HW3

October 26, 2024

By: Alex Marzban

Visit the GitHub repository to access the code for this assignment: `https://github.com/marzbana/CS521_HWs/tree/main/hw3`

There is a jupyter notebook to view.

0.1 Problem 1: Interval Bound Propagation (IBP) Training

In this problem, you will implement interval bound propagation (IBP) training for a simple neural network.

0.1.1 Network Description

Implement a fully connected neural network consisting of 3 layers (each *layer* here is a linear layer followed by a ReLU), each of size 50 neurons. Use cross-entropy loss and train on the MNIST dataset. You can use the dataloaders from previous assignments (GitHub Link).

0.1.2 1. IBP Training Procedure

Implement the IBP [GDS+19] training procedure on your network. Use the training tricks used by the paper — gradual reduction of the weighing factor k in Equation 12 of the paper from 1 to 0.5 (see hints below) and gradual increase in the robustness radius ϵ_{train} as the training progresses, starting from 0 to target $\epsilon_{\text{train}} = 0.1$. Report the standard accuracy and robust accuracy (with respect to PGD attack) of your network. Also report the training time and contrast it with that of standard training.

0.1.3 2. Box Verification

Use your box verification implementation from HW-2 and report the verified accuracy (number of test images for which the network is certified robust in an L_∞ radius of ϵ_{test} ball), where ϵ_{test} can take 10 values, evenly between 0.01 and 0.1. Analyze images of some adversarial examples with perturbations within different ϵ_{test} if your network is not certified to be robust for some ϵ_{test}.

0.1.4 Solution Requirements

You should present your solution for this in the form of a Jupyter notebook. We recommend using Google Colab since we can interact with your solution easily, but you can also just upload the notebook to your GitHub repo.

0.1.5 Hints

IBP procedure modifies the training loss function by having another robustness loss term. The overall loss looks as follows, where $\text{CE}(\cdot, \cdot)$ is the cross-entropy loss and z_K are the logits at the last layer K:

$$\mathcal{L}_{\text{IBP}} = k \cdot \text{CE}(z_K, y_{\text{true}}) + (1 - k) \cdot \text{CE}(\hat{z}_K(\epsilon_{\text{train}}), y_{\text{true}})$$

The last linear layer is typically absorbed in the robustness specification, as demonstrated in Equation 9 of the paper. You need to basically redefine the loss function in your training process, including the training tricks in the paper for your IBP training.

0.2 IBP Training Implementation

This code implements **Interval Bound Propagation (IBP)** training for a 3-layer feedforward neural network trained on the **MNIST** dataset.

0.2.1 Network Architecture

The network consists of three fully connected layers, each followed by a ReLU activation function. Each linear layer has 50 neurons.

0.2.2 IBP Function (`IBP`)

Purpose:
Computes the lower and upper bounds of the network's output given an input perturbation ϵ.

Process: 1. **Input Perturbation:**
The input `x` is reshaped and perturbed by ϵ to obtain `z_lower` and `z_upper`, ensuring that the pixel values remain within the valid range [0, 1].

2. **Layer-wise Bound Propagation:**
   For each layer in the model:
   - **Linear Layers:**
     Computes the affine transformation bounds using the absolute weights.
   - **ReLU Layers:**
     Applies the ReLU activation to the bounds.
3. **Output:**
   Returns the final lower and upper bounds of the logits.

0.2.3 Custom Loss Function (`CustomLoss`)

Purpose:
Combines the standard cross-entropy loss with a robustness loss term.

Components: - **Standard Loss ($\mathbf{L}_{\textbf{fit}}$):**
Cross-entropy loss between the logits and the true labels.

- **Robustness Loss ($\mathbf{L}_{\textbf{spec}}$):**
  Cross-entropy loss between the worst-case logits (perturbed by ϵ_{train}) and the true labels.

- **Weighting Factor k:**
  Balances the two loss components:

$$\mathcal{L}_{\text{IBP}} = k \cdot \text{CE}(z_K, y_{\text{true}}) + (1-k) \cdot \text{CE}(\hat{z}_K(\epsilon_{\text{train}}), y_{\text{true}})$$

0.2.4 Epsilon and k Scheduling

- **`getEps` Function:**
  Gradually increases the robustness radius ϵ_{train} from 0 to the final value as training progresses. The ramp-up period is linearly scaled based on the total number of training steps and the methods in the paper.

- **`getK` Function:**
  Gradually decreases the weighting factor k from 1 to 0.5, similar to the ramp-up strategy for ϵ_{train}. This balances the emphasis between standard and robustness loss over time.

0.2.5 Training Function (`train`)

Optimizer and Scheduler:
Utilizes the Adam optimizer with a learning rate scheduler that follows a custom schedule that follows a scaled version of the paper's training strategy on the MINST dataset.

Training Loop: 1. **Epoch Iteration:**
For each epoch, the model processes batches of data:

2. **Batch Processing:**
   - **Forward Pass:**
     Computes the logits and perturbed bounds using the `IBP` function.
   - **Worst-case Logits Calculation:**
     Replaces the logits of the true class with the lower bound to obtain `z_eps`.
   - **Loss Computation:**
     Calculates the custom loss using the `CustomLoss` module.
   - **Backpropagation and Optimization:**
     Performs backpropagation and updates the model parameters.
3. **Metrics Tracking:**
   Tracks and prints the loss and accuracy at specified intervals.

0.2.6 Accuracy and Robustness Results

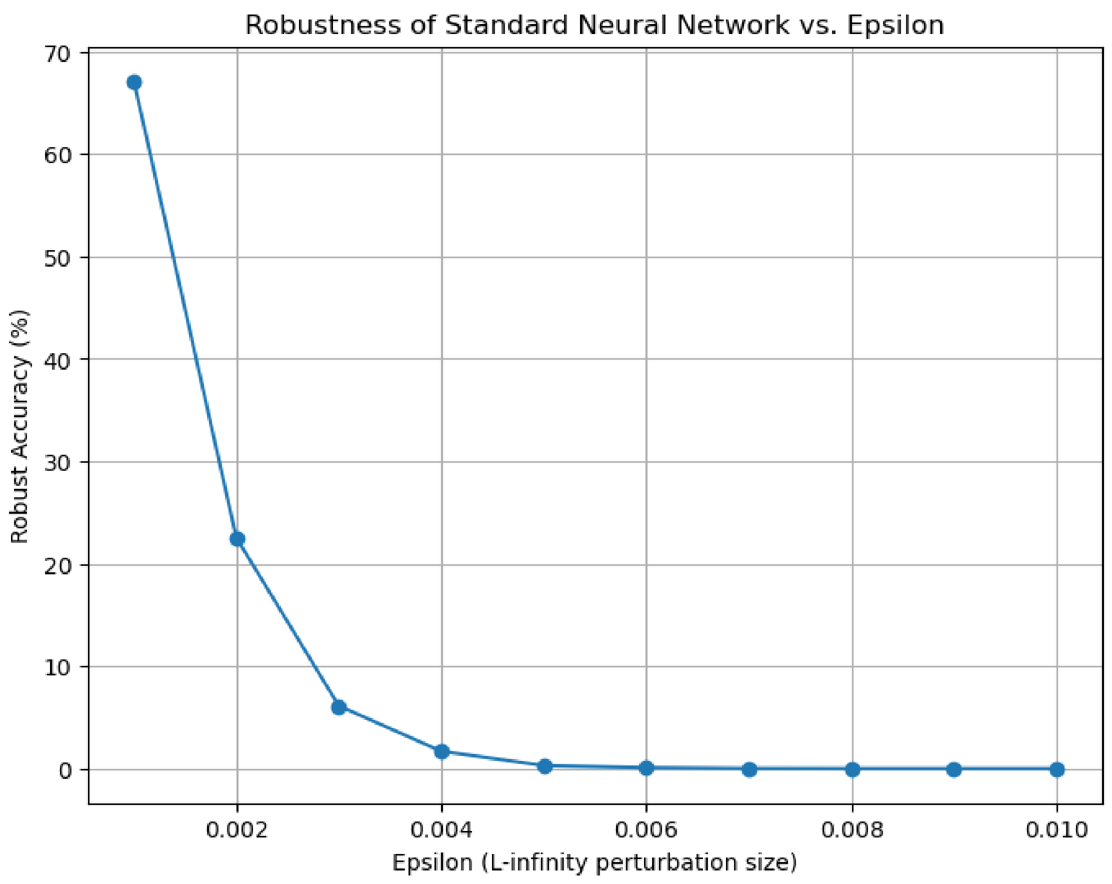
- **Standard Training:**
  - Achieved very high accuracy after 10 epochs.
  - Achieved very high accuracy quickly, reaching **100%** after **60 epochs**.
- **IBP Training:**
  - Achieved very high accuracy after 10 epochs.
  - Standard accuracy slightly decreased to **89.75%** as the model focused more on robustness.
- **Robust Accuracy:**
  - **IBP Trained Model:**
    * **74.71%** robust accuracy at $\epsilon = 0.01$
    * **35.37%** robust accuracy at $\epsilon = 0.1$

    - **Standard Trained Model:**
        - **67.14%** robust accuracy at $\epsilon = 0.001$
        - **0%** robust accuracy at $\epsilon = 0.008$
- **Training Time:**
    - Both standard and IBP trained models required approximately **11.5 minutes** to train for **200 epochs**.
    - The standard model quickly, after about **1.5 mintes**, achieved high accuracy within the initial epochs, aligning with the goal of non-robust training.
    - The IBP trained model took more epochs to balance and maximize robust accuracy while maintaining acceptable standard accuracy.

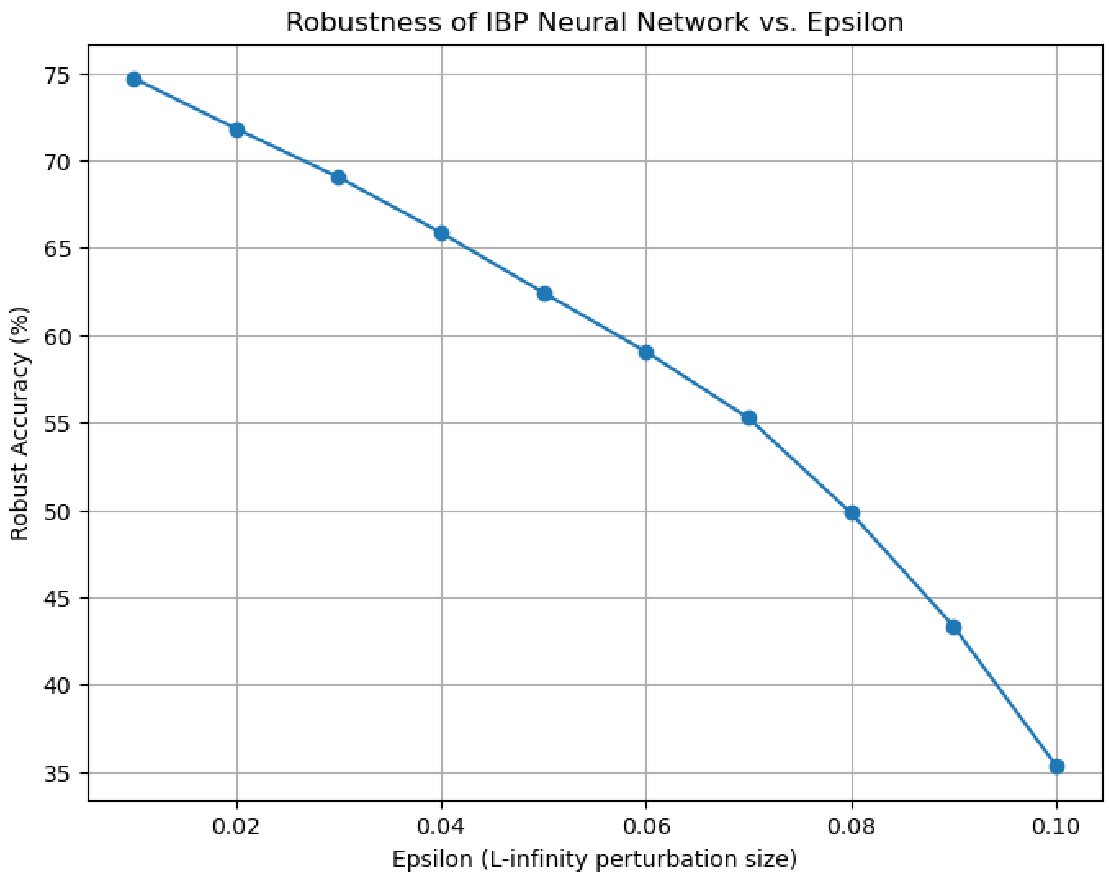
0.2.7 Additional Observations

- **Final k Values:**
    - Experimented with different final values of k (0 and 0.5):
        - Lowering k to **0** resulted in significantly reduced standard accuracy, highlighting the importance of balancing between standard and robustness objectives.

```
Standard Training: 11m 48.7s

Epoch 1/200 - Loss: 0.004657656 - Accuracy: 86.59%
Epoch 10/200 - Loss: 0.000571831 - Accuracy: 98.22%
Epoch 20/200 - Loss: 0.000205949 - Accuracy: 99.34%
Epoch 30/200 - Loss: 0.000113213 - Accuracy: 99.64%
Epoch 40/200 - Loss: 0.000065295 - Accuracy: 99.79%
Epoch 50/200 - Loss: 0.000096978 - Accuracy: 99.69%
Epoch 60/200 - Loss: 0.000002343 - Accuracy: 100.00%
Epoch 70/200 - Loss: 0.000000877 - Accuracy: 100.00%
Epoch 80/200 - Loss: 0.000000339 - Accuracy: 100.00%
Epoch 90/200 - Loss: 0.000000181 - Accuracy: 100.00%
Epoch 100/200 - Loss: 0.000000158 - Accuracy: 100.00%
Epoch 110/200 - Loss: 0.000000138 - Accuracy: 100.00%
Epoch 120/200 - Loss: 0.000000121 - Accuracy: 100.00%
Epoch 130/200 - Loss: 0.000000106 - Accuracy: 100.00%
Epoch 140/200 - Loss: 0.000000093 - Accuracy: 100.00%
Epoch 150/200 - Loss: 0.000000081 - Accuracy: 100.00%
Epoch 160/200 - Loss: 0.000000071 - Accuracy: 100.00%
Epoch 170/200 - Loss: 0.000000062 - Accuracy: 100.00%
Epoch 180/200 - Loss: 0.000000055 - Accuracy: 100.00%
Epoch 190/200 - Loss: 0.000000048 - Accuracy: 100.00%
Epoch 200/200 - Loss: 0.000000042 - Accuracy: 100.00%

IBP Training: 11m 31.9s

Epoch 1/200 - Loss: 0.004897115 - Accuracy: 86.97%
Epoch 10/200 - Loss: 0.003847742 - Accuracy: 95.76%
Epoch 20/200 - Loss: 0.006533807 - Accuracy: 93.95%
Epoch 30/200 - Loss: 0.009500962 - Accuracy: 90.87%
Epoch 40/200 - Loss: 0.009571685 - Accuracy: 89.30%
Epoch 50/200 - Loss: 0.009007934 - Accuracy: 89.34%
Epoch 60/200 - Loss: 0.008436319 - Accuracy: 89.63%
Epoch 70/200 - Loss: 0.008372996 - Accuracy: 89.66%
Epoch 80/200 - Loss: 0.008319871 - Accuracy: 89.68%
Epoch 90/200 - Loss: 0.008247389 - Accuracy: 89.69%
Epoch 100/200 - Loss: 0.008242196 - Accuracy: 89.70%
Epoch 110/200 - Loss: 0.008237586 - Accuracy: 89.70%
Epoch 120/200 - Loss: 0.008232931 - Accuracy: 89.71%
Epoch 130/200 - Loss: 0.008228309 - Accuracy: 89.71%
Epoch 140/200 - Loss: 0.008223877 - Accuracy: 89.73%
Epoch 150/200 - Loss: 0.008219712 - Accuracy: 89.72%
Epoch 160/200 - Loss: 0.008215304 - Accuracy: 89.73%
Epoch 170/200 - Loss: 0.008211124 - Accuracy: 89.73%
Epoch 180/200 - Loss: 0.008206825 - Accuracy: 89.75%
Epoch 190/200 - Loss: 0.008202595 - Accuracy: 89.75%
Epoch 200/200 - Loss: 0.008198521 - Accuracy: 89.75%

Measuring Robustness for epsilon in
[.001,.002,.003,.004,.005,.006,.007,.008,.009,.01]
Epsilon: 0.001 - Robust Accuracy: 67.14%
Epsilon: 0.002 - Robust Accuracy: 22.53%
Epsilon: 0.003 - Robust Accuracy: 6.11%
Epsilon: 0.004 - Robust Accuracy: 1.68%
Epsilon: 0.005 - Robust Accuracy: 0.29%
Epsilon: 0.006 - Robust Accuracy: 0.08%
Epsilon: 0.007 - Robust Accuracy: 0.01%
Epsilon: 0.008 - Robust Accuracy: 0.00%
Epsilon: 0.009 - Robust Accuracy: 0.00%
Epsilon: 0.010 - Robust Accuracy: 0.00%
```



```
Measuring Robustness for epsilon in
[.001,.002,.003,.004,.005,.006,.007,.008,.009,.01]
Epsilon: 0.010 - Robust Accuracy: 74.71%
Epsilon: 0.020 - Robust Accuracy: 71.82%
Epsilon: 0.030 - Robust Accuracy: 69.04%
Epsilon: 0.040 - Robust Accuracy: 65.87%
Epsilon: 0.050 - Robust Accuracy: 62.43%
Epsilon: 0.060 - Robust Accuracy: 59.05%
Epsilon: 0.070 - Robust Accuracy: 55.25%
Epsilon: 0.080 - Robust Accuracy: 49.83%
Epsilon: 0.090 - Robust Accuracy: 43.34%
Epsilon: 0.100 - Robust Accuracy: 35.37%
```



0.3 Problem 2: IBP for Text

In this problem, you will implement an Interval Bound Propagation (IBP) training procedure for a simple neural network on textual data.

0.3.1 Network Description

We will perform certified training for the **Bag-of-Words** architecture as described in [JRGL19]. The model is trained on the **IMDB movie review** dataset (PyTorch Datasets) to predict the sentiment (positive/negative) of each review.

Model Architecture: 1. **Embedding Layer:** - **Pre-trained Embeddings:** Each word in the input is embedded into a pre-trained embedding space using **GloVe** embeddings (GloVe Project Page). - **Learnable Transformation:** The embeddings are transformed into word vectors using a learnable linear transformation layer followed by a ReLU activation function.

2. **Aggregation:**
   - The transformed word vectors are averaged to form a single vector representing the entire input review.
3. **Feedforward Network:**

- **First Layer:** A fully connected layer with a 100-dimensional hidden state followed by a ReLU activation.
- **Second Layer:** Another fully connected layer with a 100-dimensional hidden state followed by a ReLU activation.
- **Output Layer:** A final linear layer to obtain the logits.

4. **Prediction:**
   - The final logits are passed through a softmax activation function to obtain the prediction probabilities for each sentiment class.

Loss Function:
Use **cross-entropy loss** for training the network.

0.3.2 Task

Implement the **IBP [JRGL19]** training procedure on your network. You can reuse your code from **Problem 1** for the same.

Requirements: - **Standard Accuracy:** Report the accuracy of your network on the standard (unperturbed) test set. - **Verified Accuracy:** Report the verified accuracy of your network, which indicates the number of test samples for which the network is certified to be robust within a specified perturbation radius. - **Training Time:** Measure and report the training time of your IBP-trained model. - **Comparison:** Contrast the training time and performance of the IBP-trained model with that of a standard (non-robust) training procedure.

0.3.3 Solution Requirements

You should present your solution for this problem in the form of a Jupyter notebook. We recommend using **Google Colab** since it allows for easy interaction with your solution. Alternatively, you can upload the notebook to your **GitHub** repository.

1 Explanation of Code

1. Data Preprocessing

- First, we load the IMDB datasets, splitting into test and train.
- Then, we use a basic tokenizer from torchtext to process the text inputs.

2. Vocabulary Building

- First, we use the tokenizer to come up with a frequency count of each token in the training data.
- Then, we add special tokens to the vocab in order to account for words that aren't in our tokenizer.
- Next, we create the vocab dictionary that maps words from the tokenized words to indices.
- Then, we load the GloVe embeddings.
- Finally, we create the embedding matrix using GloVe embeddings and random values for words not part of GloVe.

3. Neighborhood Function for Word Embeddings

- We use scikit-learn's nearest neighbors with cosine similarity to map each word index to a list of neighbor indices which defines the list of allowed substitutions for a word.

4. **Interval Bound Propogation**
   - We use the same mechanism from the previous question to derive the linear and relu bounds using IBP.
   - Then, we use the `worst_case_loss` function to propogate the lower and upper purtubations through the network.
   - The worst case loss is computed by calculating the cross-entropy loss for each case and using the `logsumexp` function to approximate the maximum loss over these cases.

5. **Precomputing Word Embedding Bounds**
   - For each word we compute the lower and upper bounds of each word embedding based on the nearest neighbor calculations from earlier.

6. **Model Architecture**
   - The model architecture begins with a GloVe embedding layer and a learnable linear layer followed by averaging a ReLU layer.
   - Then, there are two more layers consisting of an affine layer followed by ReLU.

7. **Training**
   - First, we have a collate function that tokenizes, pads sequences to the same length, and converts labels to 0s an 1s.
   - Then, we ue the torch dataloader to create iterable batches for training.
   - We trained with 10 epochs for both standard and robust training.
   - We use the same epsilon and k scheduling from the first question with a k_final value of .4 and eps_max of 1.
   - We use 100 hidden layer size and 300 embedding dimension size as from the paper.
   - We use the `train_imdb` function to train the standard and robust networks. The function uses k to balance the robust loss with standard loss during robust training.

8. **Evaluation**
   - We use the `certify_example` function to evaluate if an input is certably robust by checking if the differnce between the lower bound of the true class logit is greater than the upper bound of the other class logit.
   - Then, we use the `evaluate_certified_accuracy` function to find the percentage of test examples that the model is certifiably robust for based on the perturbation size epsilon.
   - Next, `graph_llm` is used to iterate through 10 equally spaced epsilons and graph the resulted certified robustness.

9. **Results**
   - The IBP model took about 10 minutes to train on 10 epochs. The standard model took about 2 minutes to train on 10 epochs. The standard model ended with 86% accuracy and the robust model ended with 86% accuracy as well. We plotted certified accuracy for both

models and we can see that the certified model did a lot better. The certified model was evaluated against epsilon in the interval [.01, .1] starting with a robust accuracy of about 41% and ended with 2% robust accuracy. The standard model was evaluated against epsilon in the range [.01, .1] starting with a robust accuracy of 43.7% and quickly falling to 0%. The robust model showed better results with greater epsilon values

1.0.1 Training BoW IBP Network

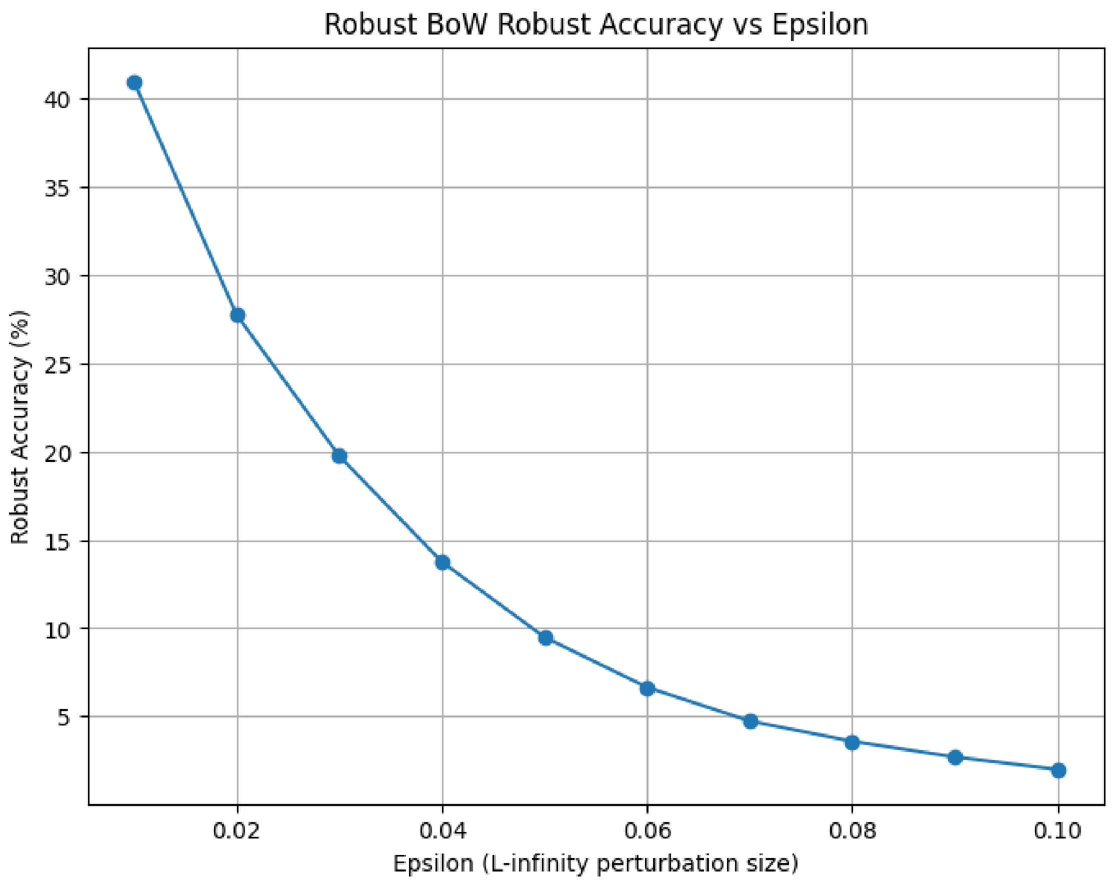
```
Epoch 1, Loss: 0.1181, Accuracy: 0.9589
Epoch 2, Loss: 4.9897, Accuracy: 0.6108
Epoch 3, Loss: 0.9078, Accuracy: 0.6348
Epoch 4, Loss: 0.6936, Accuracy: 0.5525
Epoch 5, Loss: 0.6889, Accuracy: 0.6021
Epoch 6, Loss: 0.6852, Accuracy: 0.6634
Epoch 7, Loss: 0.6830, Accuracy: 0.7469
Epoch 8, Loss: 0.6795, Accuracy: 0.8184
Epoch 9, Loss: 0.6782, Accuracy: 0.8650
Epoch 10, Loss: 0.6763, Accuracy: 0.8609
Test Accuracy: 0.8637
```

1.0.2 Training Standard BoW Network

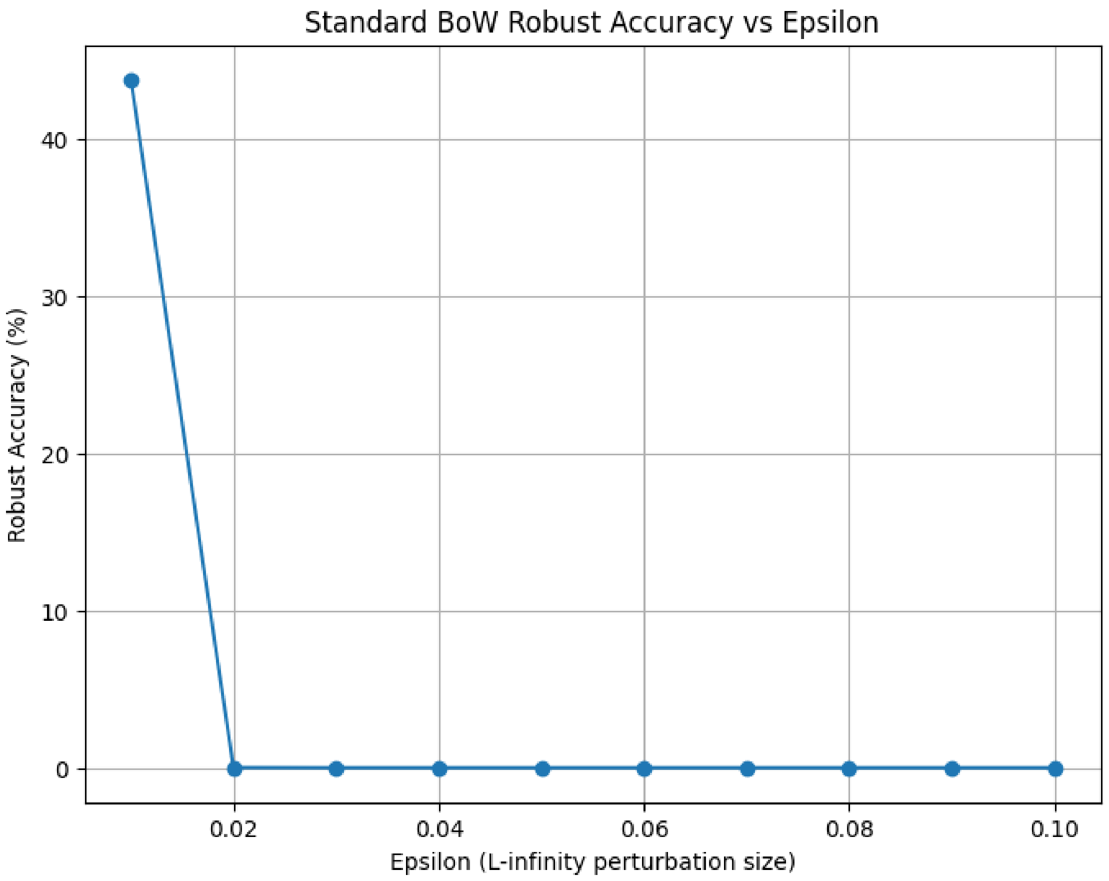
```
Epoch 1, Loss: 0.1240, Accuracy: 0.9602
Epoch 2, Loss: 0.0061, Accuracy: 0.9987
Epoch 3, Loss: 0.0021, Accuracy: 0.9995
Epoch 4, Loss: 0.0016, Accuracy: 0.9998
Epoch 5, Loss: 0.0016, Accuracy: 0.9999
Epoch 6, Loss: 0.0015, Accuracy: 0.9998
Epoch 7, Loss: 0.0015, Accuracy: 0.9997
Epoch 8, Loss: 0.0016, Accuracy: 0.9997
Epoch 9, Loss: 0.0014, Accuracy: 0.9998
Epoch 10, Loss: 0.0014, Accuracy: 0.9998
Test Accuracy: 0.8625
```

1.0.3 Testing Networks for Robustness

```
Epsilon: 0.01, Robust Accuracy: 40.967999999999996%
Epsilon: 0.020000000000000004, Robust Accuracy: 27.752%
Epsilon: 0.030000000000000006, Robust Accuracy: 19.752%
Epsilon: 0.04000000000000001, Robust Accuracy: 13.752%
Epsilon: 0.05000000000000001, Robust Accuracy: 9.484%
Epsilon: 0.06000000000000001, Robust Accuracy: 6.644%
Epsilon: 0.07, Robust Accuracy: 4.732%
Epsilon: 0.08, Robust Accuracy: 3.58%
Epsilon: 0.09000000000000001, Robust Accuracy: 2.692%
Epsilon: 0.1, Robust Accuracy: 1.992%
```



```
Epsilon: 0.01, Robust Accuracy: 43.763999999999996%
Epsilon: 0.020000000000000004, Robust Accuracy: 0.012%
Epsilon: 0.030000000000000006, Robust Accuracy: 0.0%
Epsilon: 0.04000000000000001, Robust Accuracy: 0.0%
Epsilon: 0.05000000000000001, Robust Accuracy: 0.0%
Epsilon: 0.06000000000000001, Robust Accuracy: 0.0%
Epsilon: 0.07, Robust Accuracy: 0.0%
Epsilon: 0.08, Robust Accuracy: 0.0%
Epsilon: 0.09000000000000001, Robust Accuracy: 0.0%
Epsilon: 0.1, Robust Accuracy: 0.0%
```



1.1 Problem 3: L_∞ Networks (Lp Nets)

We discussed L_∞ networks in class, as described in [ZCL+21]. In this problem, you are tasked with formally deriving a general expression or a precise lower bound for the L_∞ **certified radius** of an L_∞ **network** for any given input $\mathbf{x}$ and weight $\mathbf{w}$, using the properties of the network.

1.1.1 Setup

An L_∞ network is constructed using L_∞-dist neurons, where each neuron computes the operation:

$$u(\mathbf{x}, \theta) = \|\mathbf{x} - \mathbf{w}\|_\infty + b,$$

with parameters $\theta = \{\mathbf{w}, b\}$. The network output is defined as:

$$g(\mathbf{x}) = (-x_1^{(L)}, -x_2^{(L)}, \dots, -x_M^{(L)}),$$

where $x_i^{(L)}$ is the output of the last layer for class i, and M is the number of classes. The classifier is then:

$$f(\mathbf{x}) = \arg\max_{i\in[M]} g_i(\mathbf{x}).$$

L_∞ networks are 1-Lipschitz with respect to the L_∞ norm:

$$\|g(\mathbf{x_1}) - g(\mathbf{x_2})\|_\infty \leq \|\mathbf{x_1} - \mathbf{x_2}\|_\infty, \quad \forall \mathbf{x_1}, \mathbf{x_2} \in \mathbb{R}^d.$$

1.1.2 Certified Radius Derivation

We want a general expression or a precise lower bound for the certified radius R_{cert} of the L_∞ network, given an input $\mathbf{x}$ and weights $\mathbf{w}$. The certified radius is the minimum perturbation required to change the classifier's prediction:

$$R_{\text{cert}}(f; \mathbf{x}, y) = \inf_{\mathbf{x}': f(\mathbf{x}') \neq f(\mathbf{x})} \|\mathbf{x}' - \mathbf{x}\|_\infty,$$

1.1.3 Step 1: Define the Margin

Let's define the margin at input $\mathbf{x}$ as the difference between the logit for the true class and the largest logit for any other class:

$$\text{margin}(\mathbf{x}; g) = g_y(\mathbf{x}) - \max_{i\neq y} g_i(\mathbf{x}).$$

1.1.4 Step 2: Utilize the Lipschitz Property

Since the network g is 1-Lipschitz with respect to the L_∞ norm, any perturbation δ satisfies:

$$\|g(\mathbf{x} + \delta) - g(\mathbf{x})\|_\infty \leq \|\delta\|_\infty.$$

Meaning that each component of the output can change by at most $\|\delta\|_\infty$:

$$|g_i(\mathbf{x} + \delta) - g_i(\mathbf{x})| \leq \|\delta\|_\infty, \quad \forall i \in [M].$$

1.1.5 Step 3: Find the Minimum Perturbation to Change the Prediction

To change the prediction from class y to another class $k \neq y$, the following condition must be met:

$$g_y(\mathbf{x} + \delta) \leq g_k(\mathbf{x} + \delta).$$

Using the Lipschitz property, we get:

$$g_y(\mathbf{x} + \delta) \geq g_y(\mathbf{x}) - \|\delta\|_\infty,$$

$$g_k(\mathbf{x} + \delta) \leq g_k(\mathbf{x}) + \|\delta\|_\infty.$$

Therefore, for the prediction to change, it suffices that:

$$g_y(\mathbf{x}) - \|\delta\|_\infty \leq g_k(\mathbf{x}) + \|\delta\|_\infty.$$

Simplifying, we get:

$$g_y(\mathbf{x}) - g_k(\mathbf{x}) \leq 2\|\delta\|_\infty.$$

1.1.6 Step 4: Derive the Certified Radius

The minimum $\|\delta\|_\infty$ required to change the prediction to any other class is therefore at least half the margin:

$$\|\delta\|_\infty \geq \frac{1}{2}\left(g_y(\mathbf{x}) - \max_{i\neq y} g_i(\mathbf{x})\right).$$

Thus, the certified radius is given by:

$$R_{\text{cert}}(f;\mathbf{x},y) \geq \frac{\text{margin}(\mathbf{x};g)}{2}.$$