Homework-1 Answers

Author: Sayantani Bhattacharya

//

Question1:

For each loss, answer the following questions. You must provide an explanation for full points.

1. Perceptron Loss

a. For this loss, which point(s) have the highest loss value? Why?

Points where $y \cdot g(x) \leq 0$, will have the highest loss of 1.

From the table:

- Point (1, 2): $y=1, g(X)=-0.7 \Rightarrow y \cdot g(X)=-0.7<0$
- Point (2, 1): $y=-1, g(X)=1.5 \Rightarrow y \cdot g(X)=-1.5<0$
- **Thus Highest Loss Points:** (1, 2) and (2, 1).

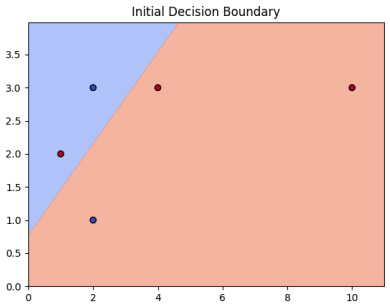
b. For this loss, which point(s) have the lowest loss value? Why?

Points where $y \cdot g(x)>0$ have the lowest loss of 0.

From the table:

- Point (2, 3): $y=-1, g(X)=-1.1 \Rightarrow y \cdot g(X)=1.1>0$
- Point (4, 3): $y=1, g(X)=0.7 \Rightarrow y \cdot g(X)=0.7>0$
- Point (10, 3): $y=1, g(X)=6.1 \Rightarrow y \cdot g(X)=6.1>0$
- Thus, **Lowest Loss Points:** (2, 3), (4, 3), and (10, 3).

c. With this dataset and loss, is it possible to find a different set of model parameters that would have a lower total loss? Don't calculate the loss for any specific w` values; just look at the plot and use your intuition. You may answer "Yes", "No", or "Maybe". If you answer "Yes", give a general description of what the new decision boundary would look like and why that would decrease the loss (e.g., "if you rotated the current boundary clockwise, then …").If you answer "No", give a general argument for why there is no decision boundary that could decrease the loss. If you answer "Maybe", give a general description of a new decision boundary that might decrease the loss, but it's hard to tell without doing the calculations. Also, provide a general argument for why there's no decision boundary that obviously decreases the loss.



Answer: Yes. For instance, observing the above decision boundary, rotating the decision boundary slightly counterclockwise could make (1, 2) correctly classified. A better alignment may increase the number of correct classifications.

d. Is this loss function a good choice for training a multilayer perceptron on a binary classification task? Why or why not?

Answer: No. Perceptron loss is not differentiable, which makes it unsuitable for gradient-based techniques typically used in training multilayer perceptrons.

//

2. Squared Error Loss
   a. Points where the difference $y-g(x)$ - $g(x)$ $y-g(x)$ is largest in magnitude will have the highest loss.

      From the table:

      - Point (10, 3): $y=1, g(X)=6.1 \Rightarrow L=(1-6.1)2=25.01$ (highest).
      - **Highest Loss Point:** (10, 3).

   b. Points where $y-g(x)=0$ have the lowest loss. So in this case the point whose y is closest to $g(X)$ has the lowest loss.

      From the table: **Lowest Loss Point:** (4, 3).

   c. Yes. Adjusting w to make $g(X)$ closer to the corresponding y values could reduce loss.
   d. No. While differentiable, squared error loss is sensitive to outliers and not ideal for classification problems, where hinge or BCE loss performs better

//

3. Binary Cross-Entropy (BCE) Loss

a. BCE loss is highest when $g(X)$predicts the opposite class with high confidence far from 0 in the wrong direction.

From the table:

- Point (2, 1): $y=-1, g(X)=1.5 \Rightarrow L=\ln(1+e1.5)y = -1$
- Point (1, 2): $y=1, g(X)=-0.7 \Rightarrow L=\ln(1+e0.7)y = 1$
- **Highest Loss Points:** (1, 2) and (2, 1).

b. BCE loss is lowest when $g(X)$ is large in magnitude and correctly aligned with y.

From the table:

- Point (10, 3): $y=1, g(X)=6.1 \Rightarrow L\approx 0$
- **Lowest Loss Point:** (10, 3).

c. Maybe. Adjusting w to increase the margins for incorrect predictions while keeping correct predictions confidently classified could lower the total loss.

d. Yes. BCE is differentiable and probabilistic, making it ideal for binary classification tasks, particularly when interpreting predictions as probabilities.

//

4. Hinge Loss

a. Hinge loss is highest for points where $y \cdot g(x) \leq 0$ (incorrect predictions).

From the table:

- Points (1, 2) and (2, 1) have $y \cdot g(x) \leq 0$
- **Highest Loss Points:** (1, 2) and (2, 1).

b. Points with $y \cdot g(x) \geq 1$ have the lowest loss (0).

From the table:

- Point (10, 3): $y=1, g(X)=6.1 \Rightarrow y \cdot g(X)=6.1>1$ $y = 1$
- **Lowest Loss Point:** (10, 3).

c. Yes. Adjusting w to push incorrect points like (1, 2) and (2, 1) further away from the decision boundary could lower total hinge loss.

d. Yes.Hinge loss enforces a margin between classes, improving the generalization of the classifier.

//

Question2:

This single-layer perceptron focuses on implementing different loss functions, each influencing how weight updates occur. It consists of a simple linear classifier with weights w and bias b, trained using gradient descent. With four loss functions, **perceptron loss, squared error loss, binary cross-entropy, and hinge loss**. The **learning rate (lr = 0.01)** and **100 training epochs** were chosen as reasonable defaults, balancing convergence speed with stability. The **sigmoid function** is used in binary cross-entropy to model probabilistic decision boundaries, while hinge loss encourages margin maximization, similar to SVMs. The perceptron loss updates weights only for misclassified points, while squared error loss can lead to gradient explosion, requiring careful tuning. While other learning methods like the standard **perceptron algorithm** could be used, it is only suitable for **linearly separable data** and does not converge for non-linearly separable datasets. By contrast, **gradient descent allows for smoother updates**, enabling learning even in more complex scenarios. However, the perceptron algorithm is computationally cheaper, updating weights only when a misclassification occurs, while gradient-based methods continuously update weights, leading to potentially better decision boundaries. The choice of the method depends on the dataset characteristics and whether the goal is strict classification (perceptron) or margin-based optimization (hinge loss, cross-entropy).

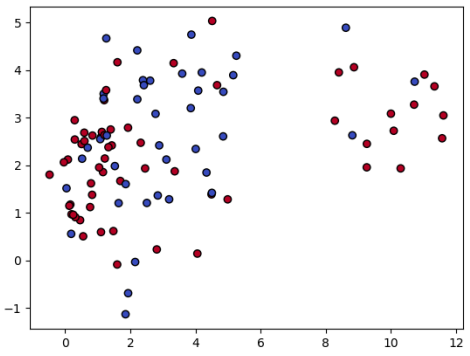
Results with Augmented data (Type1: gaussian):

Dataset representative of the initial 5 samples provided.

It is generated by adding gaussian noise to

randomly chosen points from the initial sample points.

(Attached as augmented_data_gausian.csv)

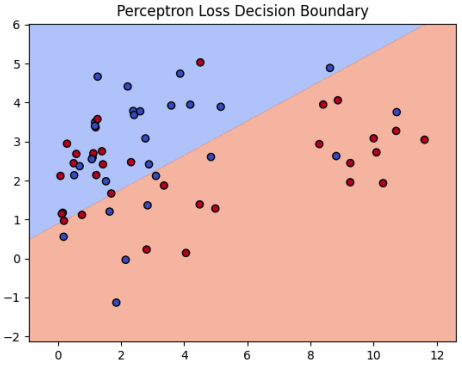


Training with Perceptron Loss...

Weights: [0.39676684 -0.90201702], Bias: 0.7999999999999999

Validation Accuracy: 30.00%

Test Accuracy: 60.00%



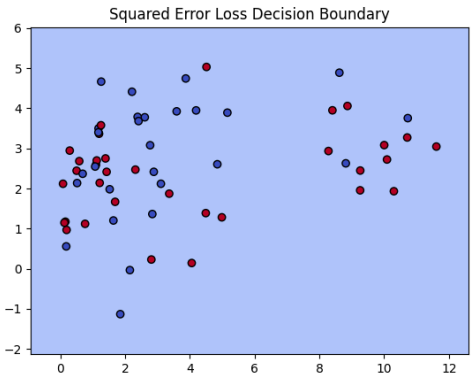
Training with Squared Error Loss...

Weights: [nan nan], Bias: nan

Validation Accuracy: 0.00%

Test Accuracy: 0.00%

Squared error loss is **not ideal** for classification since it can amplify large errors too aggressively. If the value of y-g(x) is too large, as in this case further operations with these numbers often result in nan, leading to complete training failure.

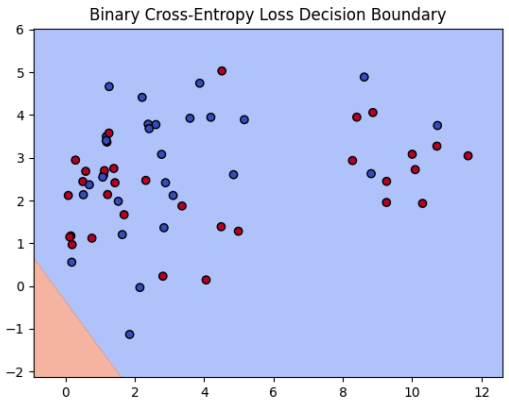


Training with Binary Cross-Entropy Loss...

Weights: [-420.36778968 -379.99504201], Bias: -134.53133908768777

Validation Accuracy: 35.00%

Test Accuracy: 35.00%

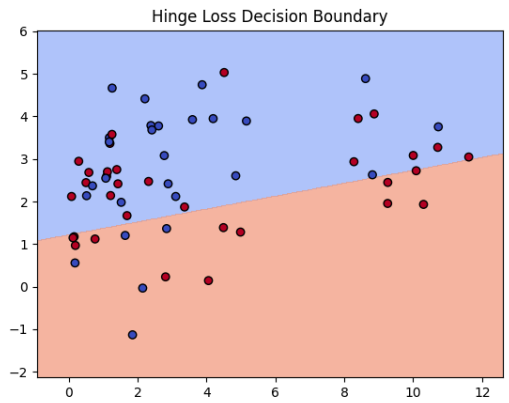


Training with Hinge Loss...

Weights: [0.23647438 -1.56055594], Bias: 1.9000000000000004

Validation Accuracy: 35.00%

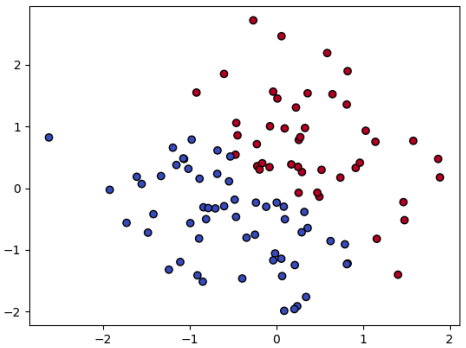
Test Accuracy: 50.00%



Results with Augmented data (Type2: uniform):

Uniform separable dataset:

(Attached as augmented_data_uniform.csv file)

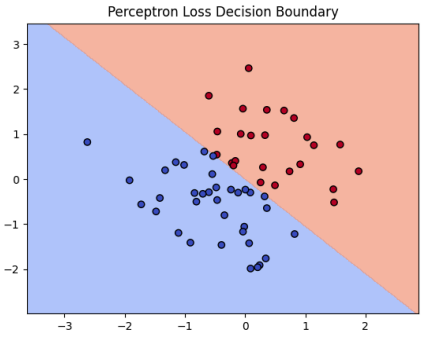


Training with Perceptron Loss...

Weights: [0.26332301 0.2506396], Bias: 0.0

Validation Accuracy: 100.00%

Test Accuracy: 100.00%

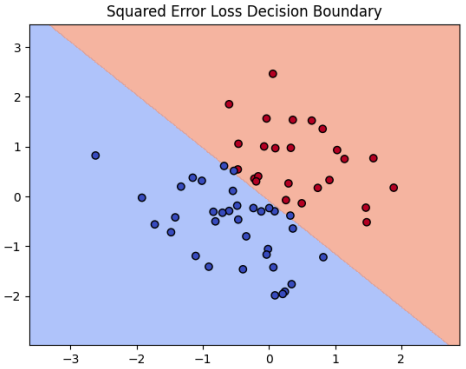


Training with Squared Error Loss...

Weights: [0.53567998 0.50313879], Bias: 0.0442054554956561

Validation Accuracy: 95.00%

Test Accuracy: 100.00%

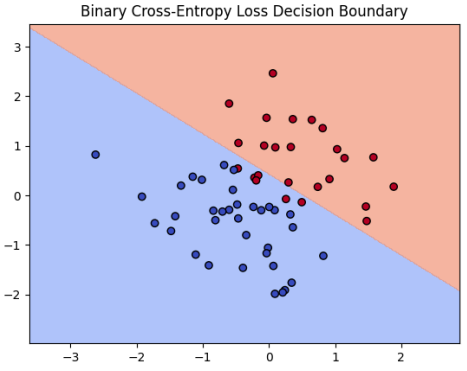


Training with Binary Cross-Entropy Loss...

Weights: [151.85025306 185.67790979], Bias: -79.51072587217323

Validation Accuracy: 95.00%

Test Accuracy: 85.00%

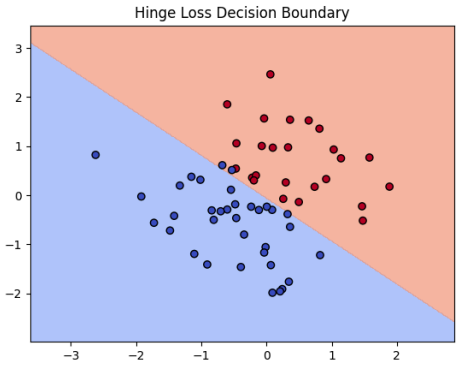


Training with Hinge Loss...

Weights: [4.39791677 5.02188899], Bias: 0.30000000000000004

Validation Accuracy: 95.00%

Test Accuracy: 100.00%



//

Question 3:

a. In the run_one_epoch function, the code uses a combination of where, argmax, and torch.nn.CrossEntropyLoss. How do these calculations work with the model's output to compute the loss and accuracy? Why does the model output a real-valued tensor of shape (N, 2)? You may want to look at the documentation for the loss function.

Answer:

The run_one_epoch function executes one training or evaluation step (based on the value of the train variable) for the neural network model. It takes in a model, optimiser, input data(X) and label(y). The function executes a forward pass on the input, using the model, and stores the output.

Since y contains values that may not be binary, torch.where(y > 0, 1, 0) maps all positive values to 1 and all non-positive values to 0. This ensures the labels are in the correct format for CrossEntropyLoss, which expects class indices (0 or 1). The argmax is used to pick the class with the highest score as the model's prediction. Which is then compared against the label, to calculate the accuracy of the model. Given logits (N, 2), the torch.nn.CrossEntropyLoss applies the softmax function to get probabilities and computes the negative log-likelihood of the correct class.

In binary classification, the model outputs two logits per sample (one for each class). PyTorch's CrossEntropyLoss requires logits of shape (N, C), where C=2 for two classes. The two outputs correspond to logits before softmax, and CrossEntropyLoss automatically applies softmax internally. This formulation allows the model to generalize easily to multi-class problems where C is greater than 2.

b. Read through the run_experiment, pretrain_and_train, and plot_results functions to understand how the figure is being built. In your own words, describe in general what the six panels show. What data is being used in which panels?

Answer:

The six panels:

1. Pretrained model on pretrain data

   This panel visualizes the decision boundary of the model after pretraining on the pretraining dataset. Input: X_pretrain, Label: y_pretrain. It shows whether the model has learned any structure from the pre-training phase before being trained on the main dataset.

2. Pretrained model on train data

   This panel visualizes how the pretrained model performs on the training dataset (before any training on this dataset). Input: X_train, Label: y_train. This gives insight into how well the pretrained model generalizes to a different dataset before fine-tuning.

3. Trained model on train data

   This panel visualizes the decision boundary of the fully trained model on the training dataset. Input: X_train, Label: y_train. It demonstrates how well the model has learned the training data, i.e., whether it has effectively captured the patterns in the dataset.

4. Trained Model on Test Data

   This panel visualizes the decision boundary of the trained model on test data. Input: X_test, Label: y_test. This helps determine the generalization ability of the model. A good model should perform well on unseen test data.

5. Loss per epoch

   A line plot of the training loss and test loss over the training epochs. Loss should decrease over time. Train loss always decreases over time, while testset might not. If the test loss increases while training loss continues decreasing, the model may be overfitting.

6. Accuracy per Epoch

A line plot showing how accuracy changes over training epochs for both training and test data. Accuracy should increase over time. If the test accuracy stagnates or decreases while training accuracy keeps increasing, it suggests overfitting.

//

Question4:

a. Overfitting Experiments: Include at least two experiments that show the model overfitting. Describe what those experiments show and how you know the model is overfitting. What arguments had the most effect on whether your model overfits?

Answer: The clearest way to identify that a model is overfitting is by observing the loss per epoch and accuracy per epoch plots. If the test loss increases while training loss continues decreasing, the model is overfitting. And if the test accuracy stagnates or decreases while training accuracy keeps increasing, it suggests overfitting.

Experiment 2 is the classic example for overfitting. Here I increased the model layer size to [500,100]. This increased the model's ability to memorize training data but led to poor generalization on the test set. The results showed that training accuracy approached **100%**, while test accuracy stagnated around **40-50%**, indicating overfitting. Sample Experiment and Experiment 1 are also examples of overfitting, observing the last two plots using the above statement.

b. Pretraining Experiments: The point of pretraining is to get the model "started" with an easier task (rings) so it can more quickly learn the task we care about (spirals). You can control the amount of ring data, the amount of spiral data, and the number of epochs the model trains on each. In your experiments, assume that all we care about is maximizing the final mean test set accuracy. Based on your experiments, when does pre training help? When does it hurt? Your answer should be at least two paragraphs and reference at least three different experiments you ran.

Answer: Pretraining was tested by varying the **amount of pretraining data**, **epochs**, and **pretraining quality** to observe its impact on test accuracy. In Experiment 5, when I set n_pretrain_epochs =0, the model trained directly on the spiral dataset. It achieved a final mean test set accuracy of 38.5%, suggesting the model struggled to learn from scratch. Especially when compared to Experiment 7, when I set n_pretrain_epochs = 100. The model started with helpful knowledge but remained flexible enough to adapt. This setup resulted in the **highest final mean test accuracy of 61.5%** . And finally in Experiment 6, I assigned n_pretrain_epochs to 5000 (a very high value.) This caused the model to over-specialize on the ring dataset, making it harder to adapt to the spiral data. Test accuracy **dropped to 30.8%.**

The fact that the super specialised model performed even worse than the model without any pre-training shows its detrimental to have a model overfit to initialising data(circle in this case). This indicates the model learned patterns that were **not transferable** to the spiral dataset. The train and test both have pretty low accuracy, and it can be observed from the plot, that it almost remains constant, further proving that model is not learning anything further. **Moderate pre-training epochs (~100-200)** give the model a useful initialization without overfitting to the pretraining task. Having a pre-training task bumps the **final mean test accuracy value from** 38.5% [Experiment 5] to **61.5%[**Experiment 7], which concludes **sufficient pre-training data** ensures the model learns generalizable features.

c. Best Experiment: Across all the experiments you ran, with which arguments did you achieve the highest Final Mean Test Accuracy? What patterns led you to find these arguments? Which arguments had the largest impact on your

experimental results? What was more difficult than you expected? Your answer should be at least two paragraphs and reference at least three different experiments you ran. At least two of these experiments must be different from those included in Pre Training Experiments above.

Answer: The highest final mean test accuracy I achieved was: **92.3%**

The parameters (Experiment 8):

```
kwargs = {
        "title": "Experiment 8",
        "radii": (4,6,10),
        "examples_per_ring": 200,
        "layer_sizes": [256, 64, 10],
        "activation": torch.relu,
        "learning_rate": 0.005,
        "n_pretrain_epochs": 200,
        "n_train_epochs": 1600,
}
```

I increased the model capacity ("layer_sizes": [256, 64, 10]), switched the activation function to **ReLU**, increased the **pre-training dataset size**, and adjusted the **learning rate**. This experiment was built on multiple earlier trials where I noticed that small network sizes ([100,10],[200,20]...) and higher learning rates (0.01) led to suboptimal results, plateauing around **70% accuracy**. The key pattern was deeper networks with a balance between capacity and generalization. A slightly lower learning rate (0.005) also played a crucial role in preventing instability while still allowing steady improvements over more epochs. Through multiple experiments, the most impactful arguments were the network size, activation function, radii of pre-training and learning rate. In early experiments, using ReLU with smaller layers caused slower learning, it performed decently on the training set, but not on the test set.

Changing the radii and number of circles in the data, had an interesting effect. Having radii similar with closer concentricity (kind of mimicking the spirals) gave noticeably better results. At the same time, increasing the number of circles made it very specific, and the model lost generalisability, reducing the mean accuracy of the testset. [Experiment 9, 10 and 11]

Switching to **ReLU** from tanh (Experiment 4 compared to Experiment 1) gave an accuracy bump.

Changing n_pretrain_epochs was also impactful, setting it too low **(e.g., 50 epochs)** made it ineffective, while setting it too high **(e.g., 400 epochs)** caused the model to overfit to the pretraining task (concentric circles) rather than learning the final spiral dataset.

The most surprising challenge was fine-tuning the balance between pretraining and final training—too much pretraining made the network resistant to adapting to the spirals, while too little made it learn slowly. Ultimately, increasing the pre-training dataset size (from 100 to 200 examples per ring) and carefully selecting the number of training epochs (1600) resulted in the best performing model.

//