CSCE 421: Machine Learning (Spring 2025)
Assignment #3

Chayce Leonard

Due: March 20, 2025, 11:59 PM

Problem 1 (10 points)

(a) Loss Functions

Consider a binary classification problem with dataset $X = [x_1, x_2, ..., x_n]$ and labels $Y = [y_1, y_2, ..., y_n]$, where $x_i \in \mathbb{R}^d$ and $y_i \in \{+1, -1\}$. For a linear classifier that computes score $s_i = w^T x_i + b$, we compare three loss functions:

Zero-one Loss:

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

Hinge Loss (SVM):

$$L_h(y_i, s_i) = \max(0, 1 - y_i s_i)$$

Log Loss (Logistic Regression):

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

Key Relationships in Unified View:

1. **Convexity Properties:**
    - Zero-one loss: Non-convex, non-differentiable
    - Hinge loss: Convex but not strictly convex

- Log loss: Strictly convex and smooth

2. **Behavior at Decision Boundary:**
   - Zero-one loss: Step function at $y_i s_i = 0$
   - Hinge loss: Linear when $y_i s_i < 1$, zero when $y_i s_i \geq 1$
   - Log loss: Smooth decay, asymptotically approaching zero
3. **Role as Surrogate Functions:** Both hinge loss and log loss serve as convex upper bounds for the zero-one loss, making them computationally tractable alternatives for optimization.

(b) SVM vs. Logistic Regression

For a point far from the decision boundary ($y_i s_i \gg 1$):

SVM (Hinge Loss):

$$L_h(y_i, s_i) = \max(0, 1 - y_i s_i) = 0$$

When $y_i s_i > 1$, the gradient $\frac{\partial L_h}{\partial w} = 0$, making this point a non-support vector with no influence on the decision boundary.

Logistic Regression (Log Loss):

$$L_{log}(y_i, s_i) = \log(1 + e^{-y_i s_i}) > 0$$

The gradient remains non-zero (though exponentially small), allowing the point to maintain some influence on the decision boundary.

This reflects the fundamental difference in their learning objectives: SVM focuses on maximizing the margin using only support vectors, while logistic regression maintains continuous influence from all points with exponentially decreasing weights based on distance.

This fundamental difference means SVM's decision boundary depends only on the support vectors (points near the margin), while logistic regression's boundary is influenced by all points to varying degrees.

Problem 2 (20 points)

Exercise 7.8 Computations:

For identity output transformation: [See next document]

Problem 3 (10 points)

Neural Network Parameters Calculation

Consider a fully-connected neural network with the following architecture:

- Input dimension: $d^{(0)} = 20$
- 5 hidden layers, each with $d^{(\ell)} = 10$ units, $\ell = 1, \ldots, 5$
- Output: scalar ($d^{(6)} = 1$)

Let's calculate the total number of trainable parameters:

1. **Input Layer to First Hidden Layer:**

$$\begin{aligned} \text{Weights: } & d^{(0)} \times d^{(1)} = 20 \times 10 = 200 \\ \text{Biases: } & d^{(1)} = 10 \\ \text{Subtotal: } & 200 + 10 = 210 \text{ parameters} \end{aligned}$$

2. **Between Hidden Layers:**

$$\begin{aligned} \text{Number of connections: } & 5 - 1 = 4 \\ \text{For each connection:} & \\ \text{Weights: } & d^{(\ell)} \times d^{(\ell+1)} = 10 \times 10 = 100 \\ \text{Biases: } & d^{(\ell+1)} = 10 \\ \text{Per connection: } & 100 + 10 = 110 \text{ parameters} \\ \text{Subtotal: } & 4 \times 110 = 440 \text{ parameters} \end{aligned}$$

3. **Last Hidden Layer to Output Layer:**

$$\begin{aligned} \text{Weights: } & d^{(5)} \times d^{(6)} = 10 \times 1 = 10 \\ \text{Biases: } & d^{(6)} = 1 \\ \text{Subtotal: } & 10 + 1 = 11 \text{ parameters} \end{aligned}$$

Total Number of Trainable Parameters:

$$210 + 440 + 11 = 661 \text{ parameters}$$

Therefore, the fully-connected neural network has a total of 661 trainable parameters. Total number of parameters: 661

Problem 4 (60 points)

0.1 Feature Extraction Process

The feature extraction process is implemented in the `prepare_X` function, which transforms raw input data into meaningful features for the model:

1. **Data Reshaping**
    - Raw input data (`raw_X`) is reshaped into an array of shape $(1, 16, 16)$
2. **Feature Computation**
    - *Measure of Symmetry*: Computed by horizontally flipping the image and normalizing the difference between original and flipped images
    - *Measure of Intensity*: Calculated as $\sum$(pixel values)$/256$
    - *Bias Term*: Constant array of ones with same number of samples as input
3. **Feature Stacking**
    - Features are stacked in order: [Bias, Symmetry, Intensity]

0.2 Model Architecture

The Multi-Layer Perceptron (MLP) model is implemented as a feedforward neural network with the following components:

- **Linear Layer 1**: Input features $\rightarrow$ hidden units
- **ReLU Activation**: Non-linear activation after first layer
- **Linear Layer 2**: Hidden units $\rightarrow$ hidden units
- **Output Layer**: Final layer producing logits for three classes
- **Dropout Layers**: Regularization layers with $p = 0.5$ dropout probability

0.3 PyTorch Implementation

The MLP model is implemented using PyTorch's neural network modules:
Key implementation details:

- Model inherits from `torch.nn.Module`
- Linear layers implemented using `nn.Linear`
- ReLU activation applied using `F.relu`
- Dropout layers with 0.5 probability for regularization
- Forward pass defines the complete data flow through the network

Model Architecture:

- Linear Layer 1
- ReLU Activation
- Linear Layer 2 (3 output channels)

(b) Evaluation Function (20 points)

The evaluation function measures the classification accuracy of our trained MLP model on the test dataset. The function follows these key steps:

1. **Data Loading:** Loads the test data from the specified NPZ file using the provided `load_data` function.
2. **Feature Extraction:** Processes the raw test data through the same feature extraction pipeline used during training via the `prepare_X` function to maintain consistency.
3. **Label Processing:** Processes the test labels for all three classes (0, 1, 2) using the `prepare_y` function.
4. **Tensor Conversion:** Converts the preprocessed numpy arrays into PyTorch tensors to make them compatible with the model.
5. **Model Restoration:** Loads the best model state (saved during training based on validation accuracy) into the model architecture.

6. **Evaluation Mode:** Sets the model to evaluation mode using `model.eval()`
   to disable dropout and other training-specific behaviors.

7. **Inference with Gradient Disabling:** Uses `torch.no_grad()` for efficient inference without gradient computation.

8. **Accuracy Calculation:** Computes classification accuracy by comparing predicted labels with ground truth labels.

The function returns the test accuracy and prints both the best validation accuracy from training and the calculated test accuracy, providing a comprehensive view of the model's performance.

Implementation Details: See Python Script

(c) Hyperparameter Tuning (20 points)

I systematically varied the `HIDDEN_SIZE` parameter to find the optimal model architecture for our MLP classifier. The hidden layer size directly impacts the model's capacity to learn complex patterns in the data. Below are the results of different configurations:

HIDDEN_SIZE	Test Accuracy
3	70.40%
32	86.60%
64	86.11%
128	85.63%
256	85.63%

Analysis of Results

The performance trends reveal several important insights:

- **Small Hidden Layer (HIDDEN_SIZE=3):** With only 3 hidden units, the model achieved a test accuracy of 70.40%. This limited capacity configuration demonstrates that even a minimal network can learn basic patterns but struggles to capture the complexity required for high accuracy.

- **Medium Hidden Layer (HIDDEN_SIZE=32):** Increasing to 32 hidden units dramatically improved performance to 86.60%, representing a 16.20% absolute improvement. This suggests that the model requires a certain threshold of capacity to effectively represent the decision boundaries between the three digit classes.

- **Larger Hidden Layers (HIDDEN_SIZE=64, 128, 256):** Interestingly, further increasing the hidden layer size did not yield additional improvements. In fact, there was a slight decrease in performance, with test accuracies of 86.11%, 85.63%, and 85.63% respectively. This suggests that beyond 32 hidden units, the model may be prone to overfitting or the optimization process becomes more challenging.

Best Performing Model

- **HIDDEN_SIZE:** 32
- **Test Accuracy:** 86.60%
- **Validation Accuracy:** 88.25%

Training Dynamics

The best model (HIDDEN_SIZE=32) showed consistent improvement throughout training:

- Initial validation accuracy: 66.35% (Epoch 1)
- Rapid improvement in early epochs (reaching 83.33% by Epoch 9)
- Gradual refinement in later epochs
- Final validation accuracy: 88.25% (Epoch 35)

Conclusions

This experiment demonstrates the classic machine learning principle of model capacity trade-off. The optimal hidden layer size of 32 provides sufficient capacity to learn meaningful representations without overfitting. The diminishing returns observed with larger hidden layers suggest that for this

relatively simple three-class classification task with limited features (symmetry, intensity, and bias), a moderate network capacity is ideal.

The small gap between validation accuracy (88.25%) and test accuracy (86.60%) indicates good generalization, suggesting that our model architecture and regularization strategy (dropout with p=0.5) are effective.

Exercise 7.8 from "Learning from Data"

Chayce Leonard

3.17.25

Exercise 7.8

Repeat the computations in Example 7.1 for the case when the output transformation is the identity. You should compute $\mathbf{s}^{(l)}$, $\mathbf{x}^{(l)}$, $\boldsymbol{\delta}^{(l)}$ and $\frac{\partial e}{\partial \mathbf{W}^{(l)}}$.

Solution

Given Information

- Weight matrices: $\mathbf{W}^{(1)} = [0.1, 0.2, 0.3, 0.4]^T$, $\mathbf{W}^{(2)} = [0.2, 1, -3]^T$, $\mathbf{W}^{(3)} = [1, 2]^T$
- Data point: $x = 2$, $y = 1$
- Hidden layers use tanh activation, output layer uses identity function

Forward Propagation

Input Layer

$$\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \text{(with bias term)} \tag{1}$$

First Hidden Layer

$$\mathbf{s}^{(1)} = (\mathbf{W}^{(1)})^T \mathbf{x}^{(0)} \tag{2}$$

$$= \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix}^T \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{3}$$

$$= \begin{bmatrix} 0.1(1) + 0.2(2) \\ 0.3(1) + 0.4(2) \end{bmatrix} \tag{4}$$

$$= \begin{bmatrix} 0.7 \\ 1.0 \end{bmatrix} \tag{5}$$

$$\mathbf{x}^{(1)} = \begin{bmatrix} 1 \\ \tanh(0.7) \\ \tanh(1.0) \end{bmatrix} \tag{6}$$

$$= \begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix} \quad \text{(with bias term)} \tag{7}$$

Second Hidden Layer

$$\mathbf{s}^{(2)} = (\mathbf{W}^{(2)})^T \mathbf{x}^{(1)} \tag{8}$$

$$= \begin{bmatrix} 0.2 & 1 & -3 \end{bmatrix} \begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix} \tag{9}$$

$$= 0.2(1) + 1(0.60) - 3(0.76) \tag{10}$$

$$= 0.2 + 0.60 - 2.28 \tag{11}$$

$$= -1.48 \tag{12}$$

$$\mathbf{x}^{(2)} = \begin{bmatrix} 1 \\ \tanh(-1.48) \end{bmatrix} \tag{13}$$

$$= \begin{bmatrix} 1 \\ -0.90 \end{bmatrix} \quad \text{(with bias term)} \tag{14}$$

Output Layer

$$\mathbf{s}^{(3)} = (\mathbf{W}^{(3)})^T \mathbf{x}^{(2)} \tag{15}$$

$$= \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ -0.90 \end{bmatrix} \tag{16}$$

$$= 1(1) + 2(-0.90) \tag{17}$$

$$= 1 - 1.8 \tag{18}$$

$$= -0.8 \tag{19}$$

Since the output transformation is the identity function:

$$\mathbf{x}^{(3)} = \mathbf{s}^{(3)} = -0.8 \tag{20}$$

Backpropagation

Output Layer Sensitivity

For the identity function, the derivative is 1, so:

$$\boldsymbol{\delta}^{(3)} = (\mathbf{x}^{(3)} - y) \cdot \frac{d}{dz} z \tag{21}$$

$$= (-0.8 - 1) \cdot 1 \tag{22}$$

$$= -1.8 \tag{23}$$

Second Hidden Layer Sensitivity

$$\boldsymbol{\delta}^{(2)} = (1 - \tanh^2(-1.48)) \cdot \mathbf{W}_2^{(3)} \cdot \boldsymbol{\delta}^{(3)} \tag{24}$$

$$= (1 - (-0.90)^2) \cdot 2 \cdot (-1.8) \tag{25}$$

$$= 0.19 \cdot 2 \cdot (-1.8) \tag{26}$$

$$= -0.684 \tag{27}$$

First Hidden Layer Sensitivity

$$\boldsymbol{\delta}^{(1)} = \begin{bmatrix} 1 - \tanh^2(0.7) \\ 1 - \tanh^2(1) \end{bmatrix} \odot \left(\begin{bmatrix} 1 \\ -3 \end{bmatrix} \cdot (-0.684) \right) \tag{28}$$

$$= \begin{bmatrix} 0.64 \\ 0.422 \end{bmatrix} \odot \begin{bmatrix} -0.684 \\ 2.052 \end{bmatrix} \tag{29}$$

$$= \begin{bmatrix} -0.438 \\ 0.866 \end{bmatrix} \tag{30}$$

Partial Derivatives

For the First Layer

$$\frac{\partial e}{\partial \mathbf{W}^{(1)}} = \mathbf{x}^{(0)} (\boldsymbol{\delta}^{(1)})^T \tag{31}$$

$$= \begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} -0.438 & 0.866 \end{bmatrix} \tag{32}$$

$$= \begin{bmatrix} -0.438 & 0.866 \\ -0.876 & 1.732 \end{bmatrix} \tag{33}$$

For the Second Layer

$$\frac{\partial e}{\partial \mathbf{W}^{(2)}} = \mathbf{x}^{(1)}(\boldsymbol{\delta}^{(2)})^T \tag{34}$$

$$= \begin{bmatrix} 1 \\ 0.60 \\ 0.76 \end{bmatrix} \cdot (-0.684) \tag{35}$$

$$= \begin{bmatrix} -0.684 \\ -0.410 \\ -0.520 \end{bmatrix} \tag{36}$$

For the Output Layer

$$\frac{\partial e}{\partial \mathbf{W}^{(3)}} = \mathbf{x}^{(2)}(\boldsymbol{\delta}^{(3)})^T \tag{37}$$

$$= \begin{bmatrix} 1 \\ -0.90 \end{bmatrix} \cdot (-1.8) \tag{38}$$

$$= \begin{bmatrix} -1.8 \\ 1.62 \end{bmatrix} \tag{39}$$

Conclusion

The key difference from Example 7.1 is that using the identity function in the output layer makes $\boldsymbol{\delta}^{(3)} = -1.8$ (versus -1.855 with tanh), which propagates through the network affecting all subsequent calculations.