CS 405: Deep Learning | Lab

Experiment 2: Implement Fundamentals and Practical Use of PyTorch

Student Name: Nabeel Shan
Registration No: 468752

1. Objective

Hands-on experience with PyTorch by exploring tensor operations, computational graphs, and neural network modules for deep learning tasks in Python.

2. Setup & Installation

Install the required dependencies and import PyTorch.

Import Libraries

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt

print(f"PyTorch Version: {torch.__version__}")
```

```
PyTorch Version: 2.9.0+cpu
```

Tensors & Shapes

```
# --- 1.1 Intro to Tensors ---

integer = torch.tensor(1234)
decimal = torch.tensor(3.14159265359)

print(f"`integer` is a {integer.ndim}-d Tensor: {integer}")
print(f"`decimal` is a {decimal.ndim}-d Tensor: {decimal}")

# Vectors
fibonacci = torch.tensor([1, 1, 2, 3, 5, 8])
count_to_100 = torch.tensor(range(100))
print(f"`fibonacci` is a {fibonacci.ndim}-d Tensor with shape: {fibonacci.shape}")
```

```
`integer` is a 0-d Tensor: 1234
`decimal` is a 0-d Tensor: 3.1415927410125732
`fibonacci` is a 1-d Tensor with shape: torch.Size([6])
```

Defining higher-order Tensors

```
'''TODO: Define a 2-d Tensor (Matrix)'''
# Creating a simple 2x3 matrix
matrix = torch.tensor([[1.0, 2.0, 3.0],
                       [4.0, 5.0, 6.0]])

assert isinstance(matrix, torch.Tensor), "matrix must be a torch Tensor object"
assert matrix.ndim == 2

'''TODO: Define a 4-d Tensor.'''
# Use torch.zeros to initialize a 4-d Tensor of zeros with size 10 x 3 x 256 x 256.
images = torch.zeros(10, 3, 256, 256)

assert isinstance(images, torch.Tensor), "images must be a torch Tensor object"
assert images.ndim == 4, "images must have 4 dimensions"
assert images.shape == (10, 3, 256, 256), "images is incorrect shape"

print("-" * 20)
```

```
print("Success: 2D and 4D Tensors created correctly.")
print(f"Images Tensor Shape: {images.shape}")
```

```
--------------------
Success: 2D and 4D Tensors created correctly.
Images Tensor Shape: torch.Size([10, 3, 256, 256])
```

Computational Graphs

```
# --- 1.2 Computations on Tensors ---

# Simple Addition
a = torch.tensor(15)
b = torch.tensor(61)
c1 = torch.add(a, b)
c2 = a + b
print(f"Simple Add Result: {c1}")

# --- TODO BLOCK 2: Defining Tensor computations ---

def func(a, b):
    '''TODO: Define the operation for c, d, e.'''
    # We define a sample computation graph:
    # c = a + b
    # d = b + 1
    # e = c * d

    a_t = torch.tensor(a)
    b_t = torch.tensor(b)

    c = a_t + b_t        # TODO: Operation 1
    d = b_t + 1          # TODO: Operation 2
    e = c * d            # TODO: Operation 3
    return e

# Execute computation
a, b = 1.5, 2.5
e_out = func(a, b)
print(f"Computation Graph Output (e_out): {e_out}")
```

```
Simple Add Result: 76
Computation Graph Output (e_out): 14.0
```

Neural Networks - Manual Layer

```
class OurDenseLayer(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(OurDenseLayer, self).__init__()
        # Initialize weights and bias randomly
        self.W = torch.nn.Parameter(torch.randn(num_inputs, num_outputs))
        self.bias = torch.nn.Parameter(torch.randn(num_outputs))

    def forward(self, x):
        # z = x * W + b
        z = torch.matmul(x, self.W) + self.bias

        # y = sigmoid(z)
        y = torch.sigmoid(z)
        return y
```

```
# Test the manual layer
num_inputs = 2
num_outputs = 3
layer = OurDenseLayer(num_inputs, num_outputs)
x_input = torch.tensor([[1.0, 2.0]]) # Shape [1, 2]
y = layer(x_input)

print(f"Manual Layer Input shape: {x_input.shape}")
print(f"Manual Layer Output shape: {y.shape}")
print(f"Manual Layer Output result: {y}")
```

```
Manual Layer Input shape: torch.Size([1, 2])
Manual Layer Output shape: torch.Size([1, 3])
Manual Layer Output result: tensor([[0.0704, 0.9569, 0.2546]], grad_fn=<SigmoidBackward0>)
```

Sequential API

```
# --- Defining a Neural Network using Sequential API ---

n_input_nodes = 2
n_output_nodes = 3

'''Use the Sequential API to define a neural network with a
    single linear (dense!) layer, followed by non-linearity to compute z'''

model = nn.Sequential(
    nn.Linear(n_input_nodes, n_output_nodes), # The Dense Layer
    nn.Sigmoid()                              # The Activation
)
```

```
# Test
x_input = torch.tensor([[1.0, 2.0]])
y_seq = model(x_input)
print(f"Sequential API Output: {y_seq}")
```

```
Sequential API Output: tensor([[0.1686, 0.2505, 0.1762]], grad_fn=<SigmoidBackward0>)
```

Subclassing nn.Module

```
# --- Defining a model using subclassing ---

class LinearWithSigmoidActivation(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(LinearWithSigmoidActivation, self).__init__()

        '''define a model with a single Linear layer and sigmoid activation.'''
        self.linear = nn.Linear(num_inputs, num_outputs)
        self.activation = nn.Sigmoid()

    def forward(self, inputs):
        linear_output = self.linear(inputs)
        output = self.activation(linear_output)
        return output
```

```
# Test
model_sub = LinearWithSigmoidActivation(2, 3)
y_sub = model_sub(x_input)
print(f"Subclassed Model Output: {y_sub}")
```

```
Subclassed Model Output: tensor([[0.3302, 0.5814, 0.6778]], grad_fn=<SigmoidBackward0>)
```

Custom Behavior

```
# --- Custom behavior with subclassing ---

class LinearButSometimesIdentity(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(LinearButSometimesIdentity, self).__init__()
        self.linear = nn.Linear(num_inputs, num_outputs)

    '''Implement behavior where network outputs input unchanged
       if isidentity is True.'''
    def forward(self, inputs, isidentity=False):
        if isidentity:
            return inputs
        else:
            return self.linear(inputs)
```

```
# Test
model_identity = LinearButSometimesIdentity(num_inputs=2, num_outputs=3)
x_input = torch.tensor([[1.0, 2.0]])

'''pass the input into the model with and without identity.'''
out_with_linear = model_identity(x_input, isidentity=False)
out_with_identity = model_identity(x_input, isidentity=True)

print(f"Input: {x_input}")
print(f"With Linear: {out_with_linear}")
print(f"With Identity: {out_with_identity}")
```

```
Input: tensor([[1., 2.]])
With Linear: tensor([[ 1.1093, -1.6380,  0.2754]], grad_fn=<AddmmBackward0>)
With Identity: tensor([[1., 2.]])
```

Autograd & Optimization

```
# --- Automatic Differentiation & Optimization ---

# 1. Simple Gradient Example (y = x^2)
x = torch.tensor(3.0, requires_grad=True)
y = x ** 2
y.backward()
print("dy_dx of y=x^2 at x=3.0 is:", x.grad)
```

```
dy_dx of y=x^2 at x=3.0 is: tensor(6.)
```

```
# 2. Function Minimization with Gradient Descent
# Problem: Minimize L = (x - x_f)^2
# Target value x_f
x_f = torch.tensor(5.0)

# Initial guess for x (random)
x = torch.tensor(0.0, requires_grad=True)

# Learning rate
lr = 0.1

print(f"\nStart Optimization: Target is {x_f.item()}, Initial Guess is {x.item()}")

# Optimization Loop
for i in range(20): # Run 20 steps
    # 1. Compute Loss
    loss = (x - x_f) ** 2

    # 2. Backpropagation (Compute Gradients)
    loss.backward()

    # 3. Update Parameters (Gradient Descent)
    # We wrap in torch.no_grad() because this update shouldn't be tracked by autograd
    with torch.no_grad():
        x -= lr * x.grad

    # 4. Zero Gradients for next step
    x.grad.zero_()

    if i % 5 == 0:
        print(f"Step {i}: x = {x.item():.4f}, Loss = {loss.item():.4f}")

print(f"Final Result: x = {x.item():.4f} (Close to target {x_f.item()})")
```

```

Start Optimization: Target is 5.0, Initial Guess is 0.0
Step 0: x = 1.0000, Loss = 25.0000
Step 5: x = 3.6893, Loss = 2.6844
Step 10: x = 4.5705, Loss = 0.2882
Step 15: x = 4.8593, Loss = 0.0309
Final Result: x = 4.9424 (Close to target 5.0)
```

```
Start coding or generate with AI.
```