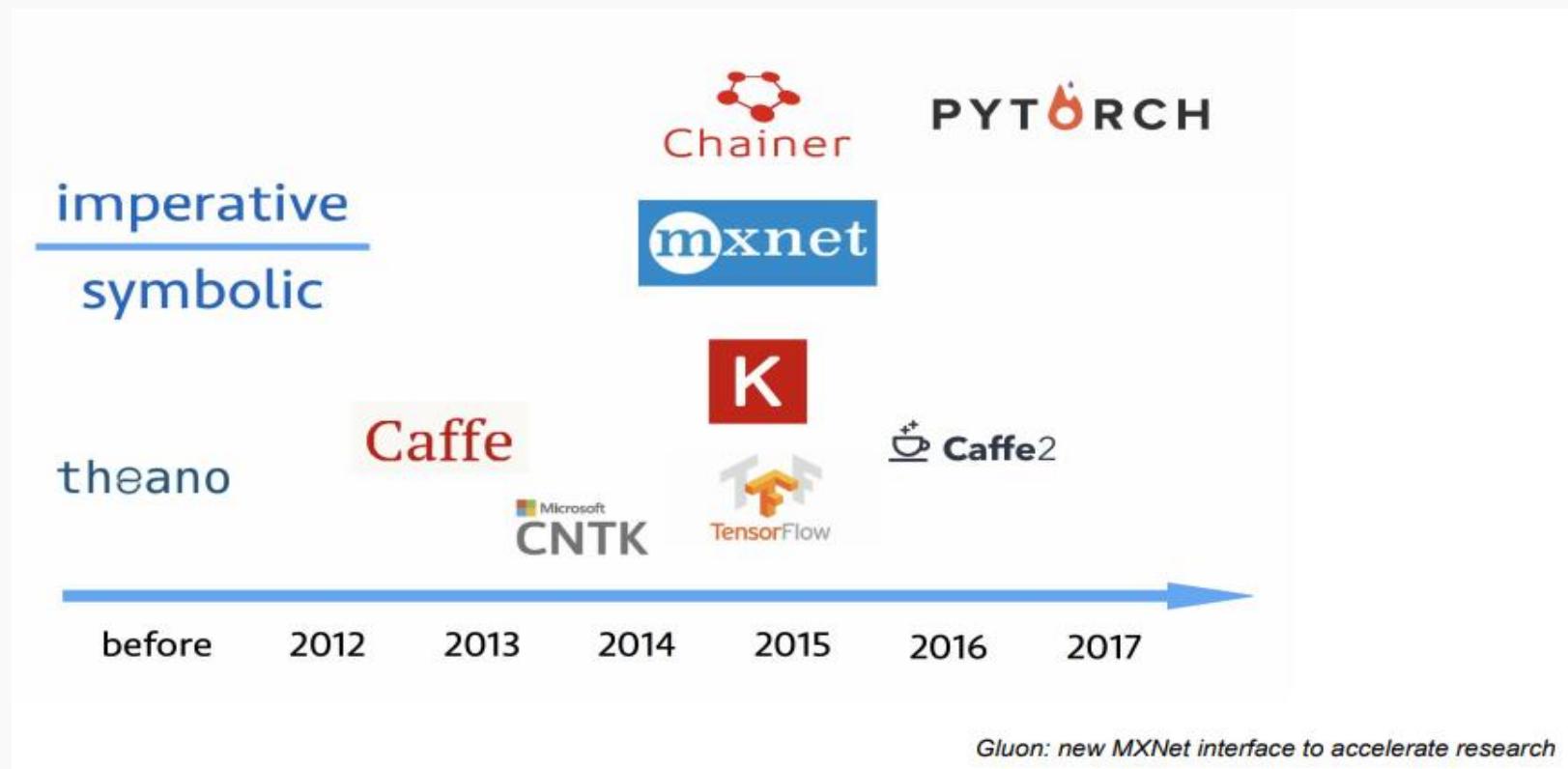
NLP LAB,
Department of Computer Engineering,
Kyung Hee University.

khuphj@gmail.com

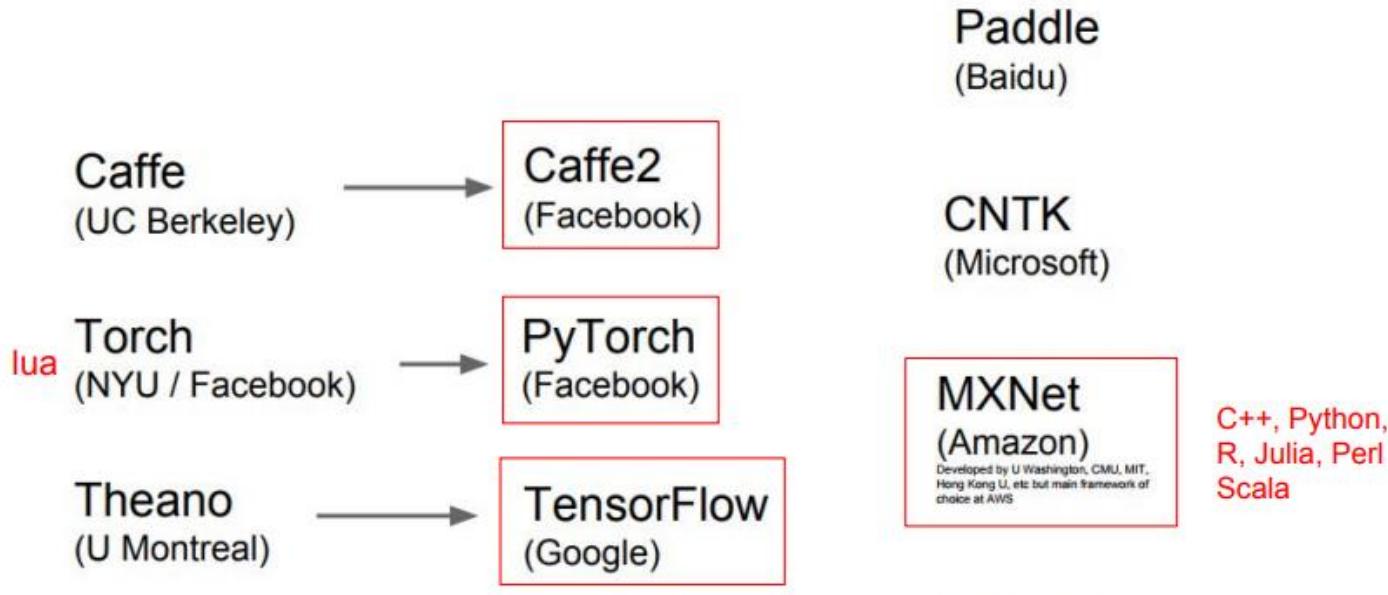
Index

- Popular DL Frameworks
- What is PyTorch ?
- Install
- Pytorch

Popular DL Frameworks (1/6)



Popular DL Frameworks (2/6)



Stanford cs231n.

Popular DL Frameworks (3/6)

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

MxNet Tutorial, CVPR 2017

Popular DL Frameworks (4/6)

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

Popular DL Frameworks (5/6)

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

Popular DL Frameworks (6/6)

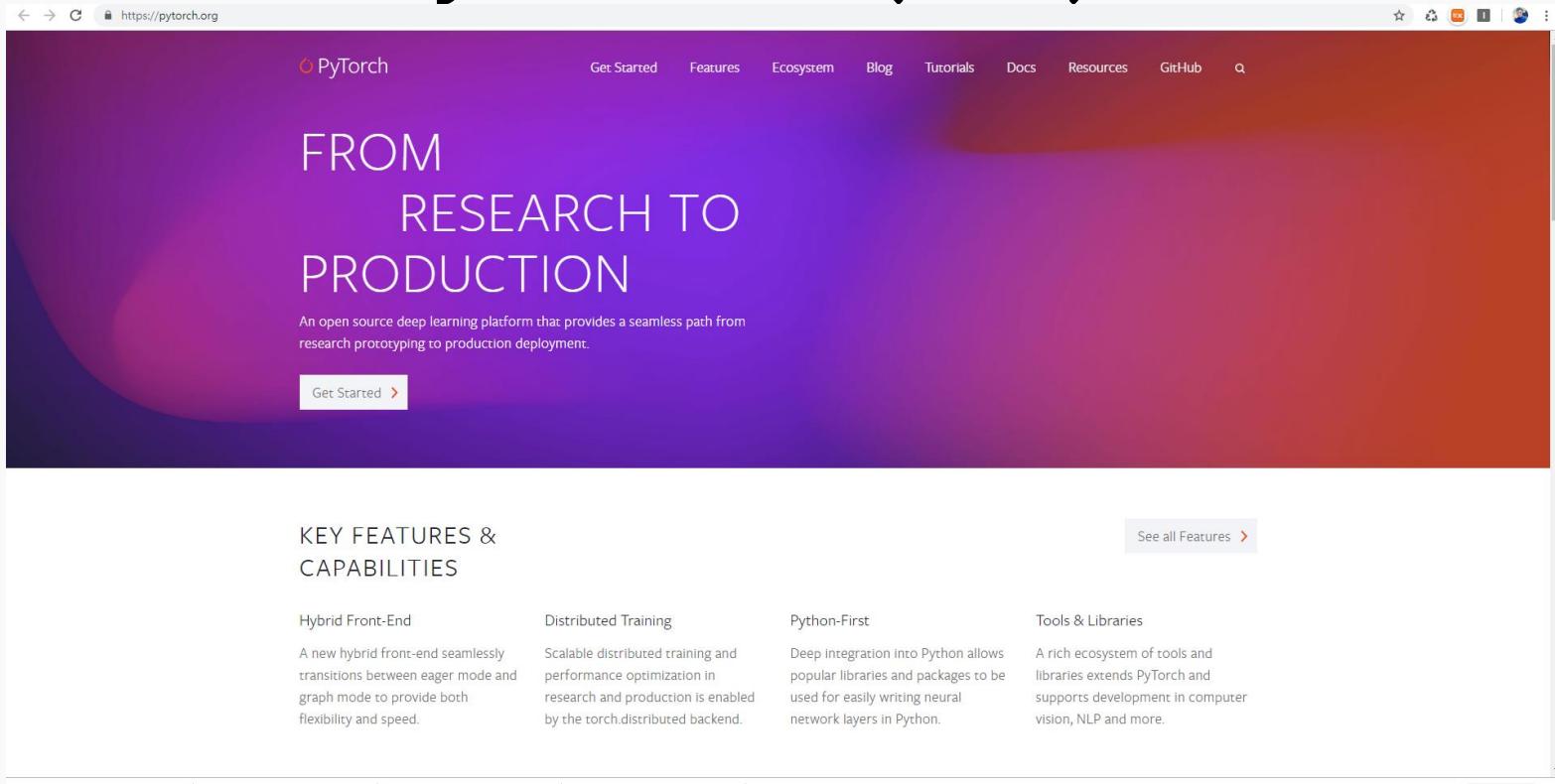
Pytorch & Chainer

```
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

What is PyTorch ? (1/9)



The screenshot shows the official PyTorch website at <https://pytorch.org>. The header features a navigation bar with links for Get Started, Features, Ecosystem, Blog, Tutorials, Docs, Resources, GitHub, and a search icon. The main title "FROM RESEARCH TO PRODUCTION" is prominently displayed in large white text against a purple-to-orange gradient background. Below the title, a subtitle reads: "An open source deep learning platform that provides a seamless path from research prototyping to production deployment." A "Get Started" button is visible. The footer contains sections for Key Features & Capabilities, including Hybrid Front-End, Distributed Training, Python-First, and Tools & Libraries, each with a brief description and a "See all Features" link.

PyTorch

Get Started Features Ecosystem Blog Tutorials Docs Resources GitHub Q

FROM RESEARCH TO PRODUCTION

An open source deep learning platform that provides a seamless path from research prototyping to production deployment.

Get Started >

KEY FEATURES & CAPABILITIES

See all Features >

Hybrid Front-End	Distributed Training	Python-First	Tools & Libraries
A new hybrid front-end seamlessly transitions between eager mode and graph mode to provide both flexibility and speed.	Scalable distributed training and performance optimization in research and production is enabled by the torch.distributed backend.	Deep integration into Python allows popular libraries and packages to be used for easily writing neural network layers in Python.	A rich ecosystem of tools and libraries extends PyTorch and supports development in computer vision, NLP and more.

What is PyTorch ? (2/9)

TensorFlow is a safe bet for most projects. Not perfect but has huge community, wide usage. Maybe pair with high-level wrapper (Keras, Sonnet, etc)

I think **PyTorch** is best for research. However still new, there can be rough patches.

Use **TensorFlow** for one graph over many machines

Consider **Caffe**, **Caffe2**, or **TensorFlow** for production deployment

Consider **TensorFlow** or **Caffe2** for mobile

From **cs231n** Lecture 8 (April 27, 2017)

What is PyTorch ? (3/9)

PyTorch or TensorFlow?

Written on August 17, 2017

This is a guide to the main differences I've found between PyTorch and TensorFlow. This post is intended to be useful for anyone considering starting a new project or making the switch from one deep learning framework to another. The focus is on programmability and flexibility when setting up the components of the training and deployment deep learning stack. I won't go into performance (speed / memory usage) trade-offs.

Summary

PyTorch is better for rapid prototyping in research, for hobbyists and for small scale projects.

TensorFlow is better for large-scale deployments, especially when cross-platform and embedded deployment is a consideration.

<https://awni.github.io/pytorch-tensorflow/>

What is PyTorch ? (4/9)

K Keras vs. PyTorch

Keras or PyTorch as your first deep learning framework

June 26, 2018 / 4 Comments / in Data Science, Deep Learning, Machine Learning / by Rafał Jakubanis and Piotr Migdal

So, you want to learn deep learning? Whether you want to start applying it to your business, base your next side project on it, or simply gain marketable skills – picking the right deep learning framework to learn is the essential first step towards reaching your goal.

We strongly recommend that you pick either Keras or PyTorch. These are powerful tools that are enjoyable to learn and experiment with. We know them both from the teacher's and the student's perspective. Piotr has delivered corporate workshops on both, while Rafał is currently learning them.

TL;DR:

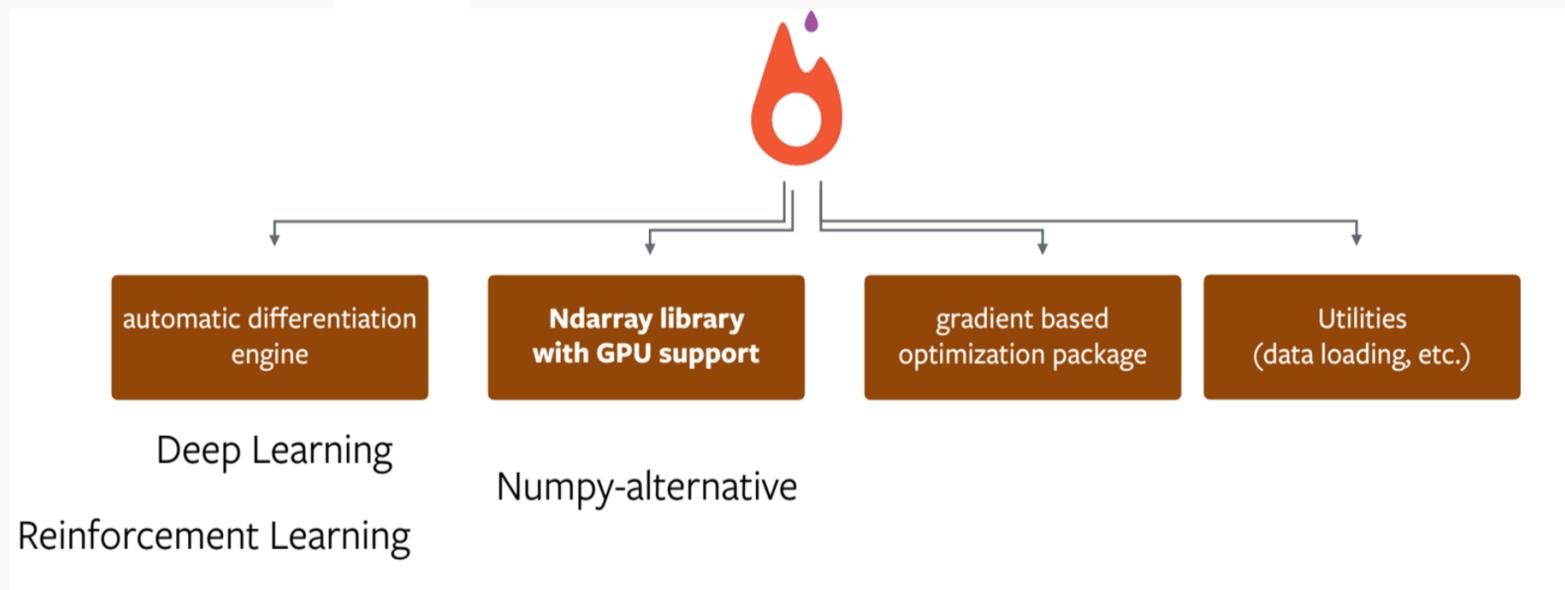
Keras may be easier to get into and experiment with standard layers, in a plug & play spirit.

PyTorch offers a lower-level approach and more flexibility for the more mathematically-inclined users.

<https://deepsense.ai/keras-or-pytorch/>

What is PyTorch ? (5/9)

- An **optimized tensor library** for deep learning using GPUs and CPUs



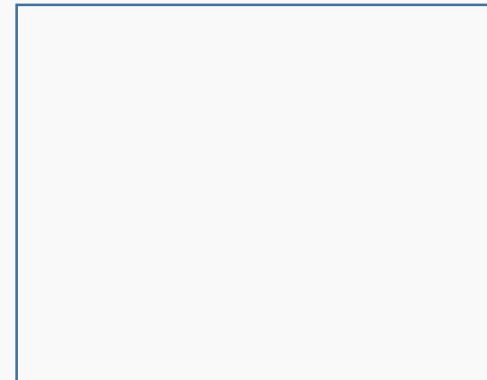
What is PyTorch ? (6/9)

- 1-dimensional **tensors** are vectors
- 2-dimensional **tensors** are matrices
- 3+ dimensional **tensors** are just referred to as tensors

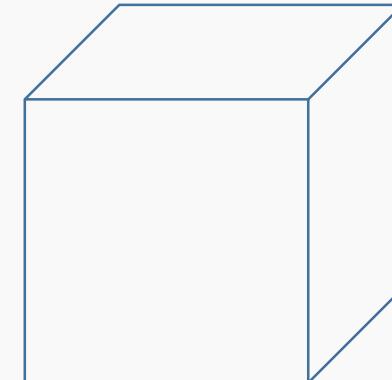
Vector



Matrix



Tensor



What is PyTorch ? (7/9)

(PyTorch) Tensor

- n-dimensional array (similar to np.ndarray)
- Provides many functions for operating on these Tensors
 - 100+ Tensor operations
 - Transposing, indexing, slicing, mathematical operations, linear algebra, random numbers, ...
 - <http://pytorch.org/docs/torch>
- PyTorch Tensor numpy array
- Fast acceleration on NVIDIA GPUs

What is PyTorch ? (8/9)

Dynamic Computational Graph

- Forward pass
 - Evaluate their predictions for given inputs
 - Define a computational graph
- Backward pass
 - Compute gradients for their parameters with respect to arbitrary scalar losses
- PyTorch Autograd!

What is PyTorch ? (9/9)

Dynamic Computational Graph

A graph is created on the fly



```
from torch.autograd import Variable  
  
x = Variable(torch.randn(1, 10))  
prev_h = Variable(torch.randn(1, 20))  
W_h = Variable(torch.randn(20, 20))  
W_x = Variable(torch.randn(20, 10))
```

Installation (1/1)

[PyTorch](https://pytorch.org)

- [Get Started](#)
- [Features](#)
- [Ecosystem](#)
- [Blog](#)
- [Tutorials](#)
- [Docs](#)
- [Resources](#)
- [GitHub](#)
- [🔍](#)

**QUICK START
LOCALLY**

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch 1.0. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, 1.0 builds that are generated nightly. Please ensure that you have met the prerequisites below (e.g., numpy), depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also [install previous versions of PyTorch](#). Note that LibTorch is only available for C++.

PyTorch Build	Stable (1.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python 2.7	Python 3.5	Python 3.6	Python 3.7
CUDA	8.0	9.0	10.0	None
Run this Command:	<pre>conda install pytorch torchvision cudatoolkit=9.0 -c pytorch</pre>			

[Previous versions of PyTorch](#)

**QUICK START WITH
CLOUD PARTNERS**

Get up and running with PyTorch quickly through popular cloud platforms and machine learning services.

 aws	AWS >
	Google Cloud Platform >
	Microsoft Azure >

- Check your OS, CUDA, Package option at <https://pytorch.org>

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

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
```

PyTorch: Tensors

Forward pass: compute predictions and loss

```
import torch

dtype = torch.FloatTensor

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

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

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

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

PyTorch: Tensors

Backward pass:
manually compute
gradients

```
import torch

dtype = torch.FloatTensor

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

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

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

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

PyTorch: Tensors

Gradient descent
step on weights

```
import torch

dtype = torch.FloatTensor

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

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

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

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

PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray,
but runs on GPU

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

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

PyTorch: Autograd

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

x.data is a Tensor

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

x.grad.data is a Tensor of gradients

```
import torch
from torch.autograd import Variable

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

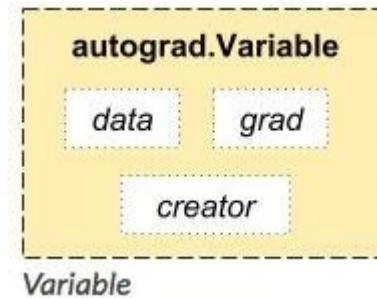
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

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

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

Variable

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



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

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

Computational Graphs

Numpy

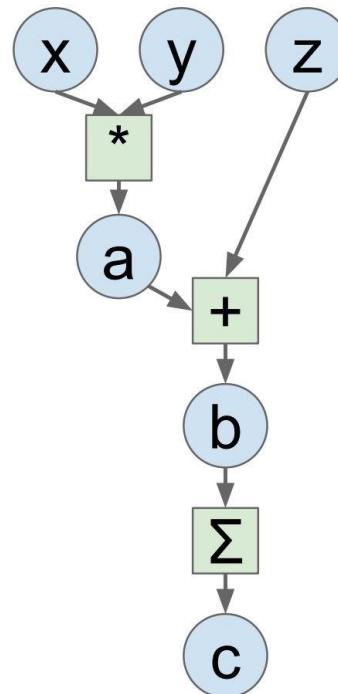
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

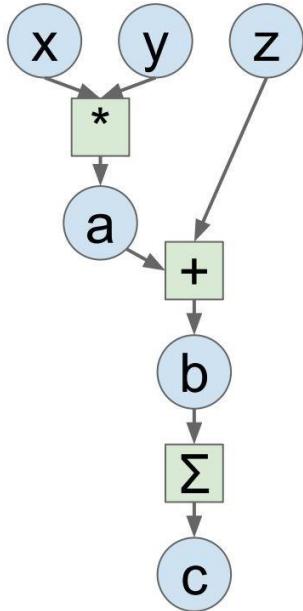
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

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

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Computational Graphs



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

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

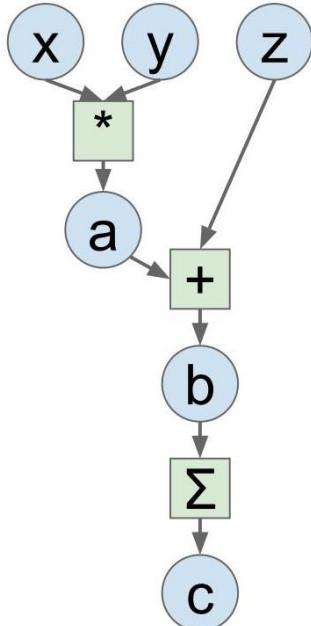
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

c.backward()

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

Computational Graphs



Forward pass
looks just like
numpy

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

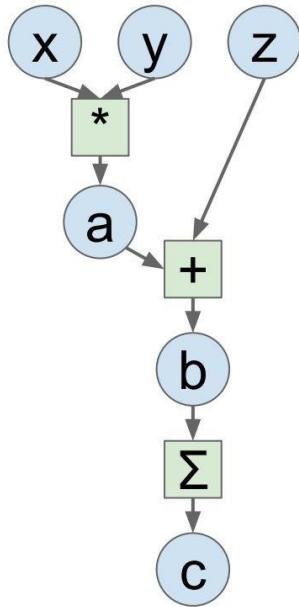
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

c.backward()

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

Computational Graphs



Calling `c.backward()`
computes all
gradients

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

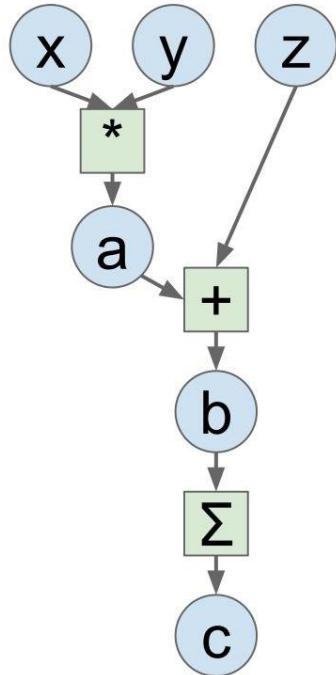
x = Variable(torch.randn(N, D),
             requires_grad=True)
y = Variable(torch.randn(N, D),
             requires_grad=True)
z = Variable(torch.randn(N, D),
             requires_grad=True)

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

c.backward()

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

Computational Graphs



Run on GPU by
casting to .cuda()

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

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

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

c.backward()

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

PyTorch: Three Levels of Abstraction

Tensor: Imperative ndarray,
but runs on GPU

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

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

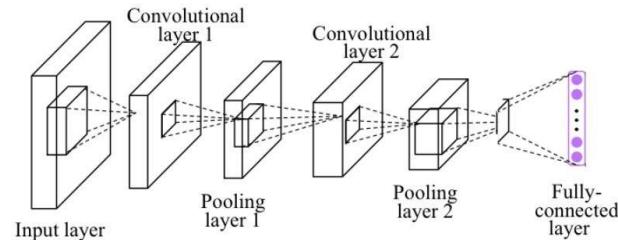
Module, single layer

The image shows three panels of the PyTorch nn module documentation:

- 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

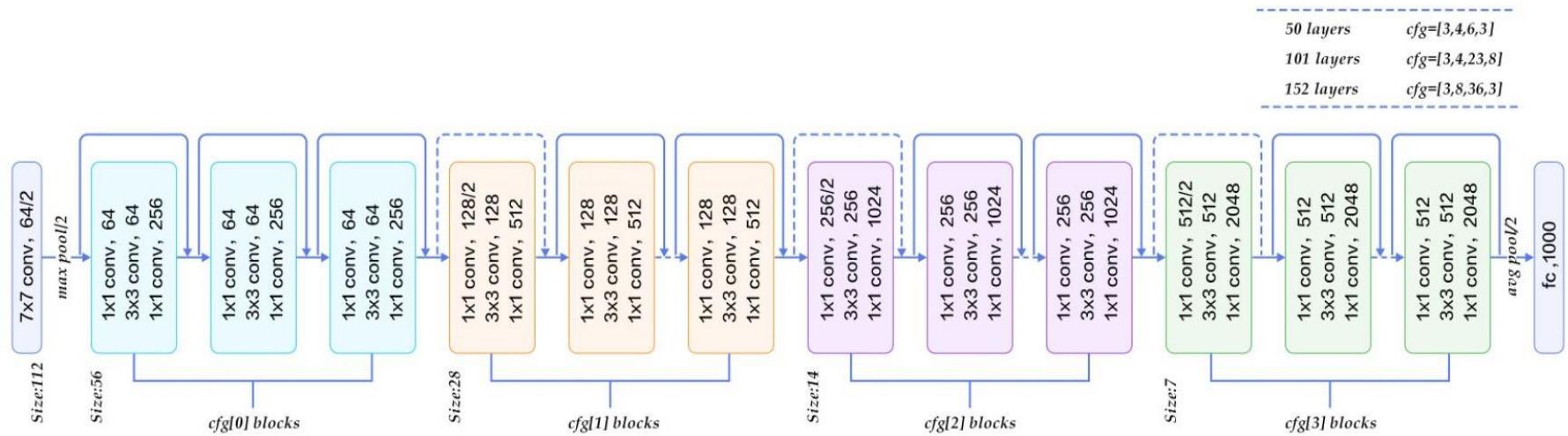


```
class Net(nn.Module):

    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



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

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,
```

<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)
    )
)
```

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)

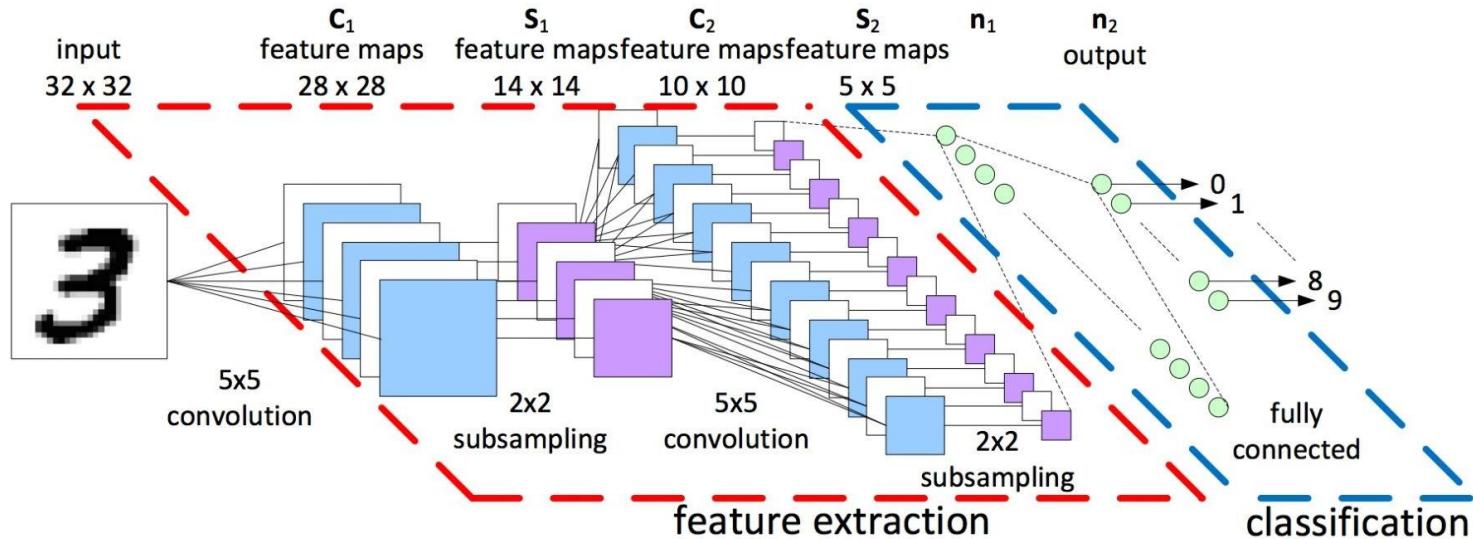
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

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)

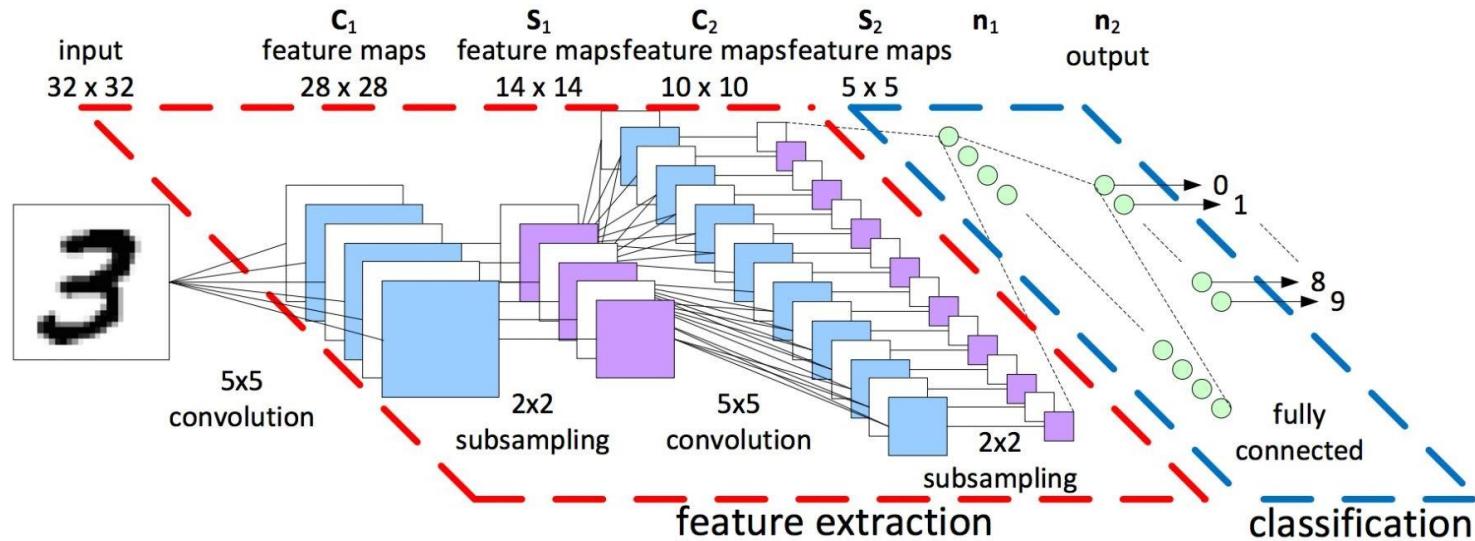
    optimizer.zero_grad()
    loss.backward()

    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

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

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)
```

<https://goo.gl/mQEw15>

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)))
```

Reference

- PyTorch Tutorials
<https://pytorch.org/tutorials/>
- PyTorch Zero To All
<https://github.com/hunkim/PyTorchZeroToAll>

Q & A