Intelligence Artificielle Avancée

Outline

- Introduction
- How PyTorch Works?
  - Data types
  - Functions
  - Differentiation in Autograd
  - PyTorch with GPU’s
  - Neural Networks
- Learning PyTorch with Examples

Introduction

- PyTorch is an open source machine learning framework that accelerates the path from research to production
- Developed primarily by Facebook AI and introduced in 2016

Introduction

- The framework combines the efficient GPU-accelerated backend libraries from Torch with Python frontend
    - Focuses on rapid prototyping, readable code, and support for the variety of deep learning models
- It allows deep learning models to be expressed in the Python programming language

Important Properties of PyTorch

- Python support
  - PyTorch is based on Python, it can be used with popular libraries and packages such as NumPy, SciPy, Numba and Cython
  - Offers developers an easy-to-learn, simple-to-code structure that's based on Python
  - Enables easy debugging with popular Python tools
- TorchScript
  - Production environment of PyTorch that enables users to transition between modes
  - TorchScript optimizes functionality, speed, ease of use and flexibility

Important Properties of PyTorch

- Offers scalability and is well-supported on major cloud platforms
- It supports CPU, GPU, and parallel processing, as well as distributed training
- The PyTorch Hub is a repository of pre-trained models that can be invoked, in some cases with just a single line of code
  - It has a large collection of tools and libraries in areas ranging from computer vision to reinforcement learning

Installing PYTORCH

OS	**Linux**	OSX		
Package Manager	conda	**pip**	Source	
Python	2.7	3.5	**3.6**	
CUDA	8	9.0	**9.1**	None

Run this command:

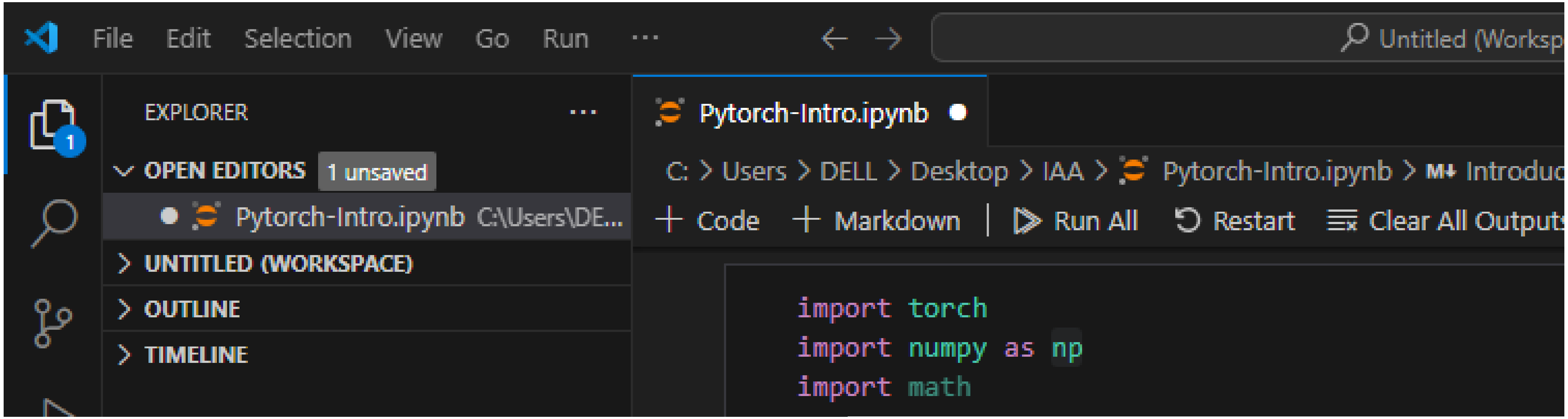
```
pip3 install http://download.pytorch.org/whl/cu91/torch-0.3.1-cp36-cp36m-linux_x86_64.whl
pip3 install torchvision
```

How to Download

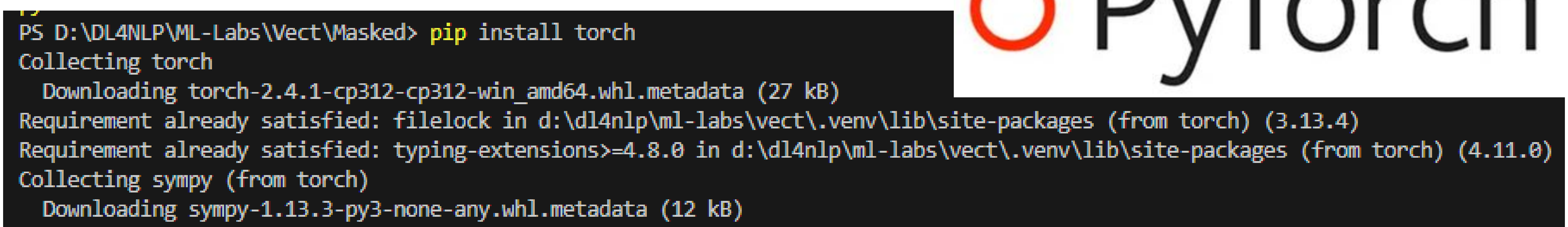
- For installation, first, you have to choose your preference and then run the install command
- From pytorch.org/get-started/locally you can install by following instructions

PyTorch Build	Stable (2.1.2)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.8	CUDA 12.1	ROCm 5.6	CPU
Run this Command:	`pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118`			

Installing Visual Studio Code (VSCode) Jupyter Notebook environment.



pip install torch



Introduction to PyTorch

Pytorch is a popular neural net framework with the following features:

- Automatic differentation
- Compiling computation graphs
- Libraries of algorithms and network primitives. Provides a high-level abstractions for working with neural networks.
- Support for graphics processing units (GPU)

Introduction to PyTorch

Pytorch is a popular neural net framework with the following features:

- Automatic differentiation
- Compiling computation graphs
- Libraries of algorithms and network primitives. Provides a high-level abstractions for working with neural networks.
- Support for graphics processing units (GPU)

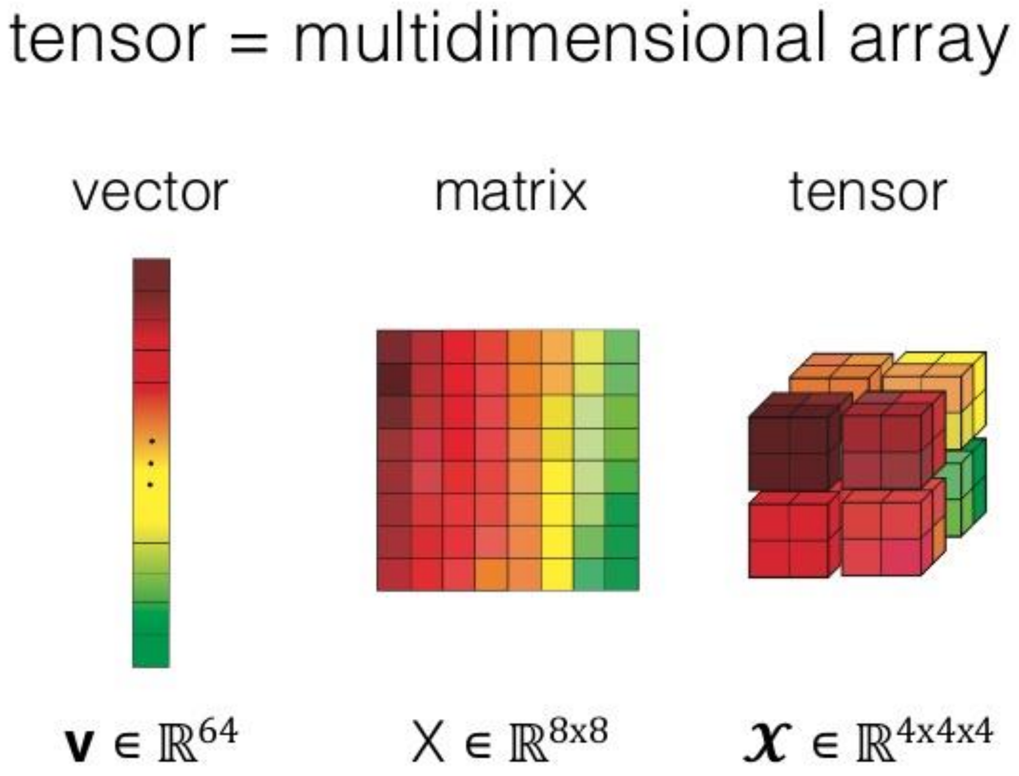
In this lesson, we will learn the basics of PyTorch. We will cover the following topics:

1. Tensors
2. Automatic differentiation
3. Building a simple neural network
4. PyTorch Datasets and DataLoaders
5. Visualizing examples from the FashionMNIST Dataset
6. Training on CPU
7. Training on GPU
8. Using pre-trained weights

How PyTorch Works? - Tensors

- Fundamentally, it's a library for programming with tensors
- Tensors are the fundamental building blocks of neural networks in PyTorch
- Tensors are a specialized data structure that are very similar to arrays and matrices
  - Which are basically just multidimensional arrays!
  - In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters
- Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs

Tensors


tensor = multidimensional array
vector
matrix
tensor
$\mathbf{v} \in \mathbb{R}^{64}$
$X \in \mathbb{R}^{8x8}$
$\mathcal{X} \in \mathbb{R}^{4x4x4}$

Tensors

- Tensors can be initialized in various ways
- The simplest way to create a tensor is with the **torch.empty()** call:

```
x = torch.empty(3, 4)
```

- Created tensor x is 2-dimensional, with 3 rows and 4 columns
- By default, PyTorch tensors are 32-bit floating numbers
- **torch.empty()** allocates memory for the tensor, but does not initialize it with any values

Introduction to PyTorch

1. Tensors

Tensors are a specialized data structure very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.

Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators.

Initializing a Tensor

```python
import torch
import numpy as np
import math

# Create a tensor directly from data
x = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
print("x:", x)
# Create a tensor of zeros
y = torch.zeros(2, 2)
print("y:", y)
# Create a tensor of ones
z = torch.ones(2, 2)
print("z:", z)
# Create a random tensor
w = torch.rand(2, 2)
print("w:", w)
# Create a tensor from a NumPy array
np_array = np.array([1,2,3])
x_np = torch.from_numpy(np_array)
print("x_np:", x_np)
```

```
x: tensor([[1., 2.],
        [3., 4.]])
y: tensor([[0., 0.],
        [0., 0.]])
z: tensor([[1., 1.],
        [1., 1.]])
w: tensor([[0.4927, 0.0661],
        [0.1687, 0.8788]])
x_np: tensor([1, 2, 3], dtype=torch.int32)
```

Introduction to PyTorch

Attributes of a tensor

```
tensor = torch.tensor([[1, 2, 3], [3, 4, 5]])

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

[9] ✓ 0.0s

```
...   Shape of tensor: torch.Size([2, 3])
      Datatype of tensor: torch.int64
      Device tensor is stored on: cpu
```

Introduction to PyTorch

Operations on Tensors

```python
# Move the tensor to GPU if available
if torch.cuda.is_available():
    tensor = tensor.to("cuda")

# Standard numpy-like indexing and slicing
tensor = torch.tensor([[1,2,3], [3,4,5]])
print("First row: ", tensor[0])
print("First column: ", tensor[:,0])
```

[10] ✓ 0.5s

```
First row:  tensor([1, 2, 3])
First column:  tensor([1, 3])
```

Introduction to PyTorch

Matrix multiplication

```python
# Matrix multiplication
tensor = torch.ones(3, 3)
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)
print("y1: ", y1)
print("y2: ", y2)
```

[11] ✓ 0.2s

```
...   y1:  tensor([[3., 3., 3.],
              [3., 3., 3.],
              [3., 3., 3.]])
      y2:  tensor([[3., 3., 3.],
              [3., 3., 3.],
              [3., 3., 3.]])
```

Introduction to PyTorch

Element wise product

```
# Element wise product
z1 = tensor * tensor
z2 = tensor.mul(tensor)
print("z1: ", z1)
print("z2: ", z2)
```

[12] ✓ 0.3s

```
z1:  tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
z2:  tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

Introduction to PyTorch

common functions

```python
# common functions
a = torch.rand(2, 4) * 2 - 1
print('Common functions:')
print(torch.abs(a))
print(torch.ceil(a))
print(torch.floor(a))
print(torch.clamp(a, -0.5, 0.5))

# Reshape
a = torch.arange(4.)
a_reshaped = torch.reshape(a, (2, 2))
b = torch.tensor([[0, 1], [2, 3]])
b_reshaped =torch.reshape(b, (-1,))
print("a_reshaped", a_reshaped)
print("b_reshaped", b_reshaped)
```

[13] ✓ 0.5s

```
Common functions:
tensor([[0.3726, 0.4314, 0.8540, 0.6978],
        [0.0827, 0.1988, 0.1837, 0.3726]])
tensor([[-0., 1., -0., -0.],
        [1., 1., -0., 1.]])
tensor([[-1.,  0., -1., -1.],
        [ 0.,  0., -1.,  0.]])
tensor([[-0.3726,  0.4314, -0.5000, -0.5000],
        [ 0.0827,  0.1988, -0.1837,  0.3726]])
a_reshaped tensor([[0., 1.],
        [2., 3.]])
b_reshaped tensor([0, 1, 2, 3])
```

Introduction to PyTorch

Tensor Broadcasting

```
x1 = torch.tensor([[1, 2, 3], [3, 4, 5]])
x2 = torch.tensor([2,2,2])
doubled = x1 * x2

print(doubled)
```

[14] ✓ 0.3s

```
... tensor([[ 2,  4,  6],
        [ 6,  8, 10]])
```

Introduction to PyTorch

2. Automatic Differentation

- Instead of computing backpropagation manually, an autodiff system performs backprop in a completely mechanical way.
- An autodiff system will convert the program into a sequence of primitive operations which have specified routines for computing derivatives.

Distinction of the concepts

- **Backpropagation**: the mathematical algorithm we use to compute the gradient.
- **Automatic differentiation (AutoDiff)**: any software that implements backpropagation.
- Examples: Autograd, TensorFlow, PyTorch, Jax, etc.
- **Reverse Mode AD**: A method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards

Introduction to PyTorch

2.1 Autograd

- Autograd is a Python package for automatic differentiation.

From the Autograd Github repository:

- Autograd can automatically differentiate native Python and Numpy code.
- It can handle a large subset of Python's features, including loops, conditional statements (if/else), recursion and closures.
- It can also compute higher-order derivatives.
- It uses reverse-mode differentiation (a.k.a. backpropagation) so it can efficiently take gradients of scalar-valued functions with respect to array-valued arguments.

```
import autograd.numpy as jnp  # Import thinly-wrapped numpy
from autograd import grad  # Basicallly the only autograd function you need
```

[16] ✓ 0.1s

Tensors

- More often we'll want to initialize our tensor with some value
  - Common cases are:
    - All zeros
    - All ones
    - Random values
- torch module provides methods for all these!

```
zeros = torch.zeros(2, 3)
```

A tensor full of zeros

```
ones = torch.ones(2, 3)
```

A tensor full of ones

```
torch.manual_seed(1729)
random = torch.rand(2, 3)
```

A tensor with random values between 0 and 1

Tensors

- While initializing tensors, such as a model’s learning weights, random values are common
  - But we need reproducibility of our results
  - Manually setting your random number generator (**torch.manual_seed()**) is the way to do this

Tensors

- Most of the time, when we are performing operations on more than one tensor, we need to have them of the same **shape**
  - Having the same number of dimensions, same number of cells in each dimension
- To initialize a tensor with the same shape as another tensor, we are using **torch.*_like()** methods:
  - torch.empty_like(another_tensor)
  - torch.zeros_like(another_tensor)
  - torch.ones_like(another_tensor)
  - torch.rand_like(another_tensor)

Tensors

- We can also specify the data of the tensor directly from a Pytorch collection:
- **torch.tensor()** is the most straightforward way to create a tensor if we already have data
- **torch.tensor()** creates a copy of the data
- Most of the time, our data starts out in NumPy arrays or pandas DataFrames
- We have to convert these data types to tensors using **torch.tensor()**
- **torch.tensor()** method takes two arguments: numerical data (NumPy array, Python list, or Python numeric variable) and desired data type (the dtype parameter)

Tensors

- We can also get the same tensor in our specified data type using methods such as float(), long() etc.
  - We can also use tensor.FloatTensor, tensor.LongTensor, tensor.Tensor classes to instantiate a tensor of particular type
- Or using **.to()**
- Available data types include:
  - torch.bool
  - torch.int8
  - torch.uint16
  - torch.float
  - torch.double

Data type	dtype
16-bit floating point [1]	torch.float16 or torch.half
32-bit floating point	torch.float32 or torch.float
64-bit floating point	torch.float64 or torch.double
64-bit complex	torch.complex64 or torch.cfloat
128-bit complex	torch.complex128 or torch.cdouble
8-bit integer (unsigned)	torch.uint8
8-bit integer (signed)	torch.int8
16-bit integer (signed)	torch.int16 or torch.short
32-bit integer (signed)	torch.int32 or torch.int
64-bit integer (signed)	torch.int64 or torch.long
Boolean	torch.bool

Tensors

- torch.arange(end): Returns a 1-D tensor with elements ranging from 0 to end-1
  - We can use the optional start and step parameters to create tensors with different ranges

```
x = torch.arange(25).view((5, 5))
```

```
tensor([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24]])
```

Attributes of a Tensor

- Shape
- Datatype
  - Float32, Float64, Integer, Boolean
- Device
  - GPU/CPU

Properties of Tensor

- We can get the size of a particular dimension with the **size()** method
  - x.size(0) get's the size of 0th dimension
- Change the shape of a tensor with the **view()** method
  - x_view = x.view(3, 2) (x_view shares the same memory as x, so changing one changes the other)
- We can also use **torch.reshape()** method for a similar purpose
  - x_reshaped = torch.reshape(x, (2, 3))

Properties of Tensor

- We can use **torch.unsqueeze(x, dim)** function to add a dimension of size 1 to the provided dim
- We can also use the corresponding use **torch.squeeze(x)**, which removes the dimensions of size 1
- If we want to get the total number of elements in a tensor, we can use the **numel()** method