

Deep Learning Frameworks / PyTorch

- ▶ What is a deep learning framework?
 - ▶ abstract certain things away to faster develop and test new ideas.
 - ▶ automatically compute gradients!
 - ▶ run it on efficiently on a GPU
 - ▶ Frameworks: PyTorch, TensorFlow, MXNet, CNTK, ...
- ▶ Why PyTorch?
 - ▶ very similar to Numpy, hence, beginner friendly.
 - ▶ great for fast and flexible development.

PyTorch

► How to install?

<https://pytorch.org/get-started/locally/>

► Recommended to install with Anaconda

PyTorch Build	Stable		Preview		
Your OS	Linux		Mac		Windows
Package	Conda	Pip		LibTorch	Source
Language	Python 2.7	Python 3.5	Python 3.6	Python 3.7	C++
CUDA	8.0	9.0	9.2		None
Run this Command:	conda install pytorch torchvision -c pytorch				

Basic Computations

Tensor

► Construct a Tensor

```
x = torch.zeros(8, 3)
print(x)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

```
x = torch.rand(8, 3)
print(x)
```

```
tensor([[0.0984, 0.3671, 0.2543],
        [0.5016, 0.8017, 0.0918],
        [0.5081, 0.6020, 0.0867],
        [0.5313, 0.4571, 0.1624],
        [0.4231, 0.3993, 0.2713],
        [0.8978, 0.6039, 0.8519],
        [0.4829, 0.6648, 0.8295],
        [0.9060, 0.2132, 0.4110]])
```

Basic Computations

Operations

- Multiple syntaxes, e.g. Addition

```
y = torch.rand(8, 3)
```

```
print(x + y)
```

```
print(torch.add(x, y))
```

```
# providing an output tensor as argument
```

```
result = torch.empty(8, 3)
```

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

```
print(result)
```

```
# adds x to y
```

```
y.add_(x)
```

```
print(y)
```

- Other operations including transposing, indexing, slicing, linear algebra etc. at <https://pytorch.org/docs/stable/torch.html>

Basic Computations

Bridge to Numpy

- ▶ PyTorch \rightarrow Numpy

```
a = torch.ones(5)
b = a.numpy()
```

- ▶ Numpy \rightarrow PyTorch

```
a = np.ones(5)
b = torch.from_numpy(a)
```

- ▶ Tensors can only be converted to Numpy when they are on CPU

Basic Computations

Difference to Numpy

► GPU acceleration

```
if torch.cuda.is_available():  
    x = x.cuda()  
    y = y.cuda()  
    z = x + y  
    print(z)  
    print(z.cpu())           # ‘.to’ can change dtype
```

```
tensor([2.9218], device='cuda:0')  
tensor([2.9218], dtype=torch.float64)
```

Basic Computations

Difference to Numpy

- ▶ GPU acceleration
- ▶ Automatic differentiation for all operations on Tensors

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()

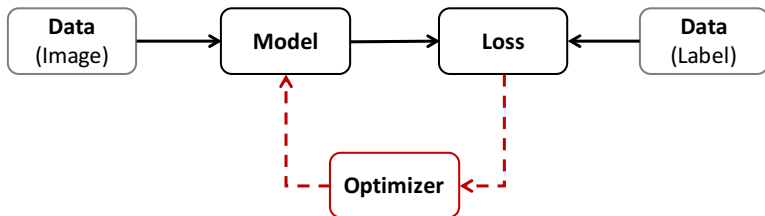
out.backward()

print(x.grad)
```

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

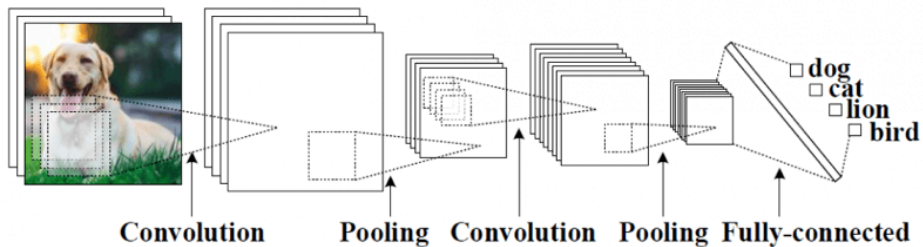
Neural Networks

- ▶ Model → `torch.nn.Module`
- ▶ Loss → `torch.nn.Module.Loss`
- ▶ Optimizer → `torch.optim`
- ▶ Data → `torch.utils.data`



Neural Networks

Model



Neural Networks

Model

- ▶ Define a model as a class that inherits from `torch.nn.Module`

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

- ▶ Define layers in the `__init__()` method

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

- ▶ Define computation flow given an input `x` in the `forward()` method

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

- ▶ `backward()` is automatically defined

Neural Networks

Model

```
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)

    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)
        return x
```

Neural Networks

Model

- ▶ PyTorch contains a bunch of standard layers which are also subclasses of `torch.nn.Module`:

▶ Convolution layers	<code>nn.Conv2d(C_{in}, C_{out}, K)</code>
▶ Pooling layers	<code>nn.MaxPool2d(K)</code>
▶ Non-linear activations	<code>nn.ReLU()</code>
▶ Normalization layers	<code>nn.BatchNorm2d(N)</code>
▶ Linear layers	<code>nn.Linear(C_{in}, C_{out})</code>
▶

- ▶ It is also easy to implement custom layers:

<https://pytorch.org/docs/stable/notes/extending.html#extending-torch-nn>

Neural Networks

Loss

- ▶ Loss function returns a non-negative value J measuring the distance between network estimation and the ground truth
- ▶ PyTorch contains a branch of loss functions which are also subclasses of `torch.nn.Module`:
 - ▶ `L1Loss`
 - ▶ `MSELoss`
 - ▶ `CrossEntropyLoss`
 - ▶ `NLLLoss`
 - ▶ `SmoothL1Loss`
 - ▶ ...

Neural Networks

Loss

- Example of using a loss function

```
loss = nn.CrossEntropyLoss()
input = torch.randn(3, 5, requires_grad=True)
target = torch.empty(3, dtype=torch.long).random_(5)
output = loss(input, target)
output.backward()
```

Neural Networks

Optimizer

- ▶ Optimizer decides how to update the parameters in the model, e.g.

$$\theta = \theta - \eta \nabla J(\theta)$$

- ▶ PyTorch implements a set of optimization algorithms in `torch.optim`:
 - ▶ SGD
 - ▶ Adam
 - ▶ ...

Neural Networks

Optimizer

- ▶ 1. Construct an Optimizer

```
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
```

- ▶ 2. Take an optimization step for every batch/sample

```
for input, target in dataset:
    # clear saved gradients before computing gradient for the new batch
    optimizer.zero_grad()

    output = model(input)
    loss = loss_fn(output, target)

    loss.backward()

    # update parameters in model
    optimizer.step()
```


Neural Networks

Data

- ▶ PyTorch provides `Dataset`, `DataLoader` in `torch.utils.data` that allows batching data, shuffling data and load data with multiple processes.
- ▶ For small scale of dataset it is fine to implement your own data loader.

Neural Networks

Saving Models

- Save/Load `state_dict` (Recommended)

```
# save
torch.save(model.state_dict(), PATH)

# load
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

- Save/Load entire model

```
# save
torch.save(model, PATH)

# load
# Model class must be defined somewhere
model = torch.load(PATH)
model.eval()
```





References

- ▶ PyTorch official tutorials:
<https://pytorch.org/tutorials/>
- ▶ Stanford Course on Deep Learning for Computer Vision:
http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf
- ▶ PyTorch tutorial with code examples:
<https://github.com/MorvanZhou/PyTorch-Tutorial>

Google Colab

Good source for you to use a free GPU.

- ▶ **Where to start?** Go to: <https://colab.research.google.com/>
- ▶ **How to use?** Go to "Upload", select the lesson notebook. Everything else is like in a standard Jupyter notebook (even most shortcuts).

Examples Recent Google Drive GitHub Upload				
Filter notebooks 				
Title		First opened	Last opened	
	Welcome To Colaboratory	5 days ago	0 minutes ago	

- ▶ **How to use a GPU?** go the colab menu tab "Runtime", select "Change runtime type", select "GPU", restart the notebook

Hyperparameter Tuning

- ▶ Number of epochs / gradient descent steps
- ▶ Preprocessing / augmentation parameters
- ▶ Optimizer and related parameters (e.g. learning rate)
- ▶ Model and related parameters (e.g. number of layers, filter size, etc)
- ▶ Some random tricks, like

torch.manual_seed(3407) is all you need: On the influence of random seeds in deep learning architectures for computer vision

David Picard

LIGM, École des Ponts, 77455 Marnes la vallée, France

DAVID.PICARD@ENPC.FR

Questions?