Introduction to Deep Learning
Exercise Session 1: Neural Network Basics with PyTorch

MSc Computer Science

Week 1

Overview

In this exercise session, you will:

- Get familiar with PyTorch tensors and basic operations
- Implement and visualize activation functions
- Understand forward propagation through manual computation
- Build a simple neural network from scratch

Prerequisites: Python 3.x, PyTorch installed. See installation instructions at `https://://pytorch.org/`

1 PyTorch Basics (15-20 minutes)

1.1 Exercise 1.1: Creating and Manipulating Tensors

(a) Create a PyTorch tensor from the list `[1, 2, 3, 4, 5]` and print its shape and data type.

(b) Create a 3×4 tensor filled with random values from a standard normal distribution.

(c) Create a 2×3 tensor of ones and a 2×3 tensor of zeros.

(d) Reshape the tensor from (b) to shape $(2, 6)$ and then to $(12, 1)$.

1.2 Exercise 1.2: Tensor Operations

(a) Create two tensors $\mathbf{x} = [1, 2, 3]$ and $\mathbf{w} = [0.5, -0.3, 0.8]$. Compute their dot product using `torch.dot()`.

(b) Create a matrix $\mathbf{W} \in \mathbb{R}^{3\times 2}$ with random values and a vector $\mathbf{x} \in \mathbb{R}^2$. Compute $\mathbf{Wx}$ using `torch.matmul()` or the `@` operator.

(c) Create two tensors of shape $(2, 3)$ and add them element-wise. Then try adding a tensor of shape $(3,)$ to the $(2, 3)$ tensor (broadcasting).

Key takeaway: PyTorch tensors work similarly to NumPy arrays but can be moved to GPU and track gradients.

2 Activation Functions (20 minutes)

2.1 Exercise 2.1: Implement Activation Functions

Implement the following activation functions using PyTorch operations:

(a) **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$

(b) **Tanh:** $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

(c) **ReLU:** $\text{ReLU}(z) = \max(0, z)$

Each function should accept a tensor `z` and return a tensor of the same shape.

2.2 Exercise 2.2: Visualize Activation Functions

(a) Create a tensor `z` with 100 values evenly spaced between -5 and 5.

(b) Apply each activation function to `z` and plot all three on the same graph.

(c) Compare your implementations with PyTorch's built-in functions: `torch.sigmoid()`, `torch.tanh()`, and `torch.relu()`.

2.3 Exercise 2.3: Derivatives of Activation Functions

(a) The derivative of sigmoid is: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

(b) The derivative of tanh is: $\tanh'(z) = 1 - \tanh^2(z)$

(c) The derivative of ReLU is: $\text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$

Implement these derivatives and plot them alongside the activation functions.

2.4 Exercise 2.4: Vanishing Gradient Problem

(a) For sigmoid, compute $\sigma'(z)$ for $z \in \{-10, -5, 0, 5, 10\}$.

(b) What do you notice about the gradient when $|z|$ is large?

(c) Why is this a problem for training deep networks?

3 Forward Propagation by Hand (25 minutes)

3.1 Exercise 3.1: Manual Computation

Consider a tiny neural network with:

- Input: $\mathbf{x} = [1, 2]$
- Hidden layer: 2 neurons with weights $\mathbf{W}^{(1)} = \begin{bmatrix} 0.5 & -0.3 \\ 0.8 & 0.2 \end{bmatrix}$ and bias $\mathbf{b}^{(1)} = [0.1, -0.2]$
- Output layer: 1 neuron with weights $\mathbf{w}^{(2)} = [1, -0.5]$ and bias $b^{(2)} = 0.3$
- Use ReLU activation for hidden layer, no activation for output

Compute by hand (show all steps):

(a) Pre-activation of hidden layer: $\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$

(b) Activation of hidden layer: $\mathbf{a}^{(1)} = \text{ReLU}(\mathbf{z}^{(1)})$

(c) Pre-activation of output layer: $z^{(2)} = \mathbf{w}^{(2)T}\mathbf{a}^{(1)} + b^{(2)}$

(d) Final output: $a^{(2)} = z^{(2)}$

3.2 Exercise 3.2: Verify with PyTorch

Implement the same computation using PyTorch tensors and verify your hand calculations.

4 Building a Simple Neural Network (30 minutes)

4.1 Exercise 4.1: Implement a 2-Layer MLP

Create a neural network class that:

- Has an `__init__` method that initializes two linear layers
- Input dimension: 2, Hidden dimension: 4, Output dimension: 1
- Has a `forward` method that applies: Linear → ReLU → Linear
- Use `torch.nn.Linear` for the layers

Starter code:

```python
import torch
import torch.nn as nn

class SimpleMLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(SimpleMLP, self).__init__()
        # TODO: Initialize layers

    def forward(self, x):
        # TODO: Implement forward pass
        return out
```

4.2 Exercise 4.2: Test on XOR Problem

(a) Create the XOR dataset:

```python
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)
```

(b) Create an instance of your `SimpleMLP` with random weights.

(c) Pass the XOR inputs through the network and observe the outputs.

(d) Note: The outputs will be random/incorrect since we haven't trained the network yet!

4.3 Exercise 4.3: Visualize Decision Boundary

(a) Create a grid of points in the range $[-0.5, 1.5] \times [-0.5, 1.5]$.

(b) Pass these points through your network to get predictions.

(c) Plot the decision boundary using a contour plot.

(d) Overlay the XOR data points on the plot.

(e) What do you observe? Can the random network solve XOR?

Hint: Use `torch.meshgrid()` to create the grid and `matplotlib.pyplot.contourf()` for plotting.

4.4 Exercise 4.4: Count Parameters

(a) Calculate the total number of parameters in your network manually.

(b) Verify using: `sum(p.numel() for p in model.parameters())`

(c) Formula: For a layer with input dim d_{in} and output dim d_{out}:
Number of parameters $= d_{in} \times d_{out} + d_{out}$ (weights + biases)

5 Wrap-up Questions

1. Why can't we use a linear activation function in hidden layers?
2. Why is ReLU more popular than sigmoid for hidden layers in modern deep learning?
3. How many parameters would a 3-layer MLP have with dimensions [10, 50, 50, 5]?
4. Can a single-layer perceptron (no hidden layer) solve the XOR problem? Why or why not?

Additional Resources

- PyTorch Documentation: `https://pytorch.org/docs/`
- PyTorch Tutorials: `https://pytorch.org/tutorials/`
- Prince, Chapter 3 (Shallow Neural Networks)
- Prince, Chapter 4 (Deep Neural Networks)

For Next Week

We'll learn about backpropagation and how to actually train these networks! Make sure you understand:

- The chain rule from calculus
- Matrix multiplication and dimensions
- How to compute gradients of simple functions