

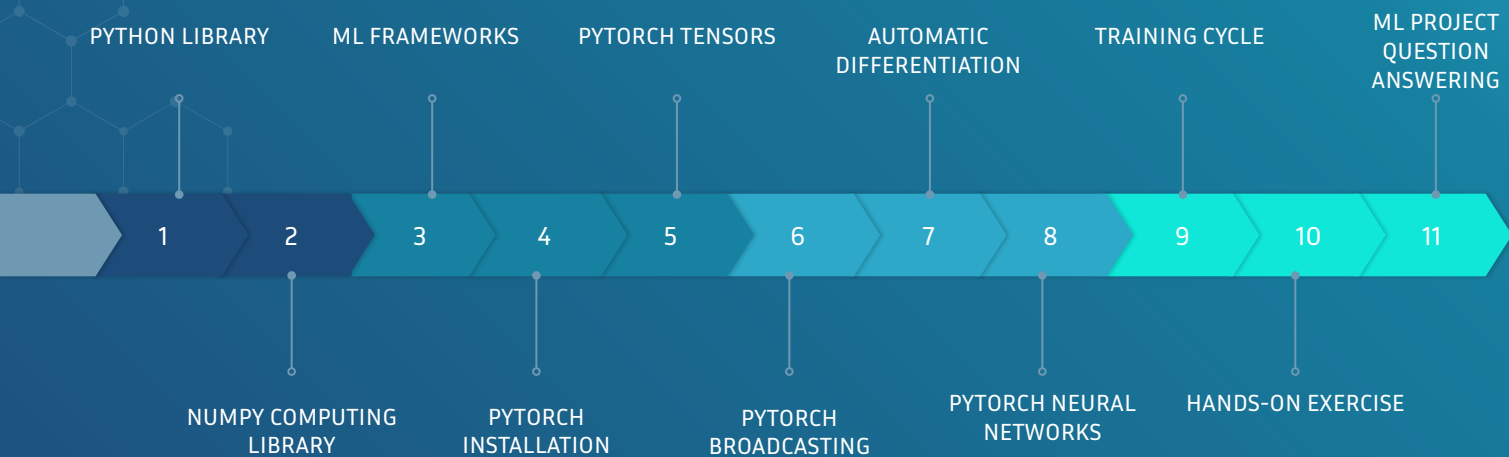
MACHINE LEARNING TOOLS AND FRAMEWORKS



LORENZO SIMONE
TEACHING ASSISTANT

OFFICE 382
E-mail lorenzo.simone@di.unipi.it

INDEX



○ INTRODUCTION

○ LIBRARIES

○ PYTORCH FRAMEWORK

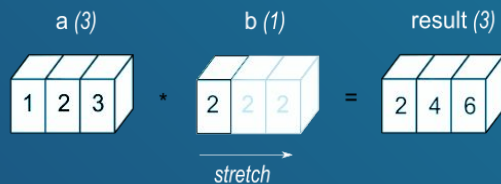
● ML PROJECT INSIGHTS



BACKGROUND ML AND PROGRAMMING KNOWLEDGE ASSESSMENT


NUMPY LIVE CODING INTRODUCTION

- `numpy.array()` properties and Python differences
- Numerical operations, slicing and broadcasting
- Linear algebra
- Random utilities and data visualization with matplotlib
- Numpy and Pytorch interaction



ML FRAMEWORKS

 PyTorch

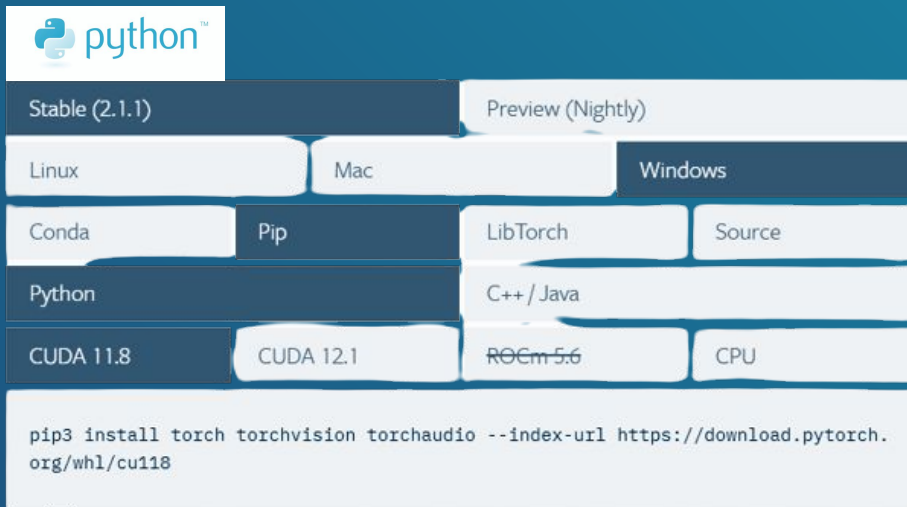

TensorFlow

EASE OF USE	DYNAMIC GRAPH, RESEARCH PURPOSE, IMPERATIVE	EAGER EXECUTION 2.0, HISTORICALLY USED FOR LARGE-SCALE DOMAINS
FLEXIBILITY	EASIER TO DESIGN CUSTOM ARCHITECTURES AND COMPLEX MODEL	FAST PROTOTYPIZATION OF KNOWN ARCHITECTURES AND DEFAULT TRAINING CYCLES
POPULARITY	RAPID POPULARITY GAIN ESPECIALLY IN ACADEMIC FIELDS AND RESEARCH MODELS.	LONG-STANDING REPUTATION IN INDUSTRY DOMAINS
DEPLOYMENT AND PRODUCTION	TORCH SCRIPT & TORCH SERVE	TENSORFLOW SERVING AND TF LITE

PYTORCH INSTALLATION

Installation details

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



The screenshot shows the PyTorch installation guide interface. At the top is the Python logo. Below it are two tabs: 'Stable (2.1.1)' and 'Preview (Nightly)'. The 'Stable (2.1.1)' tab is selected. Underneath are three tabs for operating systems: 'Linux', 'Mac', and 'Windows'. The 'Windows' tab is selected. Below these are four tabs for installation methods: 'Conda', 'Pip', 'LibTorch', and 'Source'. The 'Pip' tab is selected. Below these are two tabs for language/frameworks: 'Python' and 'C++ / Java'. The 'Python' tab is selected. Below these are four tabs for hardware acceleration: 'CUDA 11.8', 'CUDA 12.1', 'ROCm 5.6', and 'CPU'. The 'CUDA 11.8' tab is selected. At the bottom, there is a code block containing the command to install PyTorch using pip3.

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
```

TENSORS AS MATH OBJECTS



SCALAR



VECTOR



MATRIX



TENSOR



RANK 4 TENSOR

TENSOR INITIALIZATION



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

#Zero Matrix
zeros = torch.zeros(2, 3)

#Ones Matrix
ones = torch.ones(2, 3)

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


TENSOR INITIALIZATION

```
● ● ●  
  
if torch.cuda.is_available():  
    gpu_rand = torch.rand(2, 2, device='cuda')  
    print(gpu_rand)  
else:  
    print('Sorry, CPU only.')
```

After confirming the presence of one or more GPUs, the next step is to ensure that our data is accessible to the GPU. While the CPU processes data in the computer's RAM, the GPU has its own dedicated memory. To perform computations on a specific device, it is essential to transfer all the required data to the memory accessible by that device.

Default allocation CPU

TENSOR DATA TYPES

```
torch.bool  
torch.int8  
torch.uint8  
torch.int16  
torch.int32  
torch.int64  
torch.half  
torch.float  
torch.double  
torch.bfloat
```

BROADCASTING AND MATH

```
ones = torch.zeros(2, 2) + 1
ones_numpy = torch.from_numpy(np.ones(2,2))
twos = torch.ones(2, 2) * 2
threes = (torch.ones(2, 2) * 7 - 1) / 2
fours = twos ** 2
sqrt2s = twos ** 0.5

powers2 = twos ** torch.tensor([[1, 2], [3, 4]])
#tensor([[ 2.,  4.], [ 8., 16.]])

matrix = torch.randint(0,10, (2,4))
double_vec = torch.ones(4,1)*2

double_vec*matrix
RuntimeError: The size of tensor a (4) must match the size of tensor b (2)
at non-singleton dimension 0

double_vec.T*y
tensor([[2, 7, 2, 8],
        [9, 0, 6, 6]])
tensor([[ 4., 14.,  4., 16.],
        [18.,  0., 12., 12.]])
```

Broadcasting in PyTorch allows to perform operations on tensors with different shapes.

It automatically adjusts dimensions, making them compatible for element-wise operations.

An error occurs when trying to multiply a matrix by a vector with incompatible dimensions, but broadcasting succeeds after transposing the vector in the second attempt.

BROADCASTING AND MATH

```
ones = torch.zeros(2, 2) + 1
ones_numpy = torch.from_numpy(np.ones(2,2))
twos = torch.ones(2, 2) * 2
threes = (torch.ones(2, 2) * 7 - 1) / 2
fours = twos ** 2
sqrt2s = twos ** 0.5

powers2 = twos ** torch.tensor([[1, 2], [3, 4]])
#tensor([[ 2.,  4.], [ 8., 16.]])

matrix = torch.randint(0,10, (2,4))
double_vec = torch.ones(4,1)*2

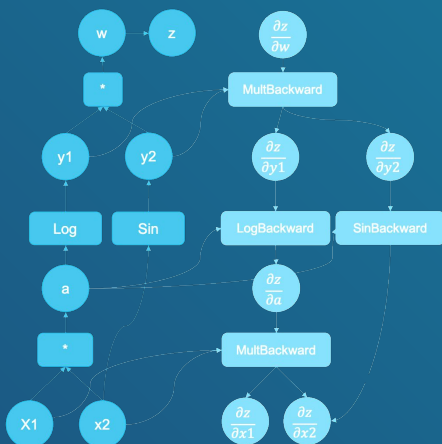
double_vec*matrix
RuntimeError: The size of tensor a (4) must match the size of tensor b (2)
at non-singleton dimension 0

double_vec.T*y
tensor([[2, 7, 2, 8],
       [9, 0, 6, 6]])
tensor([[ 4., 14.,  4., 16.],
       [18.,  0., 12., 12.]])
```

BROADCASTING RULES

- ❖ Each tensor must have at least one dimension - no empty tensors.
- ❖ Comparing the dimension sizes of the two tensors, going from last to first:
 - ❖ Each dimension must be equal, or
 - ❖ One of the dimensions must be of size 1, or
 - ❖ The dimension does not exist in one of the tensors

AUTOMATIC DIFFERENTIATION



In training neural networks, we often use **backpropagation**, adjusting model weights based on the gradient of the loss function.

torch.autograd is a tool that automatically computes these gradients for any computational graph, making it easier to optimize and improve our neural network models.

AUTOMATIC DIFFERENTIATION LIVE CODING

LET'S SEE HOW AUTOGRAD WORKS **IN PRACTICE**

Given a linearly sampled input space
defined over the real interval $[-10, 10]$

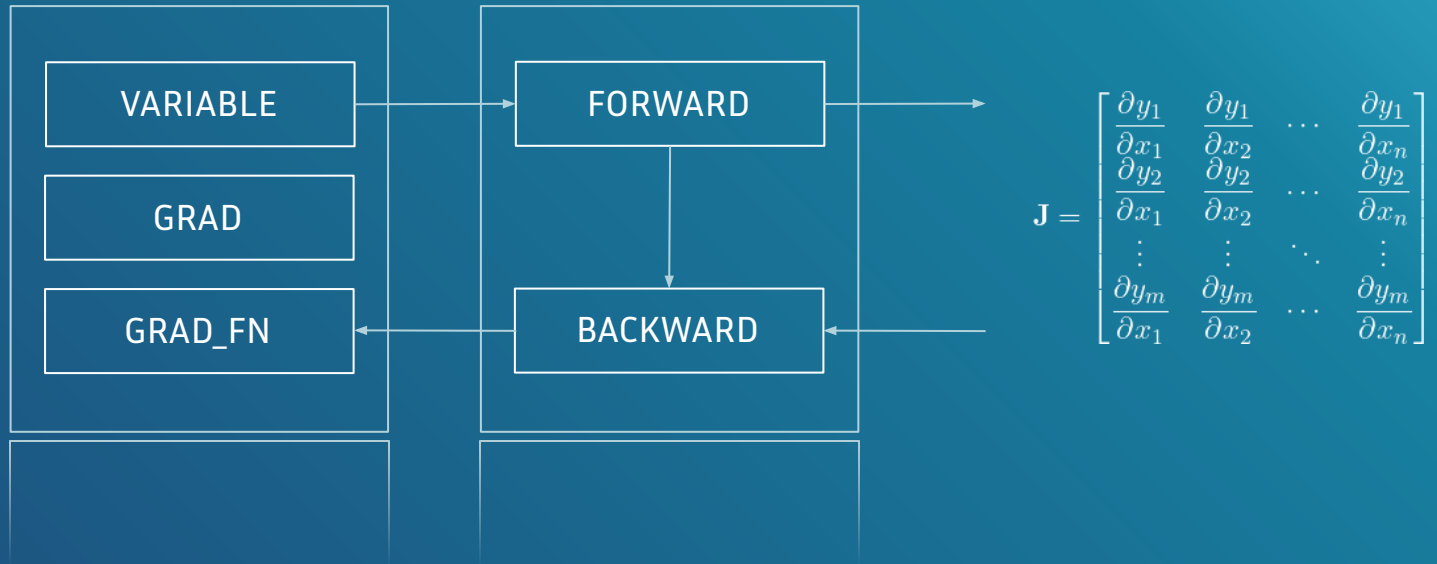
$$\mathbf{x} \in [-10, 10]^n$$

Compute an arbitrary function

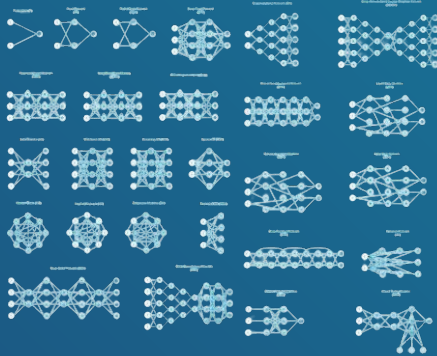
$$\mathbf{y} = 2\mathbf{x}^2 + 5$$

Compute $\frac{\partial y}{\partial x}$ independently of the function \mathbf{y} :

AUTOMATIC DIFFERENTIATION



NEURAL NETWORKS



Modules Building blocks for creating neural network layers
`nn.Linear`, `nn.RNN`, `nn.Conv2D`, `nn.Conv1D`

Activation Functions Various activation functions are available
`nn.ReLU`, `nn.Sigmoid`, `nn.Tanh`

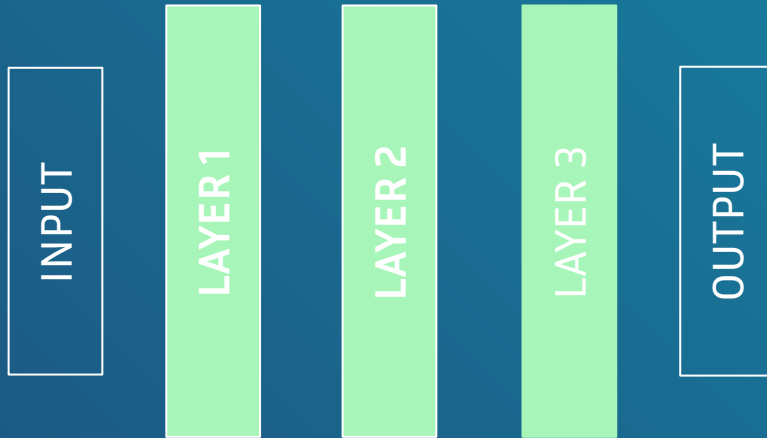
Loss Functions A variety of loss functions are provided
`nn.MSELoss`, `nn.BCELoss`, `nn.CrossEntropyLoss`

Optimizers Most popular optimization algorithms like
`optim.SGD`, `optim.Adam`, `optim.RMSprop`

Custom Layers The `nn.Module` base class enables the creation of custom layers and models by subclassing it

CUDA Support Neural network models built with `torch.nn` can be easily moved to GPU for faster training.

NEURAL NETWORKS



● ReLU

● SIGMOID

NEURAL NETWORKS nn.Module

```
import torch
import torch.nn as nn

class ThreeLayerNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ThreeLayerNN, self).__init__()

        # Define the layers
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.relu1 = nn.ReLU()

        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.relu2 = nn.ReLU()

        self.layer3 = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()
```

```
def forward(self, x):
    # Define the forward pass
    x = self.layer1(x)
    x = self.relu1(x)

    x = self.layer2(x)
    x = self.relu2(x)

    x = self.layer3(x)
    x = self.sigmoid(x)

    return x

# Create an instance of the network
net = ThreeLayerNN(input_size, hidden_size, output_size)
# Forward pass
output = net(torch.randn(5, input_size))
```

NEURAL NETWORKS nn.Sequential

```
class ThreeLayerNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ThreeLayerNN, self).__init__()

        self.layers = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size),
            nn.Sigmoid()
        )
```

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

# Create an instance of the network
net = ThreeLayerNN(input_size, hidden_size, output_size)
# Forward pass
output = net(torch.randn(5, input_size))
```

NEURAL NETWORKS TRAINING CYCLE

```
# Create an instance of the network
net = ThreeLayerNN(input_size, hidden_size, output_size)

# Loss function and optimizer
criterion = nn.BCELoss()
optimizer = optim.SGD(net.parameters(), lr=learning_rate)

# Training loop
for epoch in range(num_epochs):
    # Forward pass
    outputs = net(train_data)

    # Compute the training loss
    loss = criterion(outputs, train_labels)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Evaluate on the validation set
with torch.no_grad():
    val_outputs = net(val_data)
    val_loss = criterion(val_outputs, val_labels)
```