Lab Instructions - session 6

Introduction to PyTroch

Migeration to PyTroch

PyTorch is specifically developed for deep learning, offering features like GPU acceleration and support for automatic differentiation through its autograd module. A key advantage of PyTorch is its dynamic computational graph, which allows for on-the-fly graph construction during runtime, making it highly flexible for building and debugging neural networks.

In the previous labs, you learned to work with NumPy, a powerful library for numerical computations. However, NumPy operates exclusively on CPUs, which can make it slower for large-scale computations, particularly when dealing with high-dimensional data or complex operations.

To observe the difference in computational speed between CPU and GPU, you can use platforms like **Google Colab** or **Kaggle**, both of which provide free access to CPUs and GPUs for machine learning and other computational tasks. GPUs (Graphics Processing Units) are specifically designed to handle parallel computations efficiently, making them significantly faster than CPUs.

First, in Google Colab, go to Runtime > Change Runtime Type, and select GPU from the Hardware Accelerator dropdown menu.
You can verify whether the GPU is active in your environment by running the following code:

```
>>> import torch
>>> device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
>>> print(f"Using device: {device}")
```

You can also check the CUDA driver version on Linux devices by using the following command:

```
>>> !nvidia-smi
```

You have to bring both your model and your data to the GPU or CPU for computation. It is essential that they are on the same device for the computations to work correctly. You can do this using the following code:

In deep learning, you typically work with a model and a data input, where computations involve operations like the linear combination of weights and data to produce predictions. For these computations to work correctly, both the model and the data must reside on the same device.

```
# copy your tensors to a device  (gpu or cpu)

>>> data = data.to(device)
>>> model = model.to(device)
>>> output = model(data)

# Moves tensor `x` from the CPU to the GPU.
>>> x = x.cuda()

# Moves tensor `x` from the GPU to the CPU.
>>> x = x.cpu()
```

PyTorch works with tensors, which are multidimensional arrays similar to NumPy arrays but with additional capabilities like GPU support and automatic differentiation. For tensor creation, you can use the following instructions to initialize tensors in various ways depending on your needs.

```
# tensor with independent N(0,1) entries
x = torch.randn(*size)

# tensor with all 1's [or 0's]
x = torch.[ones|zeros](*size)

# create tensor from [nested] list or ndarray L
x = torch.tensor(L)

# clone of x
y = x.clone()

# code wrap that stops autograd from tracking tensor history
with torch.no_grad():

# arg, when set to True, tracks computation
requires_grad=True
```

```
x.size()                      # return tuple-like object of dimensions
x = torch.cat(tensor_seq, dim=0)     # concatenates tensors along dim
y = x.view(a,b,...)                  # reshapes x into size (a,b,...)

# reshapes x into size (b,a) for some b
y = x.view(-1,a)

y = x.transpose(a,b)              # swaps dimensions a and b
y = x.permute(*dims)              # permutes dimensions
y = x.unsqueeze(dim)              # tensor with added axis
y = x.unsqueeze(dim=2)            # (a,b,c) tensor -> (a,b,1,c) tensor

# removes all dimensions of size 1 (a,1,b,1) -> (a,b)
y = x.squeeze()

# removes specified dimension of size 1 (a,1,b,1) -> (a,b,1)
y = x.squeeze(dim=1)
```

Example:

1. Direct Tensor Creation

```
# Create a tensor directly from a list or array
>>> tensor = torch.tensor([1, 2, 3])

# Specify data type and device (optional)
>>> tensor_float = torch.tensor([1, 2, 3], dtype=torch.float32)
```

2. Tensors with Specific Values

```
# Zeros Tensor
>>> tensor_zeros = torch.zeros(3, 3)  # 3x3 tensor filled with zeros

# Ones Tensor
>>> tensor_ones = torch.ones(3, 3)  # 3x3 tensor filled with ones

# Full Tensor - 3x3 tensor filled with the value 7
>>> tensor_full = torch.full((3, 3), 7)

# Identity Matrix
```

```
>>> tensor_eye = torch.eye(3)  # 3x3 identity matrix
```

3. Random Tensors

```
# Uniformly Distributed Random Values
>>> tensor_rand = torch.rand(3, 3)  # Random values in [0, 1)

# Normally Distributed Random Values
>>> tensor_randn = torch.randn(3, 3)  # Random values from N(0, 1)

# Random integers in [0, 10)
>>> tensor_randint = torch.randint(0, 10, (3, 3))
```

4. Sequences

```
# Arange (Like np.arange)
>>> tensor_arange = torch.arange(0, 10, 2)  # [0, 2, 4, 6, 8]

# Linspace (Like np.linspace)
>>> tensor_linspace = torch.linspace(0, 1, 5)
# [0.0000, 0.2500, 0.5000, 0.7500, 1.0000]
```

5. From Other Data

```
# From NumPy Arrays
>>> np_array = np.array([1, 2, 3])
>>> tensor_from_numpy = torch.from_numpy(np_array)

# From Existing Tensor (Clone)
>>> tensor_clone = tensor.clone()
```

6. Tensors for Device and Dtype

```
# Specify Device (e.g., CPU or GPU)

>>> tensor_device = torch.zeros(3, 3, device='cuda') if
 torch.cuda.is_available() else torch.zeros(3, 3)

# Specify Data Type
# Double precision
>>> tensor_dtype = torch.zeros(3, 3, dtype=torch.float64)
```

7. Specialized Tensors

```
# Diagonal Matrix
>>> tensor_diag = torch.diag(torch.tensor([1, 2, 3]))

# Range with Step
>>> tensor_step = torch.arange(0, 10, step=2)

# Permutation
# Random permutation of integers [0, 9]
>>> tensor_perm = torch.randperm(10)
```

8. Tensor from Uninitialized Memory

```
# Uninitialized Tensor
# Tensor with uninitialized values
>>> tensor_empty = torch.empty(3, 3)
```

9. Sparse Tensors

```
# For handling sparse data (e.g., large matrices with mostly zero
 values)
# Indices of non-zero elements

>>> indices = torch.tensor([[0, 1, 1], [2, 0, 2]])

# Corresponding values

>>> values = torch.tensor([3, 4, 5])
>>> tensor_sparse = torch.sparse_coo_tensor(indices, values, (2, 3))
```

10. Activating Differential Mode (requires_grad=True)

```
# Enable gradient tracking for tensors used in differentiation
>>> tensor_grad = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
```

Algebra

```
ret = A.mm(B)       # matrix multiplication
ret = A.mv(x)       # matrix-vector multiplication
x = x.t()           # matrix transpose
```

Task 1

In Task 1, you will observe the differences between the implementation using PyTorch and other CPU-based methods. First, run the Task1.py script and extract the required parts for this question. It is recommended to use Google Colab to execute this code.

Task1.py

```python
import cv2
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import random

def feature_extractor(image):
    rows = 3
    cols = 3

    kernels = list()
    output_images = list()

    sobel_kernel = [[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]]
    kernels.append(sobel_kernel)

    A = 5
    for iteration in range(A):
        image_height, image_width = len(image), len(image[0])
        kernel_height, kernel_width = len(kernels[iteration]), len(kernels[iteration][0])
        pad_h, pad_w = kernel_height // 2, kernel_width // 2

        padded_image = [[0] * (image_width + 2 * pad_w) for _ in range(image_height + 2 * pad_h)]
        for i in range(image_height):
            for j in range(image_width):
                padded_image[i + pad_h][j + pad_w] = image[i][j]

        output_image = [[0] * image_width for _ in range(image_height)]
        for x in range(image_height):
            for y in range(image_width):
                sum_value = 0
                for m in range(kernel_height):
                    for n in range(kernel_width):
```

```
                        sum_value += (
                            padded_image[x + m][y + n] * kernels[iteration][m][n]
                        )
                output_image[x][y] = min(max(int(sum_value), 0), 255)

        random_kernel = [[random.randint(-2, 2) for _ in range(cols)] for _ in range(rows)]
        max_value = max(abs(num) for row in random_kernel for num in row)
        n_kernel = [[num / max_value if max_value != 0 else 0 for num in row] for row in random_kernel]
        kernels.append(n_kernel)

        output_images.append(output_image)

    return output_images

image_path = "Lenna.png"
image = Image.open(image_path).convert("L")
image_array = np.array(image).tolist()

opencv_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

convolved_image = feature_extractor(image_array)
convolved_image_array = np.array(convolved_image, dtype=np.uint8)

k=0

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(opencv_image, cmap="gray")

plt.subplot(1, 2, 2)
plt.title("output")
plt.imshow(convolved_image_array[k], cmap="gray")

plt.tight_layout()
plt.show()
```

- Explain the code.
- Write a minimal implementation using NumPy.
- Increase the value of A to 10 and repeat the implementation using PyTorch for both CPU and GPU.
- Measure the performance (execution time) of the NumPy, PyTorch (CPU), and PyTorch (GPU) implementations (you can use torch.nn.functional.conv2d).
- Verify that the outputs from the GPU match those from the CPU implementation.

Task 2

Given the function $f(x) = \sin(x^T Ax) \cdot \exp(-x^T Ax)$,where A is a symmetric matrix, calculate its gradient $\nabla f(x)$ with respect to x (a random vector) using the following methods:

- Derive the analytical gradient formula and compute it.
- Use numerical approximation (finite differences) to estimate the gradient.
- Utilize PyTorch's autograd to compute the gradient.

```
import torch
import numpy as np

size = 1000  # Size of the tensor
A = torch.randn(size, size)
A = (A + A.T) / 2

# your code


print("Numerical Gradient:", numerical_grad)
print("PyTorch Autograd Gradient:", autograd_grad)
```

- Compare the results from all three methods. Explain why the numerical approximation might differ slightly from the analytical and autograd results.