

Practical Session 3 – Introduction to PyTorch for Machine Learning & Deep Learning

The objective of this tutorial is to start using the PyTorch library for developing Machine Learning models.

1. Introduction to PyTorch

PyTorch (<https://pytorch.org/>) is a powerful open-source deep learning framework that provides flexibility and ease of use for building machine learning models. It is particularly popular among researchers and practitioners because of its dynamic computational graph and seamless integration with Python. When you are familiar with NumPy, PyTorch might feel quite intuitive as it shares many similarities but also introduces several unique features tailored for deep learning.

PyTorch is a framework designed to accelerate numerical computations, particularly for machine learning and deep learning. Unlike NumPy, PyTorch supports:

- GPU acceleration for faster computations.
- Autograd (Automatic Differentiation): Facilitates gradient calculation for optimization.
- Dynamic computation graphs: Offers flexibility for model experimentation.

Key differences between PyTorch and NumPy:

Feature	NumPy	PyTorch
Data Structure	ndarray	Tensor
GPU Support	No	Yes (with <code>.cuda()</code> support)
Automatic Differentiation	No	Yes (<code>autograd</code>)
Dynamic Graphs	No (static computation)	Yes (dynamic computation)

Installing PyTorch

To install PyTorch, use the following command:

```
pip install torch
```

Or (depending on your environment)

```
conda install torch
```

More options are provided here: <https://pytorch.org/get-started/locally/>

Creating and manipulating Tensors

Tensors vs Arrays

```
# NumPy array
np_array = np.array([[1, 2], [3, 4]])
print("NumPy Array:\n", np_array)

# PyTorch tensor
torch_tensor = torch.tensor([[1, 2], [3, 4]])
print("PyTorch Tensor:\n", torch_tensor)
```

Tensor creation

```
# From list
tensor = torch.tensor([1.0, 2.0, 3.0])

# With specific shapes
zeros = torch.zeros((2, 3))
ones = torch.ones((2, 3))
random = torch.rand((2, 3))

print("Zeros Tensor:\n", zeros)
print("Ones Tensor:\n", ones)
print("Random Tensor:\n", random)
```

Converting between Numpy and PyTorch

```
# NumPy to Tensor
np_array = np.array([1, 2, 3])
tensor = torch.from_numpy(np_array)

# Tensor to NumPy
back_to_numpy = tensor.numpy()
print("Converted Back to NumPy:", back_to_numpy)
```

Operations on Tensors

Basic operations

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# Element-wise operations
print("Addition:", a + b)
print("Multiplication:", a * b)

# Matrix multiplication
mat1 = torch.tensor([[1, 2], [3, 4]])
mat2 = torch.tensor([[5, 6], [7, 8]])
result = torch.matmul(mat1, mat2)
print("Matrix Multiplication:\n", result)
```

Broadcasting

```
# Broadcasting in PyTorch
x = torch.tensor([[1], [2], [3]])
y = torch.tensor([10, 20, 30])

print("Broadcasted Addition:\n", x + y)
```

Automatic Differenciation with PyTorch

Introduction to Autograd: PyTorch tracks operations on tensors with the *requires_grad* attribute set to *True*

```
x = torch.tensor(2.0, requires_grad=True)
y = x**2 + 3*x + 5

# Compute gradients
```

```
y.backward()  
print("Gradient of y with respect to x:", x.grad)
```

Example: Multivariate gradient

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)  
y = x**2  
z = y.sum()  
  
z.backward()  
print("Gradients:", x.grad)
```

Using GPUs with PyTorch

Checking for GPU

```
print("Is CUDA available?", torch.cuda.is_available())  
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  
print("Device:", device)
```

Moving Tensors to GPU

```
# Create a tensor  
x = torch.tensor([1.0, 2.0, 3.0])  
x = x.to(device) # Move to GPU  
print("Tensor on GPU:", x)
```

Note: When tensors are on the GPU, you must move them to the CPU before converting to NumPy arrays:

```
x_gpu = x.cuda() # Moves the tensor to GPU  
x_cpu = x_gpu.cpu().numpy()
```

Performance Comparison

```
import time  
  
# Large tensors  
size = 1000000  
cpu_tensor = torch.rand(size)  
gpu_tensor = torch.rand(size, device="cuda")  
  
# CPU computation  
start = time.time()  
cpu_tensor * cpu_tensor  
print("CPU Time:", time.time() - start)  
  
# GPU computation  
start = time.time()  
gpu_tensor * gpu_tensor  
print("GPU Time:", time.time() - start)
```

Case study: Linear Regression

```
import matplotlib.pyplot as plt

# Generate synthetic data

# True values
true_theta, true_c = 3, 2
x = torch.linspace(0, 10, 100)
y = true_theta * x + true_c + torch.randn(100) * 2 # Add noise

plt.scatter(x, y)
plt.show()

# Initialize parameters
theta = torch.randn(1, requires_grad=True)
c = torch.randn(1, requires_grad=True)

# Learning rate and optimizer
alpha = 0.01
nb_iters = 100
for epoch in range(nb_iters):
    # Predictions
    y_pred = theta * x + c

    # Loss: Mean Squared Error
    loss = ((y - y_pred)**2).mean()

    # Backpropagation
    loss.backward()

    # Update parameters
    with torch.no_grad():
        theta -= alpha * theta.grad
        c -= alpha * c.grad
        theta.grad.zero_()
        c.grad.zero_()

    if epoch % 10 == 0:
        print(f"Epoch {epoch}: Loss = {loss.item()}")

# Final parameters
print("Learned m:", theta.item())
print("Learned c:", c.item())
plt.scatter(x, y)
y_pred = theta * x + c
plt.plot(x, y_pred.detach(), color='red')
plt.show()
```

2. Writing the code for a Neural Network with Pytorch

Based on what you have just learnt, convert the code of the neural network you wrote in Session 2, using PyTorch, and compare the execution performance when run on GPU.

To run on GPU, if your laptop is not equipped, you may use one of these remote jupyter servers, where you can select the execution on GPU:

jupyter.mi90.ec-lyon.fr

This server is accessible within the campus network. If outside, you need to use a VPN. Before executing the notebook, select the kernel "Python PyTorch" to run it on GPU and have access to PyTorch module.

[Google Colaboratory](#)

Before executing the notebook, select the execution on GPU : "Execution" Menu -> "Modify type of execution " and select "T4 GPU".