Homework 3

Uzma Hamid
Machine Learning

March 14, 2025

Question 1. [10 pts]
This question is related to the loss functions we discussed in class.

(a) Describe what are hinge loss, logistic regression loss, and 0-1 loss mathematically. Describe their similarities and differences using the unified picture we developed in class.
Answer:
Hinge Loss: The support vector machines employ hinge loss to obtain a classifier with "maximum-margin". The loss function in support vector machines is defined as follows:

$$\frac{1}{n}\sum_{i=1}^{n} L_h(y_i, s_i), \tag{10}$$

where L_h is the hinge loss:

$$L_h(y_i, s_i) = \max(0, 1 - y_i s_i). \tag{11}$$

Different from the zero-one loss and perceptron loss, a data point may be penalized even if it is predicted correctly.

Logistic Regression Loss: Logistic regression employs the log loss to train classifiers. The loss function used in logistic regression can be expressed as

$$\frac{1}{n}\sum_{i=1}^{n} L_{\log}(y_i, s_i), \tag{8}$$

where $L_{\log}$ is the log loss, defined as

$$L_{\log}(y_i, s_i) = \log(1 + e^{-y_i s_i}).$$

One-Zero Loss: The zero-one loss aims at measuring the number of prediction errors for a classifier. For a given input x_i, the classifier makes a correct prediction if $y_i s_i >$

0. Otherwise, it makes a wrong prediction. Therefore, the zero-one loss function can be described as follows:

$$\frac{1}{n}\sum_{i=1}^{n} L_{0/1}(y_i, s_i), \tag{2}$$

where $L_{0/1}$ is the zero-one loss defined as

$$L_{0/1}(y_i, s_i) = \begin{cases} 1, & \text{if } y_i s_i < 0, \\ 0, & \text{otherwise.} \end{cases}$$

Similarities and Differences: All three loss functions aim to penalize incorrect predictions, but they do so in different ways:
- **Hinge Loss** is sensitive to the margin between the decision boundary and the data point. It does not penalize correct predictions that are far from the boundary, but it heavily penalizes points that are close to or on the wrong side of the boundary.
- **Logistic Regression Loss** assigns a continuous penalty based on how confident the model is in its predictions, especially when the predicted probability is far from the true label. It works well for probabilistic classification.
- **Zero-One Loss** only cares about whether the prediction is correct or incorrect, making it less sensitive to the confidence of the prediction.

In the unified picture, we can think of these losses in terms of their penalty structures. Hinge loss has a margin-based penalty, logistic regression has a probabilistic-based penalty, and zero-one loss is a simple binary indicator function.

(b) By relying on the result in the above question, consider a data point that is correctly classified and distant from the decision boundary. Why would SVM's decision boundary be unaffected by this point, but the one learned by logistic regression be affected?

The key difference between SVM (Support Vector Machine) and logistic regression is how each algorithm handles data points that are correctly classified and far from the decision boundary.

SVM: The decision boundary in an SVM is determined by the support vectors, which are the data points closest to the boundary (or within the margin). Data points that are correctly classified and far from the decision boundary do not affect the position of the boundary because they do not contribute to the margin. These points have zero hinge loss, so they do not influence the optimization process.

Logistic Regression: LogReg tries to minimize the log loss by adjusting the weights based on all data points. Even if a data point is correctly classified and far from the decision boundary, its predicted probability (which will be close to 1 for positive class or 0 for negative class) still contributes to the loss function. Thus, logistic regression may adjust the decision boundary slightly in response to these points, even if they are correctly classified, as it aims to maximize the likelihood of all points.

Therefore, while the SVM decision boundary is robust to correctly classified points far from the boundary, logistic regression's decision boundary can still be influenced by them due to its probabilistic nature and the continuous nature of the log loss function.

Question 2. [20 pts]
Exercise 7.8 (e-Chap:7-15) in the book "Learning from Data". For your convenience, the question is repeated below:

Exercise 7.8: Repeat the computations in Example 7.1 for the case when the output transformation is the identity. You should compute $s(l)$, $x(l)$, $\delta(l)$, and $\frac{\partial e}{\partial W^{(l)}}$.
Note the "output transformation" refers to the nonlinear function in the output layer only.

In Example 7.1, the output layer uses `tanh` as the activation function. In Exercise 7.8, we replace this with the identity function. Meaning:

$$h(\mathbf{x}) = s^{(3)}$$

instead of:

$$h(\mathbf{x}) = \tanh(s^{(3)})$$

This affects the backpropagation computations.

Step 2: Forward Propagation

Layer 1 (Input to First Hidden Layer):

$$W^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}$$

and the input vector:

$$\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

The pre-activation values:

$$\mathbf{s}^{(1)} = (W^{(1)})^T \mathbf{x}^{(0)} = \begin{bmatrix} 0.1 & 0.3 \\ 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.7 \\ 1.0 \end{bmatrix}$$

Applying the `tanh` activation function:

$$\mathbf{x}^{(1)} = \tanh(\mathbf{s}^{(1)}) = \begin{bmatrix} \tanh(0.7) \\ \tanh(1.0) \end{bmatrix} \approx \begin{bmatrix} 0.60 \\ 0.76 \end{bmatrix}$$

Layer 2 (First Hidden Layer to Second Hidden Layer)

$$W^{(2)} = \begin{bmatrix} 0.2 \\ 1 \\ -3 \end{bmatrix}, \quad \mathbf{x}^{(2)} = \begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix}$$

$$\mathbf{s}^{(2)} = W^{(2)T}\mathbf{x}^{(1)} = 0.2 + (1)(0.60) + (-3)(0.76) = -1.48$$

Applying `tanh`:

$$\mathbf{x}^{(2)} = \tanh(\mathbf{s}^{(2)}) \approx -0.90$$

Layer 3 (Second Hidden Layer to Output Layer)

$$W^{(3)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\mathbf{s}^{(3)} = W^{(3)T}\mathbf{x}^{(2)} = 1(1) + 2(-0.90) = -0.8$$

Since the output transformation is the identity function:

$$h(x) = s^{(3)} = -0.8$$

Step 3: Backpropagation

Error Term for Output Layer

$$\delta^{(3)} = s^{(3)} - y = -0.8 - 1 = -1.8$$

Error Term for Hidden Layer 2

$$\delta^{(2)} = (1 - x^{(2)2})W^{(3)T}\delta^{(3)}$$

$$= (1 - (-0.90)^2) \begin{bmatrix} 1 \\ 2 \end{bmatrix} (-1.8)$$

$$= (0.19) \begin{bmatrix} -1.8 \\ -3.6 \end{bmatrix} = \begin{bmatrix} -0.34 \\ -0.68 \end{bmatrix}$$

Error Term for Hidden Layer 1

$$\delta^{(1)} = (1 - x^{(1)2})W^{(2)T}\delta^{(2)}$$

$$= \begin{bmatrix} (1 - (0.6)^2) \\ (1 - (0.76)^2) \end{bmatrix} \cdot \begin{bmatrix} 0.2 \\ 1 \\ -3 \end{bmatrix} \begin{bmatrix} -0.34 \\ -0.68 \end{bmatrix}$$

$$= \begin{bmatrix} -0.44 \\ 0.88 \end{bmatrix}$$

Step 4: Compute Gradients

$$\frac{\partial e}{\partial W^{(3)}} = x^{(3)}(\delta^{(3)})^T = \begin{bmatrix} 1 \\ -0.90 \end{bmatrix} (-1.8) = \begin{bmatrix} -1.8 \\ 1.62 \end{bmatrix}$$

$$\frac{\partial e}{\partial W^{(2)}} = x^{(2)}(\delta^{(2)})^T = \begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix} \begin{bmatrix} -0.34 \\ -0.68 \end{bmatrix} = \begin{bmatrix} -0.34 \\ -0.42 \\ -0.53 \end{bmatrix}$$

$$\frac{\partial e}{\partial W^{(1)}} = x^{(0)}(\delta^{(1)}) = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} -0.44 \\ 0.88 \end{bmatrix} = \begin{bmatrix} -0.44 & 0.88 \\ -0.88 & 1.75 \end{bmatrix}$$

Key Differences from Example 7.1:

- Final output $x^{(3)}$ is not passed through tanh
- $\delta^{(3)}$ calculation doesn't include the derivative of tanh
- Different numerical values in backpropagation due to these changes

Question 3. [10 pts]
You'd like to train a fully-connected neural network with 5 hidden layers, each with 10 hidden units (i.e., $d^{(\ell)} = 10$ for these hidden layers using the notations in the textbook). The input is 20-dimensional and the output is a scalar. What is the total number of trainable parameters in your network (including bias in your calculation)?

Network Architecture:

- Input dimension: 20
- Hidden layers: 5, each with 10 hidden units ($d^{(\ell)} = 10$)
- Output dimension: 1 (scalar)

1. Input Layer to First Hidden Layer:

Number of weights $= 20 \times 10 = 200$

Number of biases $= 10$

Total parameters for this layer $= 200 + 10 = 210$

2. Between Hidden Layers:

Number of weights between two consecutive hidden layers $= 10 \times 10 = 100$

Number of biases $= 10$

Total parameters for each hidden layer connection $= 100 + 10 = 110$

Number of connections between hidden layers $= 4$

Total parameters for hidden layer connections = 4 $\times$110 = 440

3. Last Hidden Layer to Output Layer:

Number of weights = 10 $\times$1 = 10

Number of biases = 1

Total parameters for this layer = 10 + 1 = 11

4. Total Trainable Parameters:

Total trainable parameters = 210 + 440 + 11 = 661

The total number of trainable parameters in the network is **661**

Question 4. [60 pts]
Multi-layer perceptron models for Handwritten Digits Classification:
this assignment, you will implement the multi-layer perceptron model using PyTorch on a partial dataset from MNIST. In this classification task, the model will take a 16×16 image of handwritten digits as input and classify the image into three different classes (0,1,2). The `"data"` folder contains the dataset, which has already been split into a training set and a testing set. All data examples are saved in dictionary-like objects using the `".npz"` file format. For each data sample, the dictionary key `'x'` indicates its raw features, which are represented by a 256- dimensional vector where the values between $[-1, 1]$ indicate grayscale pixel values for a 16×16 image. In addition, the key `'y'` is the label for a data example, which can be 0, 1, or 2. The `"code"` folder provides the starting code (main training logic has already been implemented). You must implement the models using the starting code. Please learn the PyTorch tutorial before starting this assignment: `https://pytorch.org/tutorials/beginner/basics/intro.html`

(a) (20 points) In the function `prepare_X` from `"code/DataReader.py"`, please copy your implementation from HW2, and don't change other parts in `"code/DataReader.py"`. Then, implement the multi-layer perceptron model in `"code/MLP.py"`. Please use only one hidden layer in the MLP (Hint: you can use `torch.nn.Linear` class in PyTorch). In this assignment, we will use a new nonlinear function ReLU that will be discussed later in this class (PyTorch has implemented this function, so you do not need to be concerned about how to implement it). It outputs the same value if the input is positive and outputs zero if the input is negative (Hint: use `torch.nn.functional.relu`). The sequential layers are:

```
Linear → ReLU → Linear (3 output channels)
```

Refer to Code in MLP.py

(b) (20 points) Implement the evaluation function in `"code/main.py"`. This function should evaluate the classification accuracy of the trained model on the test set. Please remember to load the best model checkpoint (based on validation accuracy) before running the evaluation on the test set.

Refer to Code in main.py - def evaluation

(c) (20 points) Tune the `HIDDEN_SIZE` hyperparameter in `"code/main.py"` to find the best model performance on the test set. Please include the `HIDDEN_SIZE` values you've tried and the corresponding classification accuracy on the test set in your submission.

Refer to Code in main.py - tuning hidden sizes

For implementation, refer to the CODE.