

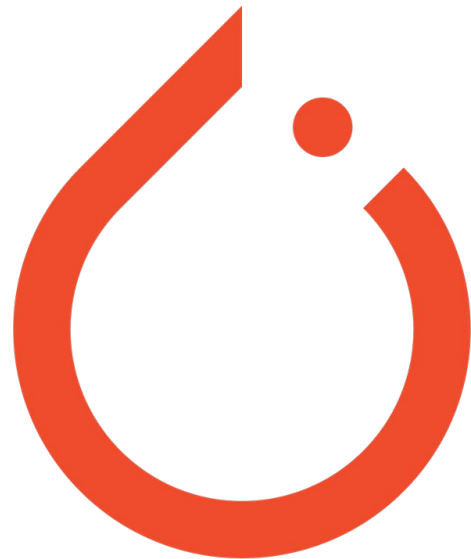


Deep Learning with PyTorch

Advance Machine Learning 2024/2025

Teaching Assistant: Simone Alberto Peirone

Slide credits: Antonio Alliegro





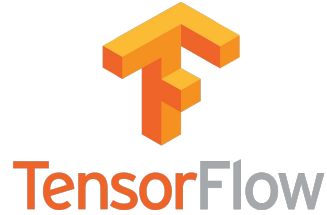
Deep Learning with PyTorch

- **Prerequisites:**
 - Object Oriented Programming
 - Python
 - Numpy
 - Basic understanding of Deep Learning

Deep Learning Frameworks



Caffe 2
(Facebook AI)



(Google, tensorflow.org)



(Facebook AI, pytorch.org)



Python Deep Learning API
(keras.io)



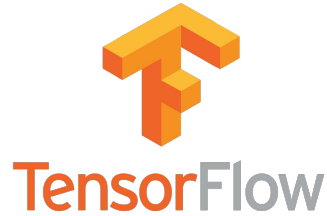
JAX
(Google, <https://github.com/google/jax>)

Several Others..

Deep Learning Frameworks



Caffe 2
(Facebook AI)



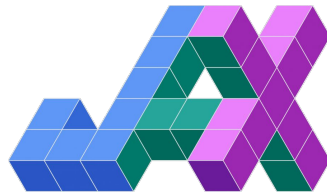
(Google, tensorflow.org)



(Facebook AI, pytorch.org)



Python Deep Learning API
(keras.io)



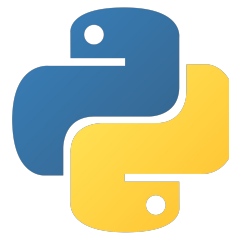
JAX
(Google, <https://github.com/google/jax>)

Several Others..

Our Choice



(Facebook AI, pytorch.org)



(Python as main
programming language)

- Why PyTorch?

Our Choice



PyTorch

+



(Facebook AI, pytorch.org)

(Python as main
programming language)

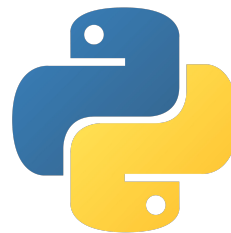
- Why PyTorch?
 - **Pythonic Nature:**
 - Follows standard Python conventions,
 - Python developers should feel more confident with PyTorch than with any other DL framework

Our Choice



PyTorch

+



(Facebook AI, pytorch.org)

(Python as main
programming language)

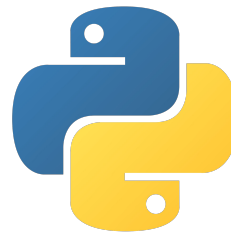
- Why PyTorch?
 - **Pythonic Nature:** ...
 - **Easy to learn:** intuitive syntax and similar to Numpy

Our Choice



PyTorch

+



(Facebook AI, pytorch.org)

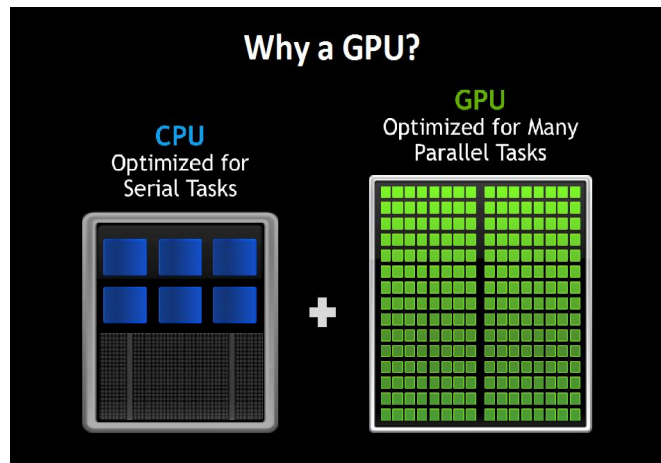
(Python as main
programming language)

- Why PyTorch?
 - **Pythonic Nature:** ...
 - **Easy to learn:** intuitive syntax and similar to Numpy
 - **Strong Community:** Find help on <https://discuss.pytorch.org>

PyTorch

https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#what-is-pytorch

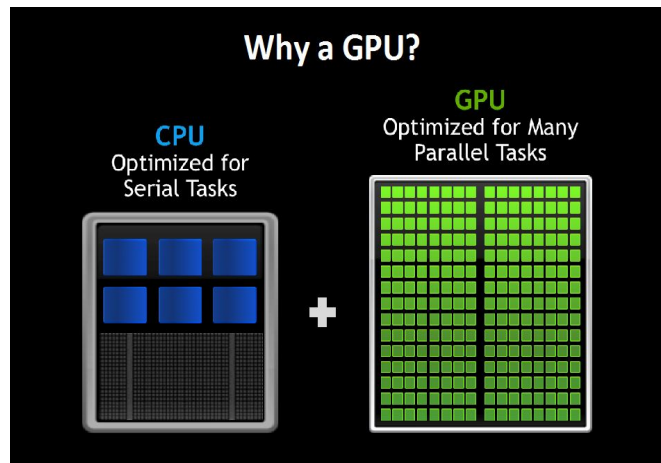
- An Open Source Framework for Deep Learning (and Machine Learning)
- **Deep Learning:**
 - Mathematical computing on **Multidimensional Arrays (aka Tensors)**
 - Highly benefit from **Parallel Computing**



PyTorch

https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#what-is-pytorch

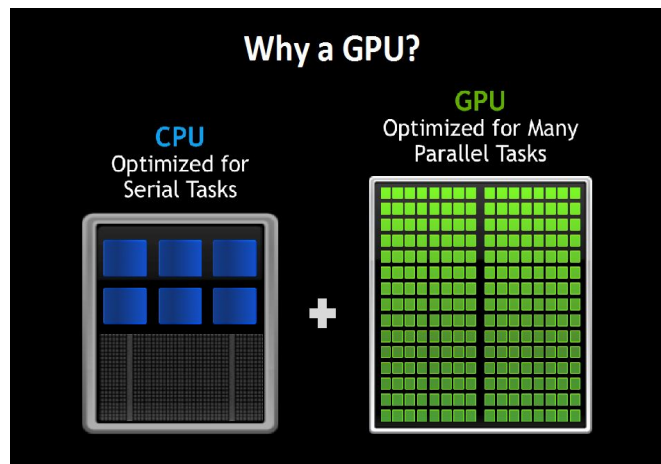
- Exploit Parallel Computing
- CPU: few cores that can handle a few threads at a time - not ideal for DL
- GPU:
 - Increased level of parallelism
 - Hundreds of cores that can handle thousands of threads simultaneously
 - e.g. NVIDIA RTX 3090 has **10496** CUDA cores!



PyTorch

https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#what-is-pytorch

- PyTorch let us easily exploit GPUs' power
- Thus being a 'replacement for Numpy to exploit the parallelism offered by GPUs'
- Offering
 - Strong performance
 - Automatic Differentiation
 - High Flexibility



Tensors

- Tensors are the PyTorch counterpart of Numpy arrays
- Contain *only numerical values*
- Used to encode:
 - **Signal to process** (e.g. images, strings of text, videos, ...)
 - **Internal states and parameter** of neural networks
- **All of PyTorch computation takes place on Tensors**



```
#           R   G   B
img = tensor([[[ 48,  80,  79],
               [175, 104, 207],
               [162,  24, 224],
               [ 97,  27,  28],
               [ 51, 137,  60],
               [124, 214, 249]],
              ...
             ]])
t.size() ==> [256, 256, 3]
t.device ==> gpu:0
t.dtype ==> torch.float32
```

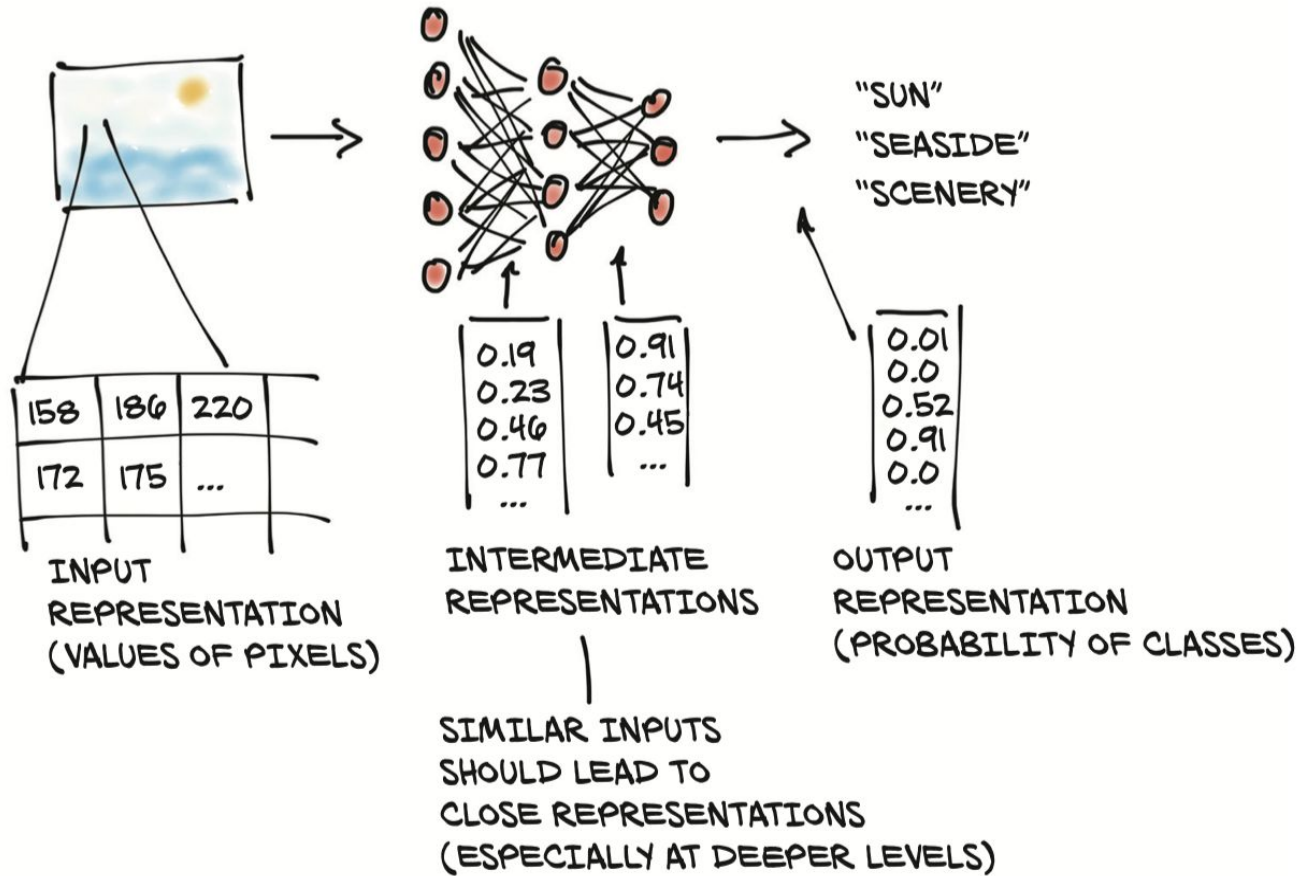


Figure 3.1 A deep neural network learns how to transform an input representation to an output representation. (Note: The numbers of neurons and outputs are not to scale.)



Tensors

- Create a Tensor 't' of size (2,3) from scratch:
 - Zeros init:
`t = torch.zeros(size=(2,3), dtype=torch.float32)`
 - Ones init:
`t = torch.ones(size=(2,3), dtype=torch.float32)`
 - Random init:
`t = torch.rand(size=(2,3), dtype=torch.float32)`
- Properties:
 - `t.size()` - returns its shape - OSS. `size()` is a method of the Tensor class! (not a property)
 - `t.device` - whether it is store on CPU or GPU (+index)
 - `t.dtype` - values type (e.g. `torch.int8`, `torch.float32`, `torch.bool`, ...)

1. Create a tensor from scratch in PyTorch

```
In [10]: torch.zeros(size=(2,3), dtype=torch.float32)
Out[10]:
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

```
In [11]: torch.ones(size=(2,3), dtype=torch.float32)
Out[11]:
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

```
In [12]: torch.rand(size=(2,3), dtype=torch.float32)
Out[12]:
tensor([[0.5818, 0.3530, 0.5550],
        [0.4265, 0.1131, 0.5708]])
```

2. Check tensor 't' properties

```
In [21]: print(t)
tensor([[0.2330, 0.1985, 0.6867],
        [0.3123, 0.6324, 0.1508]])
```

```
In [22]: t.size()
Out[22]: torch.Size([2, 3])
```

```
In [23]: t.device
Out[23]: device(type='cpu')
```

```
In [24]: t.dtype
Out[24]: torch.float32
```

Tensors

- **From NumPy array to Tensor:**
 - use `torch.from_numpy()` static method
 - line 36
- **From Tensor to NumPy array:**
 - use `tensor.numpy()` method
 - line 38
- **Move tensor between CPU and GPU:**
 - Tensor are initialized on CPU by default
 - `tensor.device` to check where it is stored
 - `tensor.cuda()` to move it on GPU:0
 - `tensor.cpu()` to move it on CPU

3. Numpy bridge

```
In [33]: import torch, numpy as np
```

```
In [34]: np_arr = np.random.rand(2, 3)
```

```
In [35]: np_arr.shape
```

```
Out[35]: (2, 3)
```

```
In [36]: t = torch.from_numpy(np_arr)
```

```
In [37]: t.size()
```

```
Out[37]: torch.Size([2, 3])
```

```
In [38]: np_arr_theReturn = t.numpy()
```

```
In [39]: np_arr_theReturn.shape
```

```
Out[39]: (2, 3)
```

```
In [40]: np_arr_theReturn
```

```
Out[40]: array([[0.20823187, 0.55286813, 0.37927967],  
               [0.31437054, 0.21745502, 0.22156234]])
```

4. Move tensor to GPU, you need a CUDA capable device

```
In [18]: print(t, " , device: ", t.device)  
tensor([[0.6417, 0.4857, 0.9736],  
        [0.1095, 0.5623, 0.1343]]) , device: cpu
```

```
In [19]: t = t.cuda() # move to gpu!
```

```
In [20]: print(t, " , device: ", t.device)  
tensor([[0.6417, 0.4857, 0.9736],  
        [0.1095, 0.5623, 0.1343]], device='cuda:0') , device: cuda:0
```

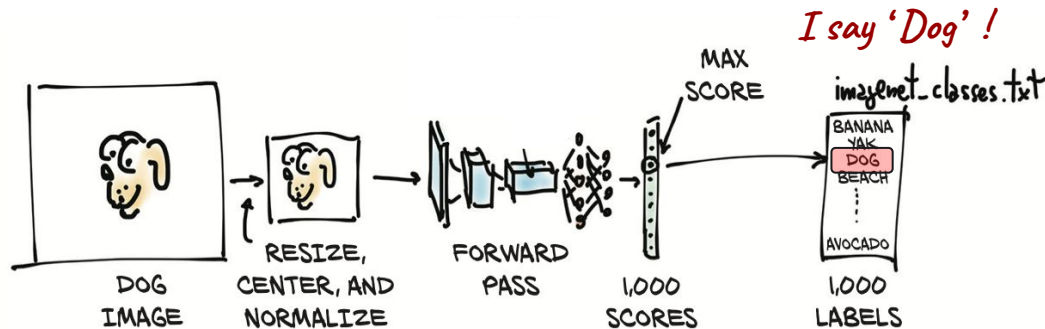


Neural networks and Backpropagation

- Define the neural network architecture
- While training:

Neural networks and Backpropagation

- Define the neural network architecture
- While training:
 1. **Forward pass:** feed input data to the network to obtain the Network Prediction





Neural networks and Backpropagation

- Define the neural network architecture
- While training:
 1. **Forward pass:** feed input data to the network to obtain the Network Prediction
 2. **Compute Loss:** compare Network Prediction with Ground Truth



Neural networks and Backpropagation

- Define the neural network architecture
- While training:
 1. **Forward pass:** feed input data to the network to obtain the Network Prediction
 2. **Compute Loss:** compare Network Prediction with Ground Truth
 3. **Backward pass:** Chain Rule to compute the gradients of Network parameters w.r.t to the loss function (**PyTorch Autograd**)



Neural networks and Backpropagation

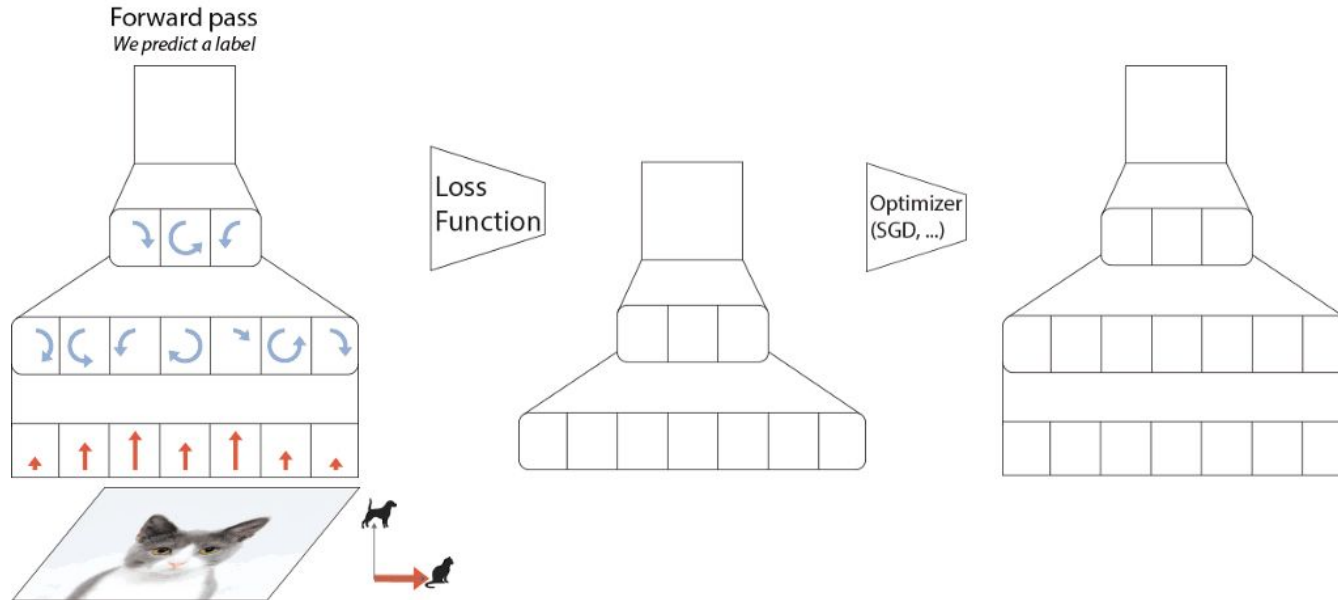
- Define the neural network architecture
- While training:
 1. **Forward pass:** feed input data to the network to obtain the Network Prediction
 2. **Compute Loss:** compare Network Prediction with Ground Truth
 3. **Backward pass:** Chain Rule to compute the gradients of Network parameters w.r.t to the loss function (**PyTorch Autograd**)
 4. **Update** the network parameters (**PyTorch Optimizer**)



Neural networks and Backpropagation

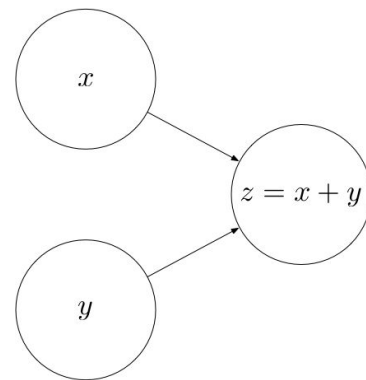
- Define the neural network architecture
- While training:
 1. **Forward pass:** feed input data to the network to obtain the Network Prediction
 2. **Compute Loss:** compare Network Prediction with Ground Truth
 3. **Backward pass:** Chain Rule to compute the gradients of Network parameters w.r.t to the loss function (**PyTorch Autograd**)
 4. **Update** the network parameters (**PyTorch Optimizer**)
- *Repeat from 1... until convergence*

Neural networks and Backpropagation



Computational Graph

- The **Computational Graph** is a directed graph keeping track of all Operations performed on Variables
- To support 'Forward Pass' and 'Backward Pass'
- **Nodes** represent:
 - **Variables:** can feed their output into operations
 - **Operations:** can feed their output into other operations



PyTorch Autograd

- Do we need to build a Computational Graph by ourselves?
- No, Autograd takes care of it!
- Every operations applied to Variables is tracked by PyTorch through the Autograd tape
- Thus providing **automatic differentiation**

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))
```





PyTorch - Basic Components

- **Network Architecture**
 - Define a subclass of 'torch.nn.Module'
- **Dataset**
 - Define a subclass of 'torch.utils.data.Dataset'
- **Loss Function + Optimizer**
 - Network Prediction penalization + Update Network parameters
- **Training Loop**
 - Simple python script: sort of main function interconnecting all components



Network

- All neural networks (and layers) in PyTorch are a subclass of the torch Module class
 - **torch.nn.Module**: base class for all neural network modules
 - Modules can also contain other Modules, allowing to nest them in a tree structure
 - <https://pytorch.org/docs/stable/nn.html#torch.nn.Module>
- Pre-defined models for addressing different tasks
 - <https://pytorch.org/vision/stable/models.html>



Network from Scratch

- Define a simple MLP
 - Two Fully Connected layers spaced out by ReLU activation function

```
import torch.nn as nn

# Fully connected neural network with one hidden layer
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, 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
```



Network from Scratch

- Define a simple MLP
 - Two Fully Connected layers spaced out by ReLU activation function

initialization:
instantiate linear
layers and relu
(encapsulated)

```
import torch.nn as nn

# Fully connected neural network with one hidden layer
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, 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
```

Network from Scratch

- Define a simple MLP
 - Two Fully Connected layers spaced out by ReLU activation function

initialization:
instantiate linear
layers and relu
(encapsulated)

```
import torch.nn as nn

# Fully connected neural network with one hidden layer
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, 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
```

forward function:
describe the flow of input data (x)
through the network layers

Network from Scratch

- Instantiate the neural network object

Instantiate
network object
⇒ model

```
input_size, hidden_size, num_classes = 784, 500, 10
model = NeuralNet(input_size=input_size, hidden_size=hidden_size, num_classes=num_classes)
print('Our model:')
print(model)
```

```
Our model:
NeuralNet(
  (fc1): Linear(in_features=784, out_features=500, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=500, out_features=10, bias=True)
)
```

```
# FORWARD
forward_res = model(img)
```

Network from Scratch

- Instantiate the neural network object
- Invoke forward on data

Instantiate
network object
⇒ model

```
input_size, hidden_size, num_classes = 784, 500, 10
model = NeuralNet(input_size=input_size, hidden_size=hidden_size, num_classes=num_classes)
print('Our model:')
print(model)
```

```
Our model:
NeuralNet(
  (fc1): Linear(in_features=784, out_features=500, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=500, out_features=10, bias=True)
)
```

```
# FORWARD
forward_res = model(img)
```

calls network forward function
on 'img' ⇒ logits



Data Reading

- Define Preprocessing
 - <https://pytorch.org/docs/stable/torchvision/transforms.html>
- Create a Dataset class
 - <https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset>
- Choose a Sampling Strategy
 - <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>



Dataset Class

```
In [8]: import torch
import pandas as pd
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms

class DatasetMNIST(Dataset):

    def __init__(self, file_path, transform=None):
        self.data = pd.read_csv(file_path)
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        # load image as ndarray type (Height * Width * Channels)
        # be carefull for converting dtype to np.uint8 [Unsigned integer (0 to 255)]
        # in this example, i don't use ToTensor() method of torchvision.transforms
        # so you can convert numpy ndarray shape to tensor in PyTorch (H, W, C) --> (C, H, W)
        image = self.data.iloc[index, 1:].values.astype(np.uint8).reshape((1, 28, 28))
        label = self.data.iloc[index, 0]

        if self.transform is not None:
            image = self.transform(image)

        return image, label
```



Dataset Class

- **torch.data.Dataset:** base class for all datasets
- **Basic Methods to override:**
 - **init():** initial processing, loading data in memory, ...
 - **len():** returns the total number of samples in the dataset
 - **getitem(idx):** given an index return a data sample from the dataset
- **Transforms:** pre-processing or data-augmentation to be applied on each sample

```
In [9]: transform = transforms.Compose([
        transforms.ToPILImage(), # because the input dtype is numpy.ndarray
        transforms.RandomHorizontalFlip(0.5), # because this method is used for PIL Image dtype
        transforms.ToTensor(), # because input dtype is PIL Image
    ])

train_dataset = DatasetMNIST(file_path='../input/train.csv', transform=transform)
```

Example: Instantiating an object Dataset



DataLoader

- PyTorch provides an efficient way to iterate a Dataset through **DataLoader** (*torch.utils.data.DataLoader*)
- Its constructor takes as input an object of type **Dataset**
- Provides:
 - **Data Batching**: we want to forward to our network batches of data (e.g. 32 images at a time)
 - **Data Shuffling**
 - **Parallel Data Loading**: multiple threads/workers loading data in parallel



DataLoader

- Constructor takes as input an object of type Dataset

```
train_loader = torch.utils.data.DataLoader(  
    torchvision.datasets.MNIST('/files/', train=True, download=True,  
                               transform=torchvision.transforms.Compose([  
                                   torchvision.transforms.ToTensor(),  
                                   torchvision.transforms.Normalize(  
                                       (0.1307,), (0.3081,))  
                               ])),  
    batch_size=batch_size_train, shuffle=True)
```

Example: Instantiating a DataLoader object



Loss

- In PyTorch there are some predefined Loss Function that we can use
- Examples are **torch.nn.CrossEntropyLoss** (classification) and **torch.nn.MSELoss** (regression)

Instantiate
loss

```
In [1]: import torch

In [2]: import torch.nn

In [3]: loss_fn = torch.nn.CrossEntropyLoss()

In [4]: # define some stub tensors for network pred and ground truth

In [5]: pred = torch.rand(3, 5, requires_grad=True)

In [6]: ground_truth = torch.empty(3, dtype=torch.long).random_(5)

In [7]: # compute loss

In [8]: loss = loss_fn(pred, ground_truth)

In [9]: # now backward

In [10]: loss.backward()
```

```
In [12]: print("pred: ", pred)
pred: tensor([[0.7878, 0.1673, 0.6948, 0.4985, 0.4380],
              [0.8667, 0.6241, 0.0718, 0.2107, 0.5227],
              [0.0934, 0.7967, 0.5874, 0.6695, 0.9948]], requires_grad=True)

In [13]: print(pred.shape)
torch.Size([3, 5])

In [14]: print("ground_truth: ", ground_truth)
ground_truth: tensor([0, 3, 4])

In [15]: print(ground_truth.shape)
torch.Size([3])
```

CrossEntropyLoss example - *CrossEntropyLoss* takes as input the network logits since it encapsulates a softmax op. inside!

Loss

- In PyTorch there are some predefined Loss Function that we can use
- Examples are **torch.nn.CrossEntropyLoss** (classification) and **torch.nn.MSELoss** (regression)

```
In [1]: import torch
```

```
In [2]: import torch.nn
```

```
In [3]: loss_fn = torch.nn.CrossEntropyLoss()
```

```
In [4]: # define some stub tensors for network pred and ground truth
```

```
In [5]: pred = torch.rand(3, 5, requires_grad=True)
```

```
In [6]: ground_truth = torch.empty(3, dtype=torch.long).random_(5)
```

```
In [7]: # compute loss
```

```
In [8]: loss = loss_fn(pred, ground_truth)
```

```
In [9]: # now backward
```

```
In [10]: loss.backward()
```

```
In [12]: print("pred: ", pred)
pred:  tensor([[0.7878, 0.1673, 0.6948, 0.4985, 0.4380],
              [0.8667, 0.6241, 0.0718, 0.2107, 0.5227],
              [0.0934, 0.7967, 0.5874, 0.6695, 0.9948]], requires_grad=True)
```

```
In [13]: print(pred.shape)
torch.Size([3, 5])
```

```
In [14]: print("ground_truth: ", ground_truth)
ground_truth:  tensor([0, 3, 4])
```

```
In [15]: print(ground_truth.shape)
torch.Size([3])
```

CrossEntropyLoss example - *CrossEntropyLoss* takes as input the network logits since it encapsulates a softmax op. inside!

Loss

- In PyTorch there are some predefined Loss Function that we can use
- Examples are **torch.nn.CrossEntropyLoss** (classification) and **torch.nn.MSELoss** (regression)

```
In [1]: import torch
```

```
In [2]: import torch.nn
```

```
In [3]: loss_fn = torch.nn.CrossEntropyLoss()
```

```
In [4]: # define some stub tensors for network pred and ground truth
```

```
In [5]: pred = torch.rand(3, 5, requires_grad=True)
```

```
In [6]: ground_truth = torch.empty(3, dtype=torch.long).random_(5)
```

```
In [7]: # compute loss
```

```
In [8]: loss = loss_fn(pred, ground_truth)
```

```
In [9]: # now backward
```

```
In [10]: loss.backward()
```

```
In [12]: print("pred: ", pred)
pred: tensor([[0.7878, 0.1673, 0.6948, 0.4985, 0.4380],
             [0.8667, 0.6241, 0.0718, 0.2107, 0.5227],
             [0.0934, 0.7967, 0.5874, 0.6695, 0.9948]]) requires_grad=True
```

```
In [13]: print(pred.shape)
torch.Size([3, 5])
```

```
In [14]: print("ground_truth: ", ground_truth)
ground_truth: tensor([0, 3, 4])
```

```
In [15]: print(ground_truth.shape)
torch.Size([3])
```



Instantiate
loss

Compute
loss

Backward

CrossEntropyLoss example - *CrossEntropyLoss* takes as input the network logits since it encapsulates a softmax op. inside!



Optimizer

- Updates network parameters depending on the gradients computed at the backward step (*loss.backward()* in previous slide)
- Optimizer constructor takes as inputs at least:
 - **parameters** : parameters to optimize (e.g. a neural network)
 - **lr** : learning rate to use in the update rule
- Basic method:
 - **step()** : update model parameters - **do this only after the loss.backward()**

Instantiate
Optimizer

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

doc: <https://pytorch.org/docs/stable/optim.html>



Optimization Step

```
for input, target in dataset:  
    optimizer.zero_grad()  
    output = model(input)  
    loss = loss_fn(output, target)  
    loss.backward()  
    optimizer.step()
```



Training loop example

- Just code..

```

1. criterion = nn.CrossEntropyLoss()
2. optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
3.
4. for epoch in range(100): # loop over the dataset multiple times
5.     tot_loss, tot_samples = 0.0, 0
6.     for i, data in enumerate(train_dataloader):
7.         # Step 1: Retrieving a batch of input from the dataloader
8.         inputs, labels = data
9.         tot_samples += inputs.size(0)
10.
11.         # Step 2: Zeroing the parameter gradients - always do this before doing loss.backward()!!!
12.         optimizer.zero_grad()
13.         # Step 3: forward (get network prediction)
14.         outputs = model(inputs)
15.
16.         # Step 4: compute loss
17.         loss = criterion(outputs, labels)
18.         # Step 5: Compute gradients for each of the model learnable parameters
19.         loss.backward()
20.         # Step 4: Update model parameters according to the gradients
21.         optimizer.step()
22.
23.         # logging, accumulating loss value for current epoch...
24.         tot_loss += (loss.data[0] * inputs.size(0))
25.     # End epoch here
26.     print("Epoch %d loss is: %.6f" % (epoch, (tot_loss * 1.0 / float(tot_samples))))
27. # End Training here

```



It's your turn!

<https://colab.research.google.com/drive/1jwBxRTNnu0oxkx6xGQ44UHxJWVYw3jeX?usp=sharing>