Introduction to PyTorch

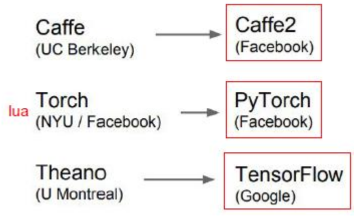
2020 - 2021

Ando Ki, Ph.D.
adki@future-ds.com

Table of contents

What is PyTorch

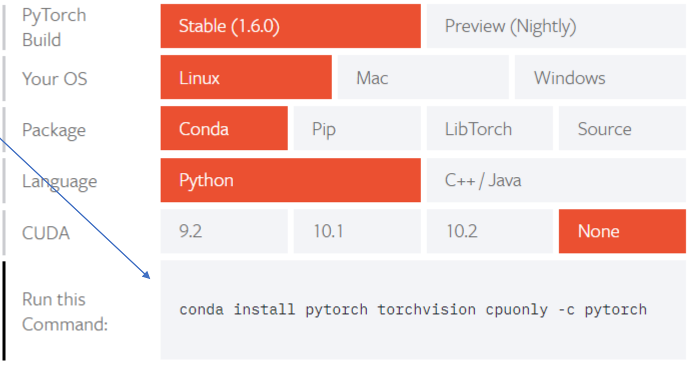
- **PyTorch** is a machine learning Python library, developed by the Facebook AI research group.
  - Based on Torch library
    - Torch is an open-source machine learning library, a scientific computing framework, and a script language based on the Lua programming language.[3] It provides a wide range of algorithms for deep learning, and uses the scripting language LuaJIT, and an underlying C implementation.
- **PyTorch** is a Python-based scientific computing package targeted at two sets of audiences:
  - A replacement for NumPy to use the power of GPUs → tensor
  - a deep learning research platform that provides maximum flexibility and speed



PyTorch

Installing PyTorch

- Visit: https://pytorch.org/get-started/locally/
- Select your preferences and run the install command.
- Run the command
  - note 'conda' is required.



PyTorch components

Component	Description
torch	a Tensor library like NumPy, with strong GPU support
torch.autograd	a tape-based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.jit	a compilation stack (TorchScript) to create serializable and optimizable models from PyTorch code
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.multiprocessing	Python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and Hogwild training
torch.utils	DataLoader and other utility functions for convenience

Package	**Description**
torch	The top-level PyTorch package and tensor library.
torch.nn	A subpackage that contains modules and extensible classes for building neural networks.
torch.autograd	A subpackage that supports all the differentiable Tensor operations in PyTorch.
torch.nn.functional	A functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations.
torch.optim	A subpackage that contains standard optimization operations like SGD and Adam.
torch.utils	A subpackage that contains utility classes like data sets and data loaders that make data preprocessing easier.
torchvision	A package that provides access to popular datasets, model architectures, and image transformations for computer vision.

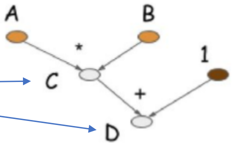
PyTorch is imperative program

- Imperative program
  - E.g., NumPy, PyTorch
  - When the line 'c=b*a' is executed, it actually runs a computation.

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

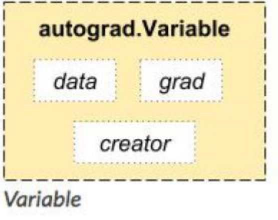
- Symbolic program
  - E.g, MXNET, TensorFlow
  - When the line 'C=B*A' is executed, no computation occurs.
  - Instead, it generates a computation graph (i.e., a symbolic graph) that represents the computation.
  - It should be compiled and then run it.

```
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)
```



Levels of abstraction

- Tensor
  - imperative ndarray can run on GPU/CPU (just like numpy array, but can run on GPU)
    - x = torch.tensor([[1,2,3],[4,5,6]])
  - Variable (it is merged to 'Tensor')
    - a wrapper around PyTorch tensor, and represents a computational graph and stores data and gradient
    - class: "torch.autograde.Variable"
      - x = torch.autograd.Variable(x, required_grad=False)
- Module
  - a neural network layer and can store state or learnable weights



PyTorch tensor v.s. NumPy ndarray

- PyTorch 's **tensors** are very similar to NumPy's **ndarrays**, but they have a device, 'cpu', 'cuda',

```
>>> t = torch.tensor([1,2,3], device='cpu',
... requires_grad=False,dtype=torch.float32)
>>> print(t.dtype)
torch.float32
>>> print(t.device)
cpu
```

```
>>> a = numpy.ones((3,4))
>>> t = torch.from_numpy(a) # get tensor

>>> b = torch.tensor([3,4])
>>> n = b.numpy() # get numpy

>>> c = torch.ones((3,4))
```

Torch data types

Data Type	T.dtype	T.dtype alias	T.tensor() CPU alias	T.tensor() GPU alias
16-bit floating point	T.float16	T.half	T.HalfTensor()	T.cuda.HalfTensor()
32-bit floating point	T.float32	T.float	T.FloatTensor()	T.cuda.FloatTensor()
64-bit floating point	T.float64	T.double	T.DoubleTensor()	T.cuda.DoubleTensor()
8-bit unsigned integer	T.unint8		T.ByteTensor()	T.cuda.ByteTensor()
8-bit signed integer	T.int8		T.CharTensor()	T.cuda.CharTensor()
16-bit signed integer	T.int16	T.short	T.ShortTensor()	T.cuda.ShortTensor()
32-bit signed integer	T.int32	T.int	T.IntTensor()	T.cuda.IntTensor()
64-bit signed integer	T.int64	T.long	T.LongTensor()	T.cuda.LongTensor()
Boolean	T.bool		T.BoolTensor()	T.cuda.BoolTensor()

Default data type: torch.FloatTensor

PyTorch functions, dimensionality

- x.size() #* *return tuple-like object of dimensions, old codes*
- x.shape # *return tuple-like object of dimensions, numpy style*
- x.ndim # *number of dimensions, also known as .dim()*
- x.view(a,b,...) #* *reshapes x into size (a,b,...)*
- x.view(-1,a) #* *reshapes x into size (b,a) for some b*
- x.reshape(a,b,...) # *equivalent with .view()*
- x.transpose(a,b) # *swaps dimensions a and b; only two dimenstions*
- x.permute(*dims) # *permutes dimensions; missing in numpy; more than two dim.*
- x.unsqueeze(dim) # *tensor with added axis; missing in numpy*
- x.unsqueeze(dim=2) # *(a,b,c) tensor -> (a,b,1,c) tensor; missing in numpy*

Tips for Torch

```
T = torch.tensor( data=x, dtype=torch.int32, device='cuda', required_grad='false')
```

- data: actual data in list or array
- dtype: data type
- device: cpu or cuda device
- required_grad: store gradient or not

how to create tensor

```
Y = torch.as_tensor(T, dtype=torch.half, device='cpu')
Z = T.to(device='cpu', dtype=torch.float64)
```

how to change attributes

```
A = Z.abs()
B = Z.abs_()
```

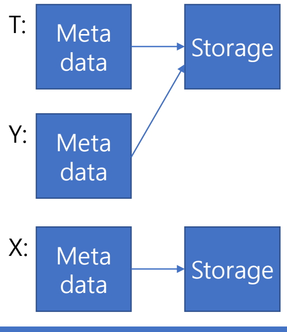
A and Z are different

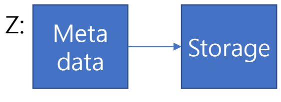
B and Z shared the same data storage; make in-place tensor; mutable (Note '_' underscore)

Tips for Torch

```
T = torch.tensor([1.,2.,3.], required_grad=True)
X = T.clone()
Y = T.detach()
Z = T.clone().detach()
```

- clone() returns an identical tensor with new memory.
  - immutable
- detach() return an identical tensor with the same memory.
  - mutable
  - not involved gradient calculation

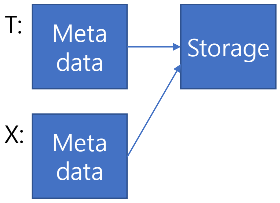




# Operation	New/Shared memory	Still in computation graph
tensor.clone()	New	Yes
tensor.detach()	Shared	No
tensor.clone().detach()	New	No

Tips for Torch

```
>>> T = torch.zeros(3,2) # all zeroes
>>> X = T.view(2,3) # transpose style
>>> T.fill_(1) # in-place
tensor([[1,1],[1,1],[1,1]])
>>> X
tensor([[1,1,1],[1,1,1]])
```



- view() returns a new shape of tensor with the same memory.
  - mutable
  - working for contiguous storage
- reshape() returns a copy or a view of the original tensor.
  - working for contiguous or non-contiguous tensor.
  - use clone() if a copy is required.
  - use view() if the same storage.

Building network (1/3): using tensor

- Two-layer net using PyTorch tensors.
  - create random tensors for data and weights
  - Forward pass
    - compute predictions and loss
  - Backward pass
    - manually compute gradients
  - 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
```

Building network (2/3): using Variable

- Two-layer net using PyTorch Variable
  - create random variables for data and weights
  - Forward pass
    - compute predictions and loss
  - Backward pass
    - manually compute gradients
  - Gradient descent step on weights

```
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
```

ReLU

SSE

Building network (3/3): using Variable

- Building network using Variable with user defined Autograd function
  - Define own autograd function by writing forward and backward for Tensors.

```
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

```
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):
    relu = ReLU()
    y_pred = relu(x.mm(w1)).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
```

Building network: creating the model using nn (1/2)

- A **model** is of a *nn.Module class* type. A model can contain other models.
  - The nn.Module's weights as called "Parameters".
  - A nn.Module consists of an ***initialization*** of the Parameters and a ***forward function***.

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        # structure definition and initialization
    def forward(self, x):
        # actual forward propagation
        result = processing(x)
        return result
```

You don't need to define a backward function!
- Autograded variable will take care of it.

Forward is called many times, expensive objects should not be recreated.

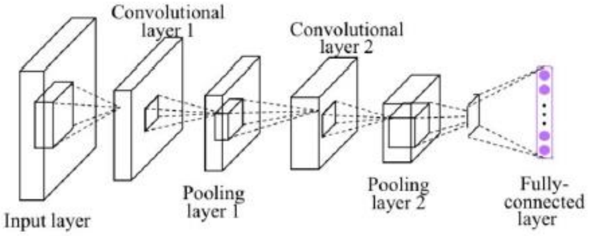
Building network: creating the model using nn (2/2)

- Neural networks can be constructed using the torch.nn package.

- Forward
  - An nn.Module contains layers, and a method forward(input) that returns the output
  - You can use any of the Tensor operations in the forward function

- Backward
  - nn depends on autograd to define models and differentiate them
  - You just have to define the forward function, and the backward function (where gradients are computed) is automatically defined for you using autograd

An example of model




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

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)
    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
        x = self.fc(x)
        return F.log_softmax(x)
```


Defining an optimizer

```
import torch
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
...
for sample in dataloader:
    input = sample['input'].to(device)
    target = sample['target'].to(device)
    prediction = model(input)
    loss = loss_fn(prediction, target) # target means expected
    optimizer.zero_grad() # clears the gradients
    loss.backward()
    optimizer.step() # performs the optimization
```


Save/load models

- Saving and loading can easily be don using “torch.save” and “torch.load”
- PyTorch uses “pickling” to serialize the data.

```
>>> state = {'model_state' : model.state_dict(),
             'optimizer': optimizer.state_dict)}
>>> torch.save(state, 'state.pth')
```

```
>>> model = Model()
>>> optimizer = optim.SGD(model_parameters(), lr=0.01)
>>> checkpoint = torch.load('state.pth')
>>> model.load_state_dict(checkpoint['model_state'])
>>> optimizer.load_state_dict(checkpoint['optimizer_state'])
```


Save/load models

- First approach
  - #save only the model parameters
  - torch.save(the_model.sate_dict(), PATH)

  - #load only the model parameters
  - the_model = TheModelCalss(*args, **kwargs)
  - the_model.load_state_dic(torch.load(PATH))
- Second approach
  - #save the entire model
  - torch.save(the_model, PATH)

  - # load the entire model
  - the_model = torch.load(PATH)