CEG5301 Machine Learning with Applications:

Part I: Homework #2

Important note: the due date is 16/09/2025 23:59. You should submit your scripts to the folder in CANVAS. Late submission is not allowed unless it is well justified. Please include the Python code (in order to run the code, you can setup environment on your laptop or use web-based platforms, such as Google Colab) as attachment if computer experiment is involved.

Q1. **Rosenbrock's Valley Problem (20 Marks)**

Consider the Rosenbrock's Valley function:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

which has a global minimum at (x, y) = (1,1) where f(x, y) = 0. Now suppose the starting point is randomly initialized in the open interval (0, 1) for x and y, find the global minimum using:

a). Steepest (Gradient) descent method

$$w(k+1) = w(k) - \eta g(k)$$

with learning rate η = 0.001. Record the number of iterations when f(x, y) converges to (or very close to) 0 and plot out the trajectory of (x, y) in the 2-dimensional space. Also plot out the function value as it approaches the global minimum. What would happen if a larger learning rate, say η = 0.5, is used?

(**10 Marks**)

b). Newton's method (as discussed on page 13 in the slides of lecture Four)

$$\Delta w(n) = -H^{-1}(n)g(n)$$

Record the number of iterations when f(x, y) converges to (or very close to) 0 and plot out the trajectory of (x, y) in the 2-dimensional space. Also plot out the function value as it approaches the global minimum.

(**10 Marks**)

Q2. Function Approximation (30 Marks)

,

$$y = 1.2\sin(\pi x) - \cos(2.4\pi x) \qquad \text{for } x \in [-2, 2].$$

The training set is generated by dividing the domain [-2, 2] using a uniform step length 0.05, while the test set is constructed by dividing the domain [-2, 2] using a uniform step length 0.01. You may use the python to implement a MLP and do the following experiments:

a). Use the **sequential mode** with BP algorithm and experiment with the following different structures of the MLP: 1-n-1 (where n = 1, 2, ..., 10, 20, 50, 100). (Check

appendix sample code for some basic reference). For each architecture plot out the outputs of the MLP for the test samples after training and compare them to the desired outputs. Try to determine whether it is under-fitting, proper fitting or over-fitting. Identify the minimal number of hidden neurons from the experiments, and check if the result is consistent with the guideline given in the lecture slides. Compute the outputs of the MLP when x=-3 and +3, and see if the MLP can make reasonable predictions outside of the domain of the input limited by the training set.

(15 Marks)

b). Use the **batch mode** with scipy.optimize.least_squares algorithm to repeat the above procedure. (Note: set hidden layer to be n = 1, 2, ..., 10, 20 to avoid parameter numbers larger than data number)

hint with sample code:

result = scipy.optimize.least_squares(cost_function, initial_params, **method='lm'**)

(15 Marks)

Important note: There are many design and training issues to be considered when you apply neural networks to solve real world problems. We have discussed most of them in the lecture four. Some of them have clear answers, some of them may rely on empirical rules, and some of them have to be determined by trial and error. I believe that you will have more fun playing with these design parameters and making your own judgment rather than solving the problem with a prescribed set of parameters. Hence, there is no standard answer to this problem, and the marking will be based upon the whole procedure rather than the final classification accuracy. (Check Google to get familiar with the functions that you don't know and Google everything that confuse you.)

Appendix

Sample code of a simple sequential BP with one hidden layer and tanh activation function.

Link:

https://colab.research.google.com/drive/1KaeVDJbBWib6oIHliWEodxRjZwPGKlMC?usp=sharing

Codes:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple neural network with one hidden layer
class SimpleNet(nn.Module):
    def __init__(self, N=10):  # N is the number of hidden units
        super(SimpleNet, self).__init__()
```

```
        self.hidden = nn.Linear(1, N)  # One input, N hidden
units
        self.activation = nn.Tanh()
        self.output = nn.Linear(N, 1)  # N hidden units, one
output

    def forward(self, x):
        x = self.activation(self.hidden(x))
        x = self.output(x)
        return x

# Create the model with N hidden units
N = 2  # Define number of hidden neurons
model = SimpleNet(N)

# Define the loss function (Mean Squared Error)
criterion = nn.MSELoss()

# Define the optimizer (Stochastic Gradient Descent)
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training data (batch size = 1)
x_train = torch.tensor([[1.0], [2.0], [3.0], [4.0]],
dtype=torch.float32)
y_train = torch.tensor([[3.0], [5.0], [7.0], [9.0]],
dtype=torch.float32)  # y = 2x + 1

# Training loop
num_epochs = 20
for epoch in range(num_epochs):
    total_loss = 0
    for i in range(len(x_train)):
        x = x_train[i].unsqueeze(0)  # Batch size = 1
        y = y_train[i].unsqueeze(0)

        # Forward pass
        y_pred = model(x)
        loss = criterion(y_pred, y)
        total_loss += loss.item()

        # Backward pass
        optimizer.zero_grad()  # Reset gradients
        loss.backward()  # Compute gradients
        optimizer.step()  # Update weights
```

```
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
{total_loss:.4f}')

# Testing the model
with torch.no_grad():
    test_input = torch.tensor([[5.0]])  # Expecting ~11 (2*5 + 1)
    test_output = model(test_input)
    print(f'Prediction for input 5: {test_output.item():.4f}')
```