Tutorial on Pytorch

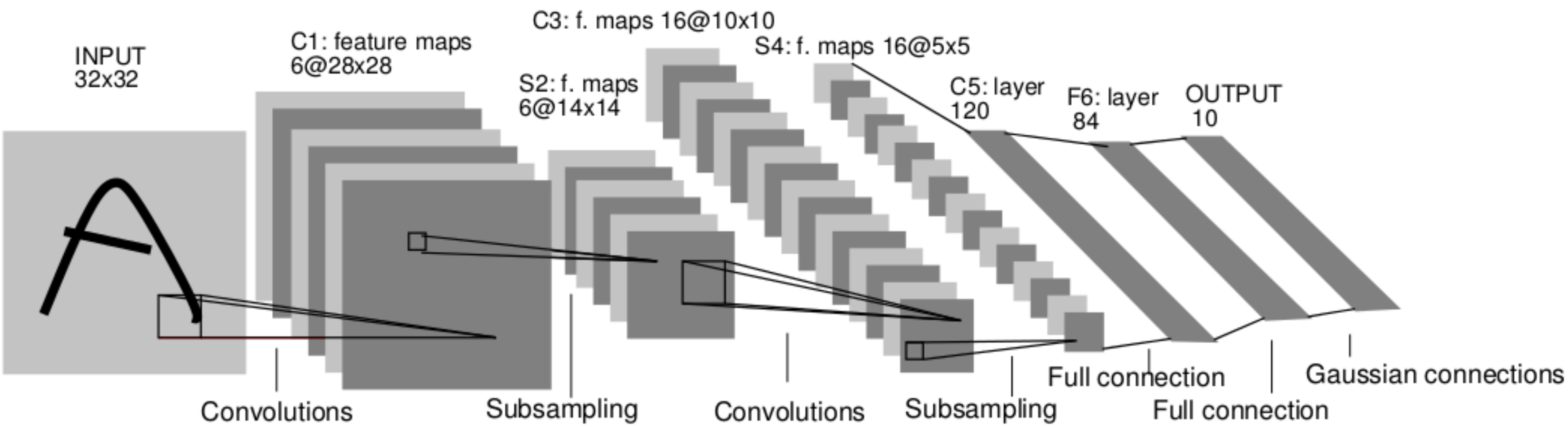
Part 1

Review: CNN architecture

Classic CNN architecture:

Output layer

Convolution layer 1 | Activation 1 | Pooling layer 1 | .... | Convolution layer n | Activation n | Pooling layer n | Fully connected (FC) layers



Roadmap

- PyTorch basics
- PyTorch CNN

Install PyTorch in CPU mode

1) Install Anaconda (python environment magnager)
2) Browse pytorch.org
3) Select and get installation command:

PyTorch Build	Stable (1.11.0)	Preview (Nightly)	LTS (1.8.2)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 10.2	CUDA 11.3	~~ROCm 4.2 (beta)~~	CPU
Run this Command:	`conda install pytorch torchvision torchaudio cpuonly -c pytorch`			

4) Create a python environment with Anaconda

```
conda create -n your_env_name python=x.x
```

5) Activate the environment `Windows: activate your_env_name`
6) Install PyTorch with the command

Install PyTorch in CPU mode

- *Alternatively you can install PyTorch offline
- *You can install in GPU mode for fast training
- Recommended IDE:
    - PyCharm community (free)
    - VSCode (free)

Check if installation is successful:

```
>>> import torch
>>> import torchvision
>>> print(torch.__version__)
1.10.2
```

What's 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
  - PyTorch CPU is slow, but it's easier to install and good for learning (Coding is conceptually the same as GPU)

Tensor

- Tensors are similar to NumPy’s ndarray
- Tensors support fast computation using GPU

Construct a 5x3 matrix:

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

Out:

```
>>> x = torch.empty(5,3)
>>> print(x)
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
>>>
```

Tensor

Construct a randomly initialized matrix:

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

Out:

```
tensor([[0.6519, 0.6639, 0.5846],
        [0.5429, 0.5386, 0.6401],
        [0.5687, 0.1522, 0.1158],
        [0.8838, 0.4172, 0.3364],
        [0.4867, 0.5550, 0.8775]])
```

Construct a matrix filled zeros and of dtype long:

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

Out:

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

Construct a tensor directly from data:

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

Out:

```
tensor([5.5000, 3.0000])
```

Tensor

Create a tensor based on an existing tensor. These methods will reuse properties of the input tensor, e.g. dtype, unless new values are provided by user

```
x = x.new_ones(5, 3, dtype=torch.double)      # new_* methods take in sizes
print(x)

x = torch.randn_like(x, dtype=torch.float)    # override dtype!
print(x)                                      # result has the same size
```

Out:

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[ 0.5863, -1.1109,  0.5939],
        [-0.4763, -1.5667, -1.1475],
        [ 0.0997,  0.8229,  0.2515],
        [ 3.2231, -1.0759,  0.0389],
        [ 0.5416, -0.6303,  0.9335]])
```

Get its size: `print(x.size())`

Out:

```
torch.Size([5, 3])
```

Operations

There are multiple syntaxes for operations.

Addition: syntax 1

```
y = torch.rand(5, 3)
print(x + y)
```

```
tensor([[ 0.7257, -0.6739,  1.4463],
        [ 0.2272, -0.8705, -0.2774],
        [ 0.4370,  1.7662,  0.5940],
        [ 3.3525, -0.6376,  0.5068],
        [ 0.6681, -0.0581,  1.7418]])
```

Addition: providing an output tensor as argument

```
result = torch.empty(5, 3)
torch.add(x, y, out=result)
print(result)
```

```
tensor([[ 0.7257, -0.6739,  1.4463],
        [ 0.2272, -0.8705, -0.2774],
        [ 0.4370,  1.7662,  0.5940],
        [ 3.3525, -0.6376,  0.5068],
        [ 0.6681, -0.0581,  1.7418]])
```

Addition: syntax 2

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

```
tensor([[ 0.7257, -0.6739,  1.4463],
        [ 0.2272, -0.8705, -0.2774],
        [ 0.4370,  1.7662,  0.5940],
        [ 3.3525, -0.6376,  0.5068],
        [ 0.6681, -0.0581,  1.7418]])
```

Addition: in-place

```
y.add_(x) # adds x to y
print(y)
```

```
tensor([[ 0.7257, -0.6739,  1.4463],
        [ 0.2272, -0.8705, -0.2774],
        [ 0.4370,  1.7662,  0.5940],
        [ 3.3525, -0.6376,  0.5068],
        [ 0.6681, -0.0581,  1.7418]])
```

Operations

- Any operation that mutates a tensor in-place is post-fixed with an _. For example: x.copy_(y), x.t_(), will change x.
- You can use standard NumPy-like indexing with all bells and whistles!

```
print(x[:, 1])
```

Output: tensor([-1.1109, -1.5667, 0.8229, -1.0759, -0.6303])

- Resizing: If you want to resize/reshape tensor, you can use torch.view

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
print(x.size(), y.size(), z.size())
```

Output: torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])

PyTorch supports 100+ Tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random numbers, etc., are described here.

NumPy bridge

Converting a Torch Tensor to a NumPy array and vice versa is a breeze. The Torch Tensor and NumPy array will share their underlying memory locations (if the Torch Tensor is on CPU), and changing one will change the other.

Converting a Torch Tensor to a NumPy Array:

```
a = torch.ones(5)
print(a)
```

```
b = a.numpy()
print(b)
```

```
a.add_(1)
print(a)
print(b)
```

```
tensor([1., 1., 1., 1., 1.])
```

```
[1. 1. 1. 1. 1.]
```

```
tensor([2., 2., 2., 2., 2.])
[2. 2. 2. 2. 2.]
```

Converting NumPy array to Torch tensor

See how changing the np array changed the Torch Tensor automatically:

```
import numpy as np

a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

Out:

```
[2. 2. 2. 2. 2.]
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
```

CUDA tensors

Tensors can be moved onto any device using the .to method.

```
# let us run this cell only if CUDA is available
# We will use ``torch.device`` objects to move tensors in and out of GPU
x = torch.randn(1)
print(x)
if torch.cuda.is_available():
    device = torch.device("cuda")          # a CUDA device object
    y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
    x = x.to(device)                       # or just use strings ``.to("cuda")``
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # ``.to`` can also change dtype together!
```

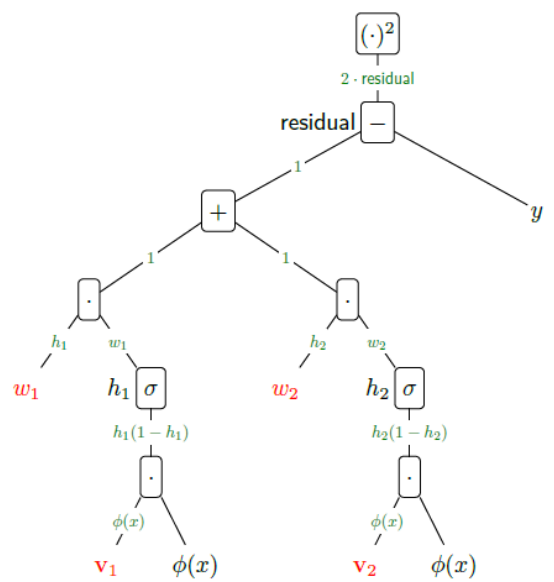
Out:

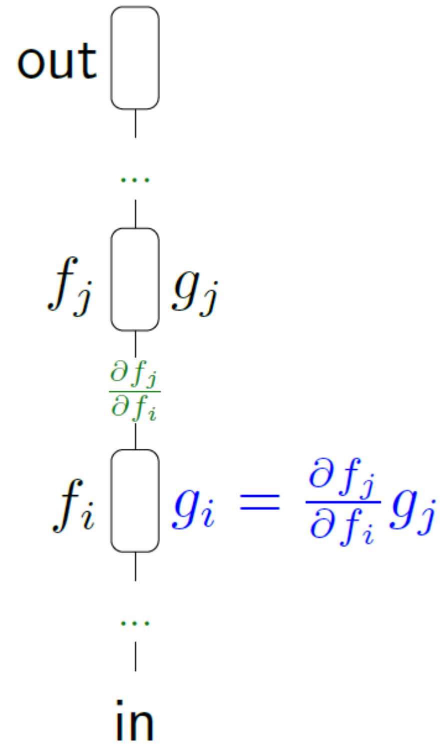
```
tensor([0.8843])
tensor([1.8843], device='cuda:0')
tensor([1.8843], dtype=torch.float64)
```

PyTorch auto-gradient

- Tensors in PyTorch can be viewed as a variables
- PyTorch tracks all operations on tensors if you set .requires_grad as True
- After finishing all operations, just call .backward to compute all gradients automatically

Recall the computation graph we used:





Auto-gradient

Create a tensor and set requires_grad=True to track computation with it:

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

Out:

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

Do a tensor operation:

```
y = x + 2
print(y)
```

y was created as a result of an operation, so it has a grad_fn

Out:

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

Auto-gradient

Do more operations on y:

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

Out:

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>)
tensor(27., grad_fn=<MeanBackward0>)
```

.requires_grad_(...) changes an existing Tensor’s requires_grad flag in place. The input flag defaults to False if not given.

```
a = torch.randn(2, 2)
a = ((a * 3) / (a - 1))
print(a.requires_grad)
a.requires_grad_(True)
print(a.requires_grad)
b = (a * a).sum()
print(b.grad_fn)
```

Out:

```
False
True
<SumBackward0 object at 0x7fcb3e741b00>
```

Back propagation with auto-gradient

Let's backpropagate now by calling out.backward()

```
out.backward()
```

Print gradients d(out)/dx

```
print(x.grad)
```

Out:

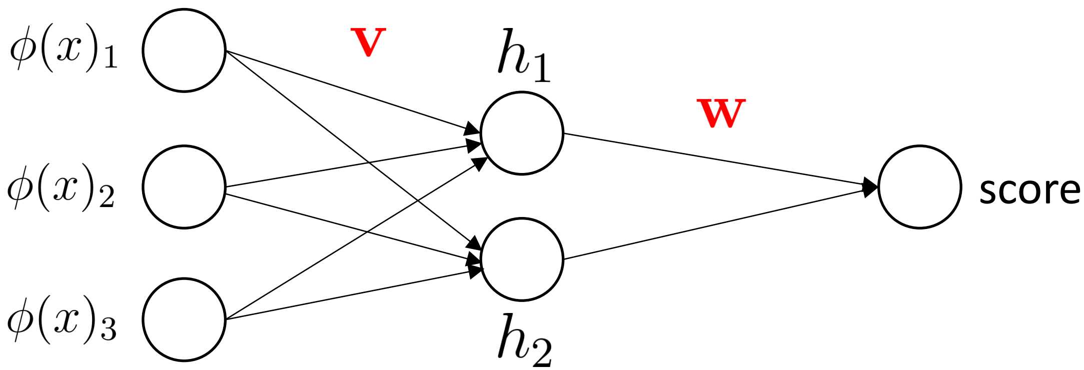
```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

How to verify the output? $o = \frac{1}{4}\sum_i z_i \quad z_i = 3(x_i+2)^2 \quad z_i\big|_{x_i=1} = 27$

$$\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i+2) \qquad \frac{\partial o}{\partial x_i}\Big|_{x_i=1} = \frac{9}{2} = 4.5$$

Demo

Recall the neural network we used:



Intermediate hidden units:

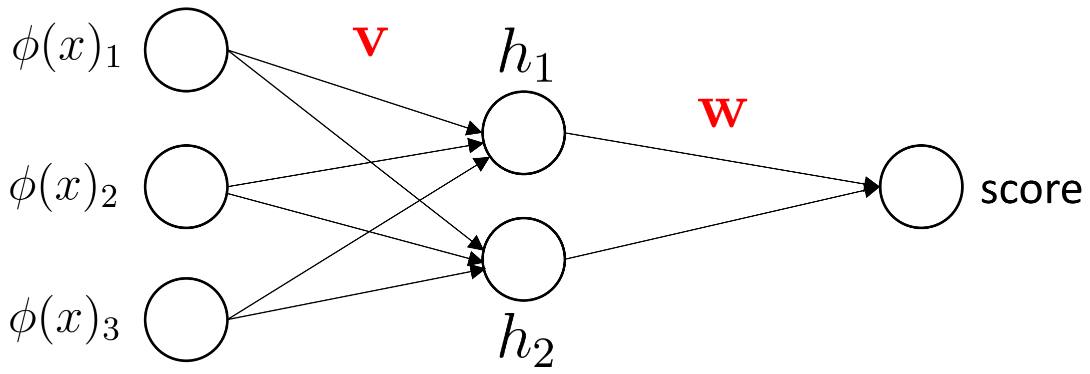
$$h_j = \text{ReLU}(\mathbf{v}_j \cdot \phi(x)) \quad \text{ReLU}(z) = \max(z, 0)$$

Output:

$$\text{score}(\phi(x); \mathbf{v}, \mathbf{w}) = \mathbf{w} \cdot [h_1, h_2]$$

How to compute implement this network with PyTorch?

A mathematical way



$$\frac{\partial}{\partial w_1}\mathrm{score}(\phi(x);\mathbf{v},\mathbf{w}) = \frac{\partial \sum_{j=1}^{2} w_j \mathrm{ReLU}(\mathbf{v}_j \cdot \phi(x))}{\partial w_1} = \mathrm{ReLU}(\mathbf{v}_1 \cdot \phi(x))$$

$$\left.\begin{aligned} &\phi(x) = [1.0, 0.5, 0.2] \\ &\mathbf{v} = [\underbrace{[0.2, 0.0, 0.5]}_{\mathbf{v}_1}, \underbrace{[0.0, 1.0, 0.5]}_{\mathbf{v}_2}] \\ &\mathbf{w} = [0.5, 0.5] \end{aligned}\right\} \Rightarrow \frac{\partial}{\partial w_1}\mathrm{score}(\phi(x);\mathbf{v},\mathbf{w}) = 0.3$$

PyTorch implementation

```
C:\Users\30535\AppData\Local\Programs
score: 0.45
gradient w.r.t. w: 0.30, 0.60
tensor([0.3000, 0.6000])
gradient w.r.t. v:
tensor([[0.5000, 0.2500, 0.1000],
        [0.5000, 0.2500, 0.1000]])
```

[live solution]

Summary

- PyTorch is able to track all computations automatically associated with tensors (variables)
- PyTorch is able to calculate all gradients automatically

Acknowledgement

Reference and thanks to:

- **Sandford University CS221 Course:**

Artificial Intelligence: Principles and Techniques

https://stanford-cs221.github.io/autumn2022/

- **PyTorch Introduction**

https://courses.cs.washington.edu/courses/cse446/19au/section9.html