



PYTORCH

- a pythonic DL framework



>>> WHAT IS PYTORCH?

It's a Python-based scientific computing package targeted at two sets of audiences:

- *A replacement for NumPy to use the power of GPUs
- *A deep learning research platform that provides maximum flexibility and speed

facebook



ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIE
PARIS INSTITUTE OF TECHNOLOGY

Carnegie
Mellon
University



Digital
Reasoning

Stanford
University



Inria





>>> Outline

1. Installation
2. Basic Concepts
3. Autograd: Automatic Differentiation
4. Neural Networks
5. Example: An Image Classifier
6. More Adventures



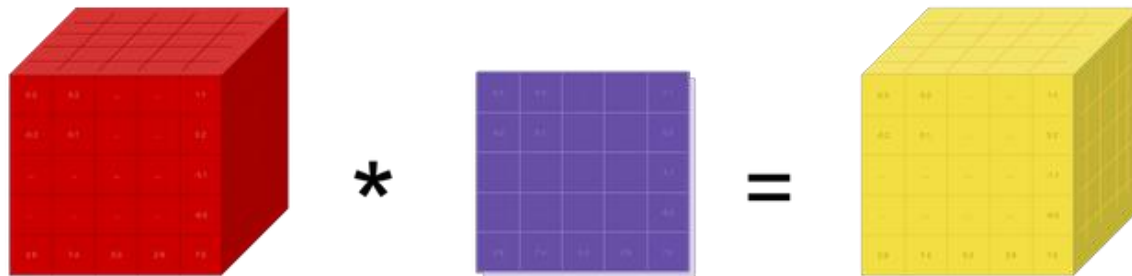
>>> Installation

PyTorch Build	Stable (1.4)		Preview (Nightly)	
Your OS	Linux	Mac		Windows
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
CUDA	9.2	10.1		None
Run this Command:	conda install pytorch torchvision -c pytorch			

<https://pytorch.org/>

- *Anaconda (recommend for new hands): easy to install and run; automatically download dependencies
- *Pip install (a flexible choice for the experienced): more fine-grained control of installation options

>>> Tensors



Tensors are similar to NumPy's ndarrays. Start with:

```
import torch
```



>>> Tensors

Initialize tensors:

```
# Construct a 5x3 matrix, uninitialized
```

```
x = torch.empty(5, 3)
```

```
# Construct a randomly initialized matrix
```

```
x = torch.rand(5, 3)
```

```
# Construct a matrix filled zeros and of dtype long
```

```
x = torch.zeros(5, 3, dtype=torch.long)
```

```
# Construct a tensor directly from data
```

```
x = torch.tensor([5.5, 3])
```



>>> Operations

Addition operation:

```
x = torch.rand(5, 3)
```

```
y = torch.rand(5, 3)
```

```
# Syntax 1
```

```
z = x + y
```

```
# Syntax 2
```

```
z = torch.empty(5, 3)
```

```
torch.add(x, y, out=z)
```

```
# In-place addition, adds x to y  y.add_(x)
```

Explore the subtraction operation(*torch.sub*),

multiplication operation(*torch.mul*), etc.

>>> Torch Tensor & NumPy Array



Convert Torch Tensor to NumPy Array:

```
a = torch.ones(5) # Torch Tensor
```

```
b = a.numpy() # NumPy Array
```

Convert NumPy Array to Torch Tensor:

```
import numpy as np
```

```
a = np.ones(5) # NumPy Array
```

```
b = torch.from_numpy(a) # Torch Tensor
```




>>> CUDA Tensors

Tensors can be moved onto any device using the `.to` method. A Nvidia GPU must be available.

```
# move the tensor to a Nvidia GPU
```

```
x = x.to("cuda")
```

```
# or
```

```
x = x.cuda()
```

```
# directly create a tensor on a Nvidia GPU
```

```
device = torch.device("cuda")
```

```
y = torch.ones_like(x, device=device)
```

```
# move the tensor to CPU    x = x.to("cpu")
```

```
# or
```

```
x = x.cpu()
```





>>> Autograd

Track all operations by setting Tensors' attribute `.requires_grad` as True:

```
x = torch.ones(2, 2, requires_grad=True)    # or
```

```
x = torch.ones(2, 2)  x.requires_grad_(True) # in-place
```

Do operations:

```
y = x + 2
```

```
z = y * y * 3  out = z.mean()
```

Let's backpropagate:

```
out.backward()
```

>>> Autograd

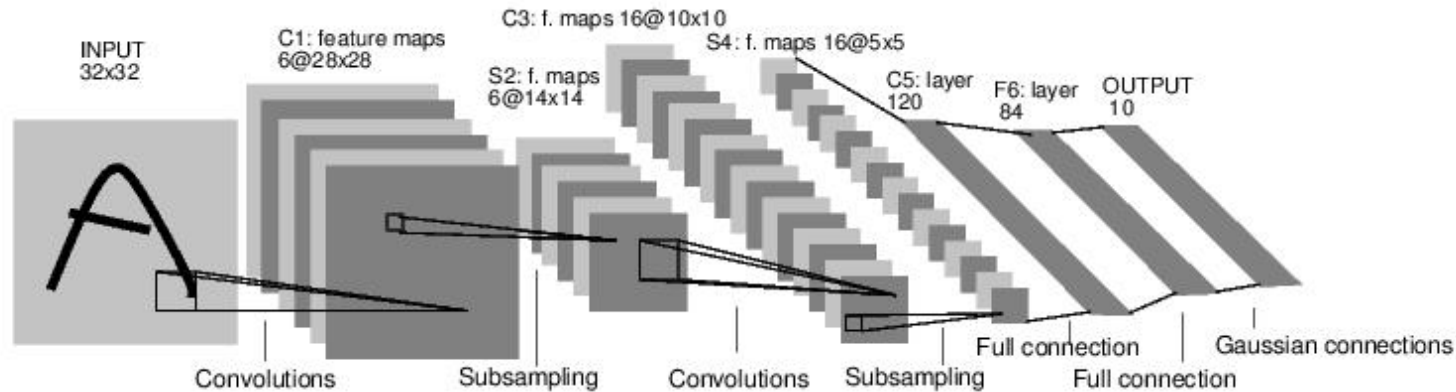


Stop autograd on Tensors with `.requires_grad=True`
by:

```
>>> print(x.requires_grad)
>>> True
```

```
with torch.no_grad():
    # Do operations on x
```

>>> Training procedure



1. Define the neural network that has some learnable parameters/weights
2. Process input through the network
3. Compute the loss (how far is the output from being correct)
4. Propagate gradients back into the network's parameters, and update the weights of the network, typically using a simple update rule:

$$\text{weight} = \text{weight} - \text{learning_rate} * \text{gradient}$$

Repeat step 2-4 by iterating over a dataset of inputs.



>>> Define the network (step 1)

Only need to define `forward` function, `backward` function is automatically defined.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```



```
>>> Define the network (step 1)
```

View the network structure:

```
>>> net = Net()  
>>> print(net)  
>>> Net(  
    (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))  
    (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))  
    (fc1): Linear(in_features=400, out_features=120, bias=True)  
    (fc2): Linear(in_features=120, out_features=84, bias=True)  
    (fc3): Linear(in_features=84, out_features=10, bias=True)  
)
```

The learnable parameters of a model are returned by `net.parameters()`.



>>> Process inputs (step 2)

Try a random input:

```
input = torch.randn(1, 1, 32, 32)
```

```
out = net(input)
```



>>> Compute the loss (step 3)

Example: **nn.MSELoss** which computes the mean-squared error between the input and the target.

```
output = net(input)
target = torch.randn(10) # a dummy target, for example
target = target.view(1, -1) # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
```

Look into several different loss functions by
<https://pytorch.org/docs/stable/nn.html>.



>>> Backprop and update the weights (step 4)

Set up an update rule such as SGD, Adam, etc,
by using torch.optim package.

```
import torch.optim as optim  
optimizer = optim.SGD(net.parameters(), lr=0.01)
```

>>> Backprop and update the weights (step 4)



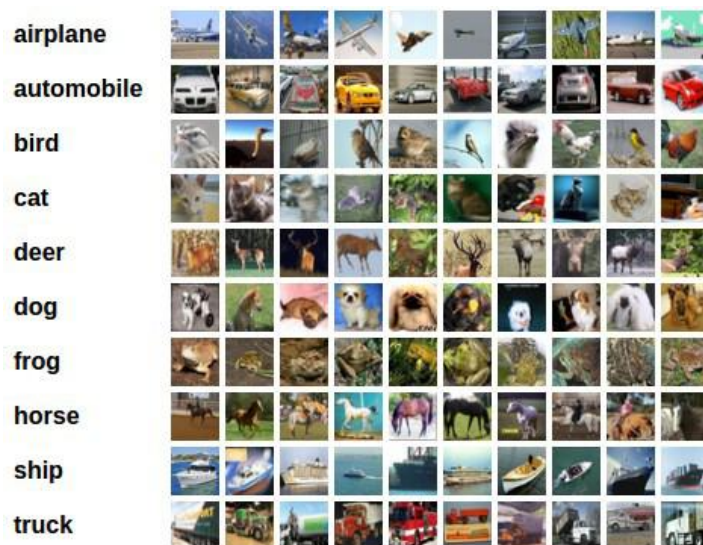
Set up an update rule such as SGD, Adam, etc, by using `torch.optim` package.

```
import torch.optim as optim  
optimizer = optim.SGD(net.parameters(), lr=0.01)
```

Then backpropagate the error and update the weights:

```
optimizer.zero_grad() # zero the gradient buffers  
loss = criterion(output, target)  
loss.backward()  
optimizer.step() # Does the update
```

>>> Training an image classifier



1. Load and normalizing the training and test datasets.
2. Define a Convolutional Neural Network
3. Define a loss function
4. Train the network on the training data
5. Test the network on the test data

>>> Load data



Deal with images,

- 1.load data into a numpy array by packages such as Pillow, OpenCV
- 2.convert this array into a torch.*Tensor
- 3.normalize data by torchvision.transforms
- 4.assign mini batches by torch.utils.data.DataLoader

Exist data loaders for common datasets such as Imagenet, CIFAR10, MNIST, etc in torchvision.datasets (replace step 1-2).



```
>>> Load data (step 1)
```

Example: Loading and normalizing CIFAR10

```
import torch  import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose( [transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                          download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                           shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                         download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                          shuffle=False, num_workers=2)
```



>>> Define the network (step 2) Same as before:

```
import torch.nn as nn
```

```
import torch.nn.functional as F
```

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

```
    def __init__(self):
```

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

```
        self.conv1 = nn.Conv2d(1, 6, 5)
```

```
        self.conv2 = nn.Conv2d(6, 16, 5)
```

```
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
```

```
        self.fc2 = nn.Linear(120, 84)
```

```
        self.fc3 = nn.Linear(84, 10)
```

```
    def forward(self, x):
```

```
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
```

```
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
```

```
        x = x.view(-1, 16 * 5 * 5)
```

```
        x = F.relu(self.fc1(x))    x = F.relu(self.fc2(x))
```

```
        x = self.fc3(x)
```

```
        return x
```

```
net = Net()
```

>>> Define a loss function and optimizer(step 3)



Use Cross-Entropy loss and SGD with momentum:

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```



>>> Train the network (step 4) Loop over data:

```
for epoch in range(2): # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward() optimizer.step()
        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999: # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f'%
                (epoch + 1, i + 1, running_loss / 2000))
    running_loss = 0.0
```


>>> Train the network (step 4)



Out:

```
[1, 2000] loss: 2.258
[1, 4000] loss: 1.877
[1, 6000] loss: 1.699
[1, 8000] loss: 1.594
[1, 10000] loss: 1.533
[1, 12000] loss: 1.475
[2, 2000] loss: 1.425
[2, 4000] loss: 1.380
[2, 6000] loss: 1.350
[2, 8000] loss: 1.347
[2, 10000] loss: 1.332
[2, 12000] loss: 1.277
```

>>> Test the network (step 5) Check prediction:



```
correct = 0
```

```
total = 0
```

```
with torch.no_grad():
```

```
    for data in testloader:
```

```
        images, labels = data
```

```
        outputs = net(images)
```

```
        _, predicted = torch.max(outputs.data, 1)
```

```
        total += labels.size(0)
```

```
        correct += (predicted == labels).sum().item()
```

```
print('Accuracy of the network on the 10000 test images:  %d'%  
      (100 * correct / total))
```

>>> Test the network (step 5) Check prediction:



```
correct = 0
```

```
total = 0
```

```
with torch.no_grad():
```

```
    for data in testloader:
```

```
        images, labels = data
```

```
        outputs = net(images)
```

```
        _, predicted = torch.max(outputs.data, 1)
```

```
        total += labels.size(0)
```

```
        correct += (predicted == labels).sum().item()
```

```
print('Accuracy of the network on the 10000 test images: %d'%  
      (100 * correct / total))
```

Out:

Accuracy of the network on the 10000 test images: 54



>>> Option: training on GPU

Transfer the network and tensors onto the GPU:

```
device = torch.device("cuda:0")  
# training on the first cuda device  
net.to(device)  
  
inputs, labels = inputs.to(device), labels.to(device)
```





>>> Option: training on multiple GPUs

You can easily run your operations on multiple GPUs by making your model run parallelly using:

```
net = nn.DataParallel(net)
```

Advantages: larger batch size, higher speed, *etc.*

>>> More Adventures



*Tutorials <https://github.com/pytorch/tutorials>

*Examples <https://github.com/pytorch/examples>

*Docs <http://pytorch.org/docs/>

*Discussions <https://discuss.pytorch.org/>

*PyTorch深度学习快速入门教程(视频)

<https://www.bilibili.com/video/av74281036>

*Handson Tutorial <https://github.com/yunjey/pytorch-tutorial>

