MACHINE LEARNING TOOLS AND FRAMEWORKS

LORENZO SIMONE
TEACHING ASSISTANT

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

IN SUPREMÆ DIGNITATIS · 1343 ·

INDEX

1. PYTHON LIBRARY
2. NUMPY COMPUTING LIBRARY
3. ML FRAMEWORKS
4. PYTORCH INSTALLATION
5. PYTORCH TENSORS
6. PYTORCH BROADCASTING
7. AUTOMATIC DIFFERENTIATION
8. PYTORCH NEURAL NETWORKS
9. TRAINING CYCLE
10. HANDS-ON EXERCISE
11. ML PROJECT QUESTION ANSWERING

INTRODUCTION

LIBRARIES

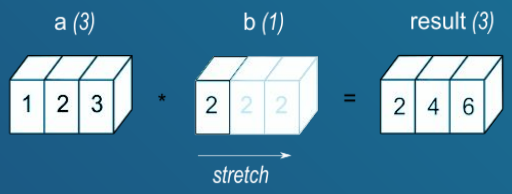
PYTORCH FRAMEWORK

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



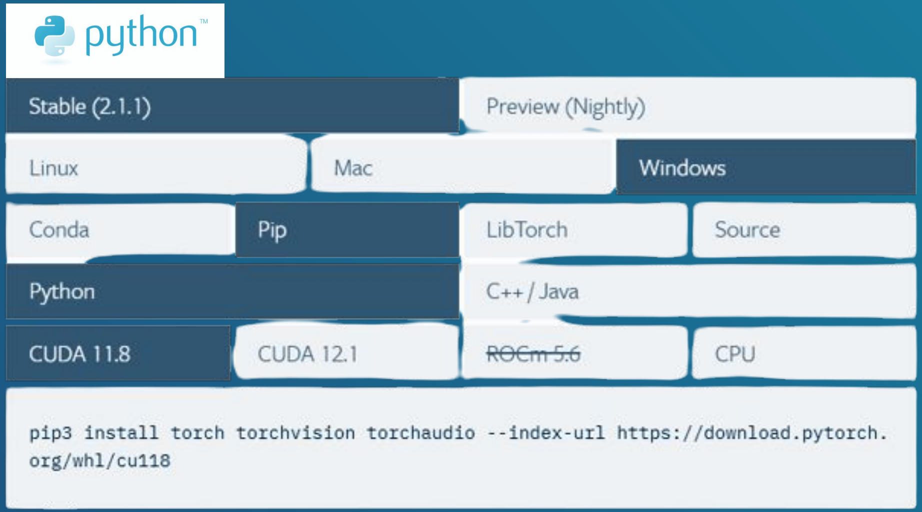
ML FRAMEWORKS

	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/



TENSORS AS MATH OBJECTS

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

```python
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]

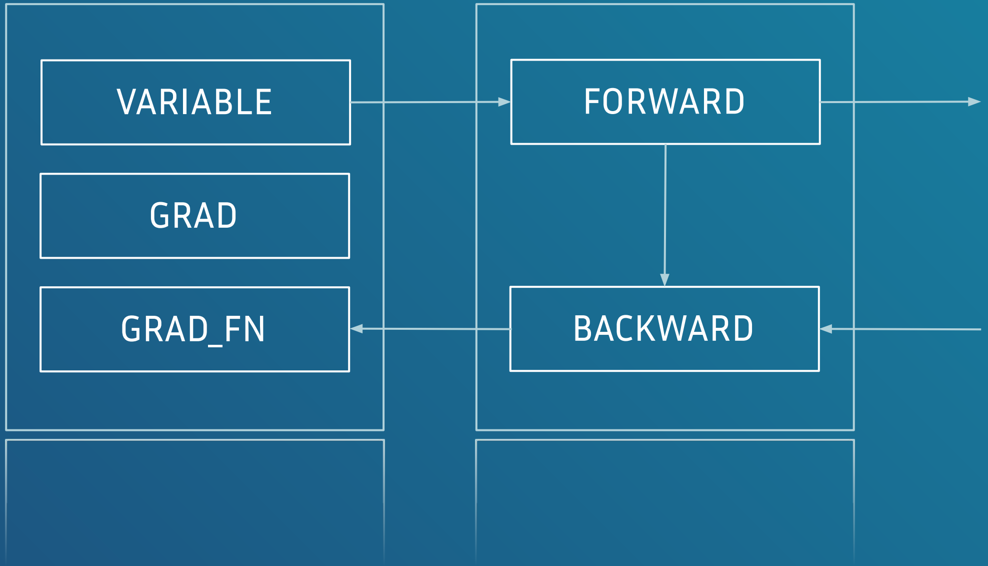
$$\mathbf{x} \in [-10, 10]^n$$

Compute an arbitrary function

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

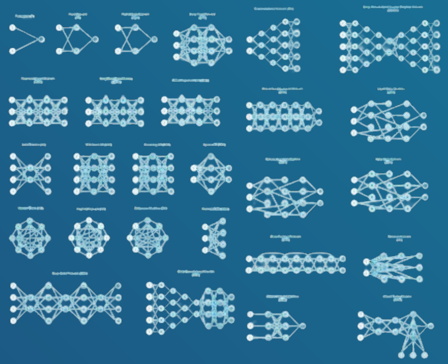
Compute $\frac{\partial y}{\partial x}$ independently of the function $\mathbf{y}$

AUTOMATIC DIFFERENTIATION



$$\mathbf{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

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

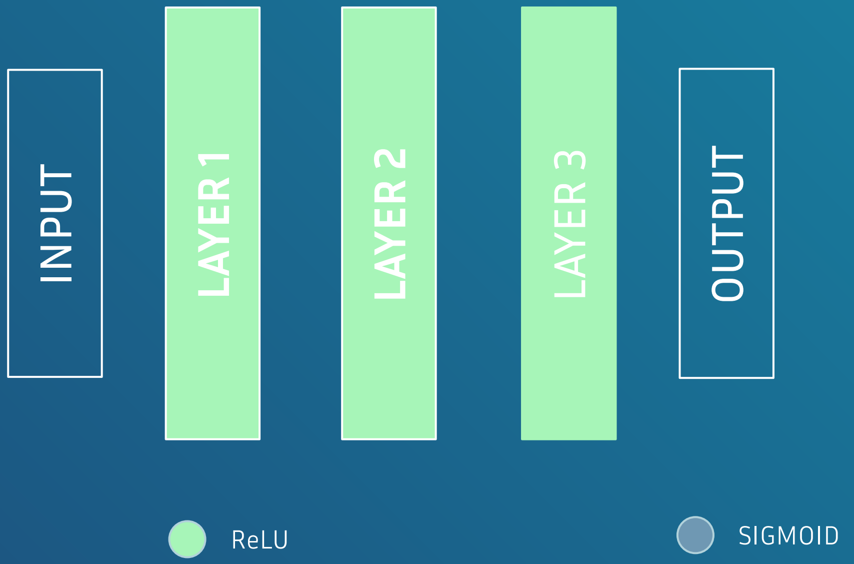
Loss Functions A variety of loss functions are provided
nn.MSELoss,nn.BCELoss, nn.CrossEntropyLoss

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



NEURAL NETWORKS nn.Module

```python
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

```python
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()
        )
```

```python
    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))
```

IN SUPREMÆ DIGNITATIS • 1343 •

NEURAL NETWORKS TRAINING CYCLE

```python
# 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)
```

IN SUPREMÆ DIGNITATIS 1343

NEURAL NETWORKS DATALOADER

```python
# Define your dataset class (replace MyDataset with your actual dataset)
class MyDataset:
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

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

    def __getitem__(self, index):
        return self.data[index], self.labels[index]

# Split the dataset into training and validation sets
train_data, val_data, train_labels, val_labels = train_test_split(data, labels, test_size=0.2, random_state=42)

# Create DataLoader for training data
train_dataset = MyDataset(train_data, train_labels)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

NEURAL NETWORKS MODEL SELECTION

```python
from sklearn.model_selection import train_test_split
from itertools import product

# Split the dataset into training and validation sets
train_data, val_data, train_labels, val_labels = train_test_split(data, labels, test_size=0.2, random_state=42)

# Hyperparameter grid
hidden_sizes = [10, 20, 30]
learning_rates = [0.001, 0.01, 0.1]
num_epochs = 100

# Perform grid search
for hidden_size, learning_rate in product(hidden_sizes, learning_rates):
    # Create an instance of the network with current hyperparameters
    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)

    # Print the results for the current hyperparameters
    print(f'Hyperparameters: Hidden Size={hidden_size}, Learning Rate={learning_rate}')
    print(f'Validation Loss: {val_loss.item():.4f}\n')
```

HANDS-ON EXERCISE

Polynomial regression using torch autograd or numpy library

HANDS-ON EXERCISE

Given a torch tensor input space in the real interval
$x \in [-\pi, \pi]^{2000}$

Compute a 3rd grade polynomial approximation to the sine function
$y = \sin(x) \quad \text{as} \quad \tilde{y} = a + b \cdot x + c \cdot x^2 + d \cdot x^3$

The idea is to iteratively compute the gradient w.r.t. a,b,c and d as
$\frac{\partial \mathcal{L}}{\partial a}, \frac{\partial \mathcal{L}}{\partial b}, \frac{\partial \mathcal{L}}{\partial c}, \frac{\partial \mathcal{L}}{\partial d} \quad \text{where} \quad \mathcal{L} = (\tilde{y} - y)^2$

AUTOGRAD SOLUTION

```python
import torch
import numpy as np

dtype = torch.float
device = "cuda" if torch.cuda.is_available() else "cpu"
torch.set_default_device(device)

# Create Tensors to hold input and outputs.
# By default, requires_grad=False, which indicates that we do not need to
# compute gradients with respect to these Tensors during the backward pass.
x = torch.linspace(-np.pi, np.pi, 2000, dtype=dtype)
y = torch.sin(x)

# Create random Tensors for weights. For a third order polynomial, we need
# 4 weights: y = a + b x + c x^2 + d x^3
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
a = torch.randn((), dtype=dtype, requires_grad=True)
b = torch.randn((), dtype=dtype, requires_grad=True)
c = torch.randn((), dtype=dtype, requires_grad=True)
d = torch.randn((), dtype=dtype, requires_grad=True)

learning_rate = 1e-6
```

AUTOGRAD SOLUTION

```python
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        # Manually zero the gradients after updating weights
        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

k-NN ALGORITHM SCIKIT-LEARN

CLASS A CLASS B

k-NN ALGORITHM SCIKIT-LEARN

We are exploring the task of recognizing handwritten digits using the K-Nearest Neighbors (KNN) algorithm

Data Source: We will use the load_digits dataset from scikit-learn, containing 8x8 pixel images of handwritten digits (0-9).

Data Splitting: We divide the dataset into training and testing sets using a 70-30 Hold-out split ratio

Classifier Selection: We employ the k-NN classifier choosing an appropriate k.

Assessment: Using the trained model, we make predictions on the test set to evaluate the classifier's generalization performance

MNIST Dataset


k-NN ALGORITHM

PYTHON LIBRARY

ML PROJECT Q&A

This is a collective opportunity for you and your colleagues to clarify project doubts or ask technical questions