Deep Learning Assignment 2

Junyou Su 12110911

November 9, 2024

1 Instructions on how to run the code

File Structure

- **Assignment_2_CNN&RNN/**
  - **Part 1/**
    - data/ *(Folder containing datasets)*
    - **data_config.py** *(Configuration for dataset paths and settings)*
    - **mlp_numpy.py** *(MLP implementation using NumPy)*
    - **modules.py** *(Various modules for MLP model)*
    - **p1.ipynb** *(Jupyter notebook for Part 1)*
    - **pytorch_mlp.py** *(MLP model implemented in PyTorch)*
    - **pytorch_train_mlp_CIFAR10.py** *(Script to train MLP on CIFAR-10 dataset)*
    - **pytorch_train_mlp.py** *(General MLP training script in PyTorch)*
    - **train_mlp_numpy.py** *(Training script for NumPy-based MLP)*
  - **Part 2/**
    - data/ *(Folder containing datasets)*
    - **cnn_model.py** *(CNN model implemented in PyTorch)*
    - **cnn_train.py** *(Training script for the CNN model)*
    - **p2.ipynb** *(Jupyter notebook for Part 2)*
  - **Part 3/**
    - **dataset.py** *(Dataset class definition for Part 3)*
    - **p3.ipynb** *(Jupyter notebook for Part 3)*
    - **train.py** *(Training script for RNN model)*
    - **utils.py** *(Utility functions for Part 3)*
    - **vanilla_rnn.py** *(Implementation of RNN in PyTorch)*
  - **CS324__Deep_Learning_Assignment_2.pdf**
  - **report.pdf**

1.1 How to Run

All instructions for running the code can be found in the Jupyter notebooks: `p1.ipynb`, `p2.ipynb`, and `p3.ipynb`.

2 The Structure of Models

2.1 Part 1

2.1.1 Task 1 & Task 2

- **Model Architecture:**

  - **Layer 1:**
    - **Type:** Linear
    - **Input Features:** 2
    - **Output Features:** 20
    - **Activation:** ReLU
  - **Layer 2:**
    - **Type:** Linear
    - **Input Features:** 20
    - **Output Features:** 2
    - **Activation:** None (Output Layer)

2.1.2 Task 3

- **Model Architecture:**
  - **Layer 1:**
    - **Type:** Linear
    - **Input Features:** 3072
    - **Output Features:** 1024
    - **Activation:** ReLU
  - **Layer 2:**
    - **Type:** Linear
    - **Input Features:** 1024
    - **Output Features:** 512
    - **Activation:** ReLU
  - **Layer 3:**
    - **Type:** Linear
    - **Input Features:** 512
    - **Output Features:** 256
    - **Activation:** ReLU
  - **Layer 4:**
    - **Type:** Linear
    - **Input Features:** 256
    - **Output Features:** 256
    - **Activation:** ReLU
  - **Layer 5:**
    - **Type:** Linear
    - **Input Features:** 256
    - **Output Features:** 128
    - **Activation:** ReLU
  - **Layer 6:**
    - **Type:** Linear
    - **Input Features:** 128
    - **Output Features:** 10
    - **Activation:** None (Output Layer)

2.2 Part 2

The model architecture and training setup for this task are as follows:

- **Model Architecture:** Convolutional Neural Network (CNN)
  - **Input:** Image Datasets.
  - **Convolutional Layers:**
    - **Block 1:**
      - **Convolution:** 64 filters, 3x3
      - **Batch Normalization:** Yes
      - **Activation:** ReLU
      - **Max Pooling:** 3x3, stride 2, padding 1
    - **Block 2:**
      - **Convolution:** 128 filters, 3x3
      - **Batch Normalization:** Yes
      - **Activation:** ReLU
      - **Max Pooling:** 3x3, stride 2, padding 1
    - **Block 3:**
      - **Convolution 1:** 256 filters, 3x3
      - **Batch Normalization:** Yes
      - **Activation:** ReLU
      - **Convolution 2:** 256 filters, 3x3
      - **Batch Normalization:** Yes
      - **Activation:** ReLU
      - **Max Pooling:** 3x3, stride 2, padding 1
    - **Block 4:**
      - **Convolution 1:** 512 filters, 3x3
      - **Batch Normalization:** Yes
      - **Activation:** ReLU
      - **Convolution 2:** 512 filters, 3x3
      - **Batch Normalization:** Yes
      - **Activation:** ReLU
      - **Max Pooling:** 3x3, stride 2, padding 1
    - **Block 5:**
      - **Convolution 1:** 512 filters, 3x3
      - **Batch Normalization:** Yes
      - **Activation:** ReLU
      - **Convolution 2:** 512 filters, 3x3
      - **Batch Normalization:** Yes
      - **Activation:** ReLU
      - **Max Pooling:** 3x3, stride 2, padding 1
  - **Fully Connected Layer:** Linear layer with 512 input features and `n_classes` output units for classification.

2.3 Part 3

- **Model Architecture:** Vanilla Recurrent Neural Network (RNN)
  - **Input:** Sequence data with shape `(batch_size, input_length, input_dim)`.
  - **RNN Layers:**

        - **Layer:**
            - **Type:** Linear
            - **Input Features:** input_dim + hidden_dim
            - **Output Features:** hidden_dim
            - **Activation:** Tanh
    - **Output Layer:**
        - **Type:** Linear
        - **Input Features:** hidden_dim
        - **Output Features:** output_dim
        - **Activation:** None (Output Layer)
    - **Processing:**
        - **Hidden State Initialization:** Zero vector with shape `(batch_size, hidden_dim)`
        - **Time Steps:** Iterates through each time step in the input sequence, updating the hidden state using a combination of the current input and previous hidden state.

3 Procedure of Training

3.1 Setup

- **Model:** Ubuntu 22.04 Server
- **GPU:** 8 × NVIDIA GeForce RTX 4090
- **CPU:** AMD EPYC 7542 32-Core Processor
    - **Cores:** 64 cores per socket, 2 sockets (128 threads total)
    - **Frequency:** 1.5 GHz (base) to 2.9 GHz (max)
    - **Caches:** L1: 2 MiB, L2: 32 MiB, L3: 256 MiB
    - **NUMA Nodes:** 2 (node0: CPUs 0-31, 64-95; node1: CPUs 32-63, 96-127)

3.2 Part 1

3.2.1 Task 1 & Task 2

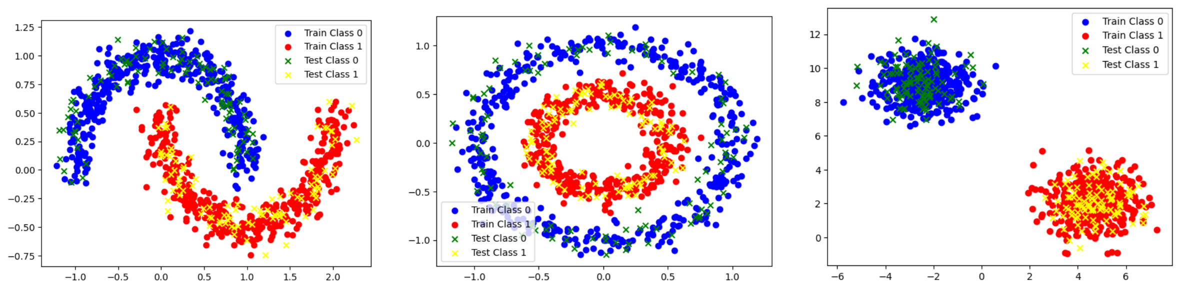

Figure 1: **The Visualization of Moon, Circle, Blob Dataset**

- **Parameters:**
    - learning_rate = 0.01
    - max_steps = 1500
    - batch_size = 30
    - hidden_layer = ”20”

  - eval_freq = 30
- **Dataset: Moon, Circle, Blob (show in Figure 1)**
- **Process:**
  - Input shape: `torch.Size([30(Batch Size), 2(Input Dim)])`
  - After layer 1 (Linear), shape: `torch.Size([30(Batch Size), 20(Hidden Layer1)])`
  - After layer 2 (Linear), shape: `torch.Size([30(Batch Size), 2(Number of Classes)])`
- **Optimization:**
  - Using mini-batch gradient descent
  - Optimizer: SGD
  - Loss function: CrossEntropy

3.2.2 Task 3

- **Parameters:**
  - learning_rate = 0.001
  - max_steps = 50
  - batch_size = 256
  - hidden_layers = "1024, 512, 256, 256, 128"
  - eval_freq = 1
- **Dataset: CIFAR10**
- **Process:**
  - Input shape: `torch.Size([256(Batch Size), 3072(Input Dim)])`
  - After layer 1 (Linear), shape: `torch.Size([256(Batch Size), 1024(Hidden Layer1)])`
  - After layer 2 (Linear), shape: `torch.Size([256(Batch Size), 512(Hidden Layer2)])`
  - After layer 3 (Linear), shape: `torch.Size([256(Batch Size), 256(Hidden Layer3)])`
  - After layer 4 (Linear), shape: `torch.Size([256(Batch Size), 256(Hidden Layer4)])`
  - After layer 5 (Linear), shape: `torch.Size([256(Batch Size), 128(Hidden Layer5)])`
  - After layer 6 (Linear), shape: `torch.Size([256(Batch Size), 10(Number of Classes)])`
- **Optimization:**
  - Using mini-batch gradient descent
  - Optimizer: AdamW
  - Loss function: CrossEntropy

3.3 Part 2

- **Parameters:**
  - learning_rate = 0.0001
  - max_steps = 30
  - batch_size = 256
  - eval_freq = 1
- **Dataset: CIFAR10**
- **Process:**

  - Input shape: `torch.Size([32(Batch Size), 3(Channels), 32(Height), 32(Width)])`
  - After first block: `torch.Size([32(Batch Size), 64, 16, 16])`
  - After second block: `torch.Size([32(Batch Size), 128, 8, 8])`
  - After third block: `torch.Size([32(Batch Size), 256, 4, 4])`
  - After fourth block: `torch.Size([32(Batch Size), 512, 2, 2])`
  - After fifth block: `torch.Size([32(Batch Size), 512, 1, 1])`
  - After flatten: `torch.Size([32(Batch Size), 512])`
  - After fully connected layer: `torch.Size([32(Batch Size), 10(Number of Classes)])`

- **Optimization:**
  - Using mini-batch gradient descent
  - Optimizer: AdamW
  - Loss function: CrossEntropy

3.4 Part 3

- **Parameters:**
  - input_length = 15
  - input_dim = 10
  - num_classes = 10
  - num_hidden = 128
  - batch_size = 128
  - learning_rate = 0.001
  - max_epoch = 30
  - max_norm = 10.0
  - data_size = 100000
  - portion_train = 0.8

- **Process:**
  - Input shape: `torch.Size([128(Batch Size), 10(input_dim)])`
  - Concatenate with `h` of shape `torch.Size([128(Batch Size), 128(Num_hidden)])` to form `torch.Size([128(Batch Size), 138(input_dim + Num_hidden)])`
  - Pass through a fully connected layer (Linear(138(input_dim + Num_hidden), 128(Num_hidden))), resulting in `torch.Size([128(Batch Size), 128(input_dim + Num_hidden)])`
  - Pass through a final fully connected layer (Linear(128(Num_hidden), 10(output_dim))), resulting in `torch.Size([128(Batch Size), 10(output_dim)])`

- **Optimization:**
  - Using mini-batch gradient descent
  - Optimizer: RMSprop
  - lr_scheduler: StepLR
  - Loss function: CrossEntropy

4 Analysis of Results

4.1 Part 1

4.1.1 Task 1 & Task 2

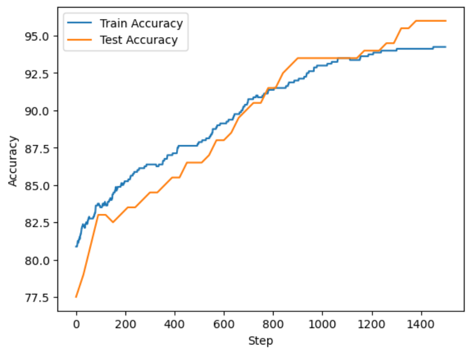

Figure 2: **Training and Testing Accuracy of MLP on Moon Dataset (NumPy Implementation)**

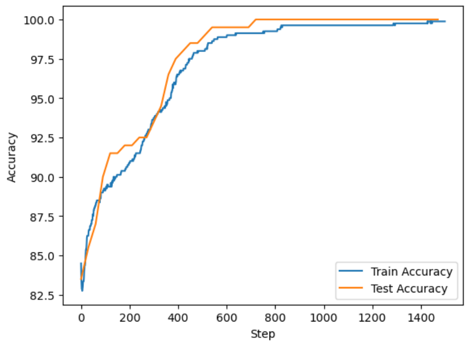

Figure 3: **Training and Testing Accuracy of MLP on Moon Dataset (PyTorch Implementation)**

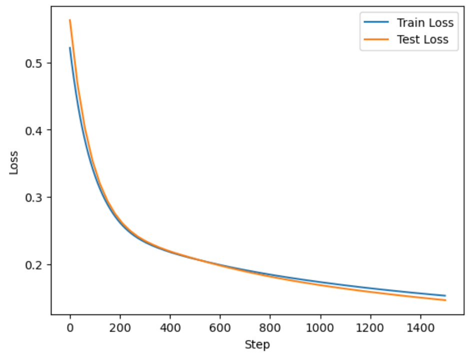

Figure 4: **Training and Testing Loss of MLP on Moon Dataset (NumPy Implementation)**

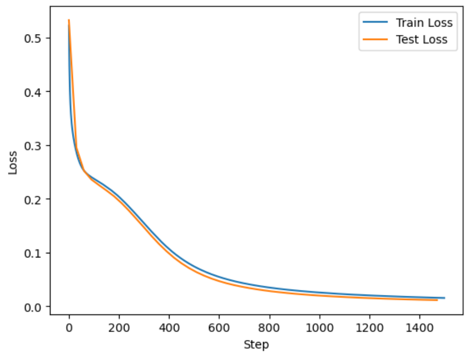

Figure 5: **Training and Testing Loss of MLP on Moon Dataset (PyTorch Implementation)**

Analysis

After completing the experiment where both the NumPy and PyTorch versions of the MLP architecture were implemented and trained on the same dataset, the following conclusions were drawn:

Accuracy

Both versions of the MLP model (NumPy and PyTorch) achieved similar accuracy on both the training and test datasets (see the Circle and Blob Accuracy and Loss Curves in Appendix A). This consistency suggests that the MLP architecture is reproducible across different frameworks.

Training Speed

- In some cases, the PyTorch implementation was slower than the NumPy version.
- This could be attributed to the complexity of PyTorch, which includes additional processes for managing computation graphs, device allocation, and multi-GPU optimization. While these features are highly beneficial for large-scale tasks, they introduce overhead in smaller tasks.

- In contrast, NumPy is a lightweight library that directly operates on arrays, which might explain its faster training speeds for smaller, simpler tasks.

Training Curves

The training curves revealed differences between the two implementations, possibly due to variations in random seed generation or initialization. These differences can impact the convergence rates, causing the models to learn and optimize at different speeds. Despite these variations, both models ultimately converged to similar performance levels. Both NumPy and PyTorch are effective tools for implementing MLP architectures. The choice between the two depends on the problem size and required features. For quick experimentation with smaller datasets, NumPy may be more efficient, while PyTorch is better suited for large-scale training tasks, thanks to its advanced features like automatic differentiation, GPU acceleration, and distributed computing.

4.1.2 Task 3

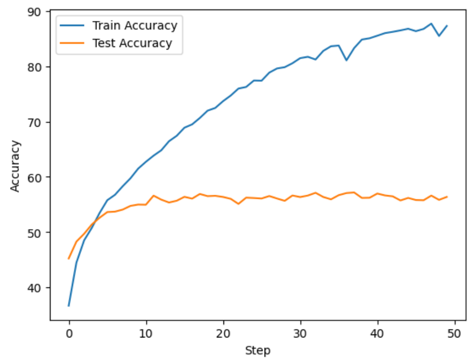

Figure 6: **Training and Testing Accuracy of MLP on CIFAR-10 Dataset**

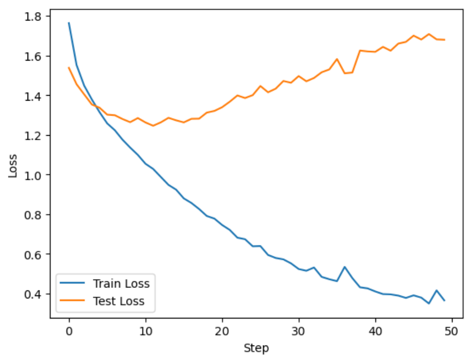

Figure 7: **Training and Testing Loss of MLP on CIFAR-10 Dataset**

Analysis

However, from the training curves Fig 6, it is evident that learning within the first 5 epochs is particularly effective. Beyond this point, the loss starts to increase, indicating severe overfitting. So the pure MLP architecture cannot effectively capture the spatial characteristics of image data, leading to lower overall accuracy compared to convolutional neural networks.

Improve

To mitigate overfitting and improve accuracy, we incorporated dropout and batch normalization into the model. Dropout helps prevent the model from relying too heavily on specific neurons, which enhances generalization, while batch normalization stabilizes the training process by normalizing inputs across layers. These techniques led to more robust learning outcomes and helped counteract the model's tendency to overfit. Additionally, we employed multi-GPU parallelism to accelerate training, allowing the model to process larger batches more efficiently and reduce the overall training time.

4.2 Part 2

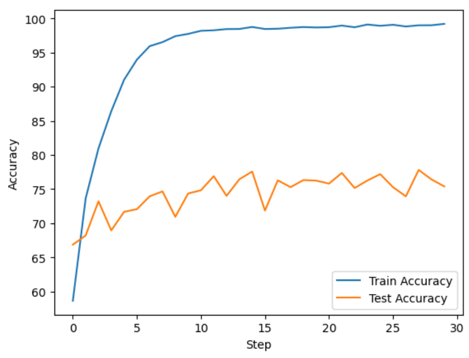

Figure 8: **Training and Testing Accuracy of CNN on CIFAR-10 Dataset**

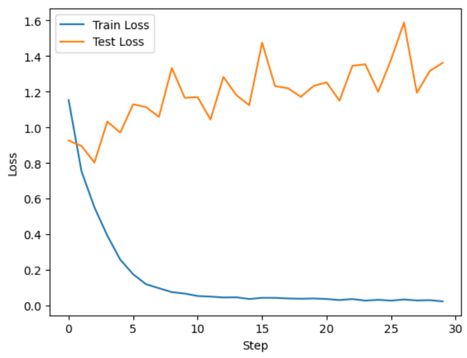

Figure 9: **Training and Testing Loss of CNN on CIFAR-10 Dataset**

Analysis

From the training curves in Fig 8 and Fig 9, we observe that the CNN model learns effectively during the initial epochs, with a rapid increase in accuracy and a sharp decrease in loss within the first 10 steps. However, while the training accuracy continues to improve and approaches nearly 100%, the test accuracy plateaus around 75%, and test loss fluctuates, indicating some degree of overfitting. Despite this, CNNs demonstrate a clear advantage over traditional multi-layer perceptrons (MLPs) in handling image data.
Unlike MLPs, which treat each pixel independently, CNNs leverage convolutional layers to capture spatial hierarchies in the data, preserving local structure and reducing the number of parameters. This ability to recognize patterns and edges in images makes CNNs particularly well-suited for image classification tasks, where spatial relationships are crucial. As a result, CNNs achieve significantly higher accuracy than MLPs in image-based tasks like CIFAR-10, as seen in these results.

4.3 Part 3

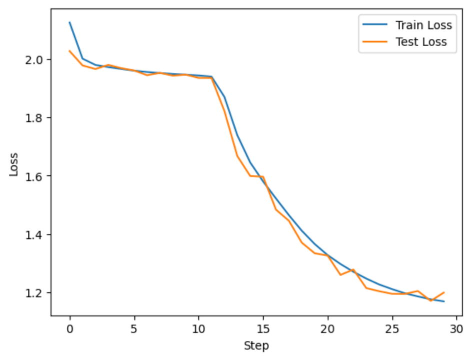

Figure 10: **Training and Testing Loss of RNN Model (T=19)**

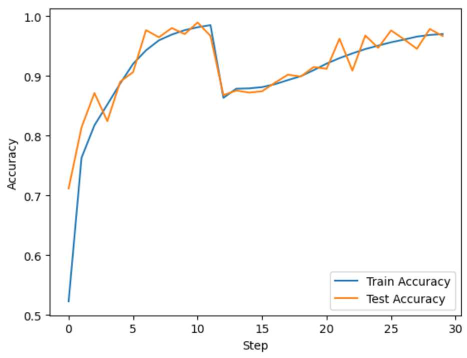

Figure 11: **Training and Testing Accuracy of RNN Model (T=19)**

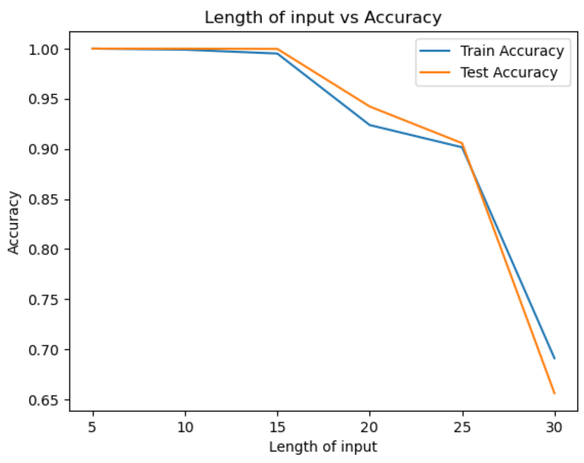

Figure 12: **Sequence Length vs. Accuracy of RNN Model**

Detail Accuracy and Loss Curve see in Appendix B

Analysis

In Fig 10, we observe the training and testing loss curves for the RNN model, showing a consistent decrease in loss as training progresses, with both train and test loss aligning closely. This close alignment indicates that the RNN model is generalizing well without significant overfitting. The loss stabilizes at around step 25, suggesting that the model has converged.
Fig 11presents the training and testing accuracy curves. Both curves demonstrate a sharp increase in accuracy in the initial steps, reaching around 90% by step 10. The training and test accuracies remain closely aligned, further supporting that the model generalizes effectively on the test set. The high and consistent performance of the RNN indicates it has effectively learned temporal patterns, which are critical for sequential data.
In Fig 12, we analyze the impact of sequence length on accuracy. For shorter sequences (up to 15 steps), both training and testing accuracy are close to 100%, suggesting the model performs exceptionally well with shorter input lengths. However, as the sequence length increases beyond 15, accuracy begins to drop, with a sharp decline at lengths 25 and 30. This decline indicates that the model struggles to capture long-range dependencies, which is a known limitation of standard RNNs. See detail Accuracy and Loss Curves in Appendix B

A Other Results of Part 1

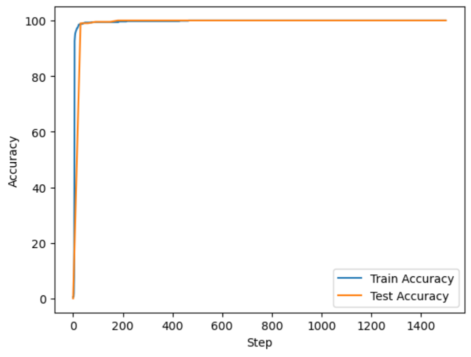

Figure 13: **Training and Testing Accuracy of MLP on Blob Dataset (NumPy Implementation)**

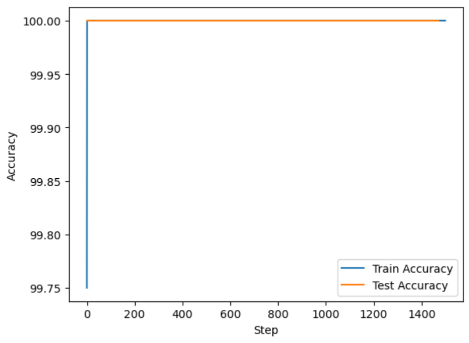

Figure 14: **Training and Testing Accuracy of MLP on Blob Dataset (PyTorch Implementation)**

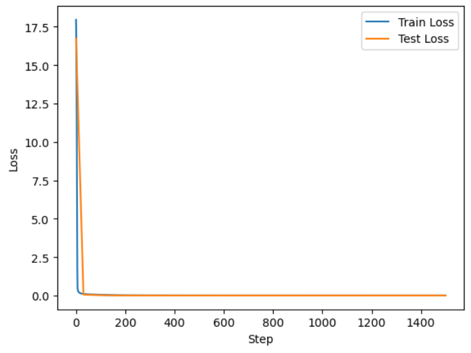

Figure 15: **Training and Testing Loss of MLP on Blob Dataset (NumPy Implementation)**

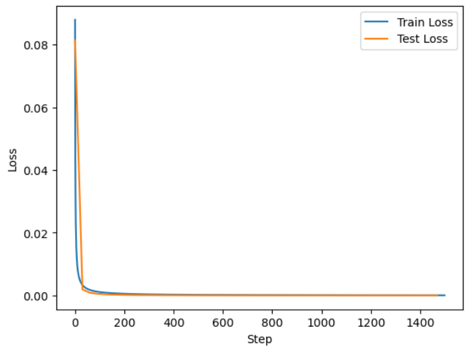

Figure 16: **Training and Testing Loss of MLP on Blob Dataset (PyTorch Implementation)**

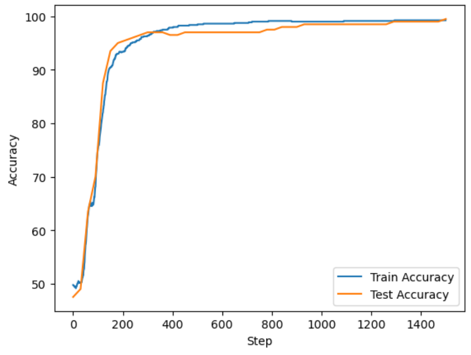

Figure 17: **Training and Testing Accuracy of MLP on Circle Dataset (NumPy Implementation)**

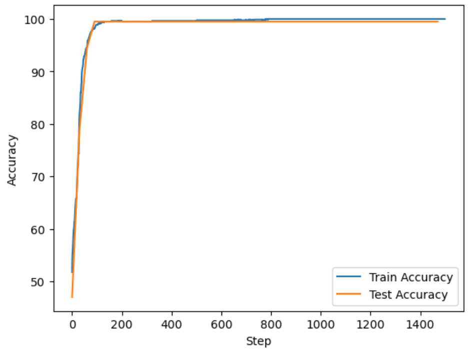

Figure 18: **Training and Testing Accuracy of MLP on Circle Dataset (PyTorch Implementation)**

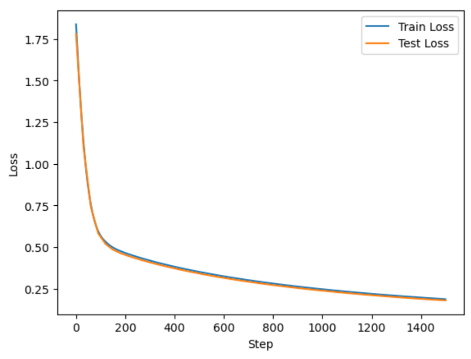

Figure 19: **Training and Testing Loss of MLP on Circle Dataset (NumPy Implementation)**

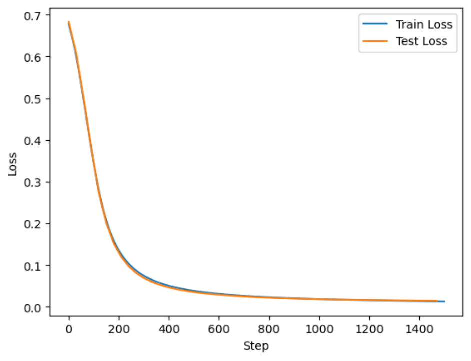

Figure 20: **Training and Testing Loss of MLP on Circle Dataset (PyTorch Implementation)**

B Detail Results of Part 3

B.1 T = 5

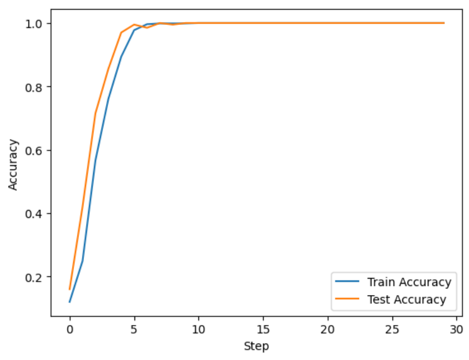

Figure 21: **Training and Testing Accuracy of RNN Model (T=5)**

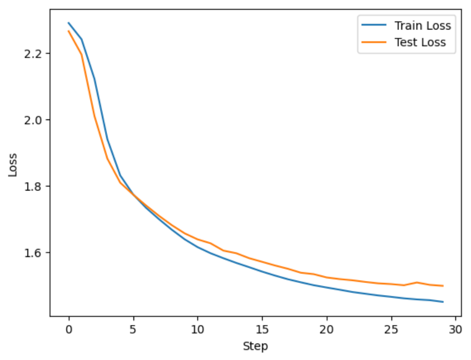

Figure 22: **Training and Testing Loss of RNN Model (T=5)**

B.2 T = 10

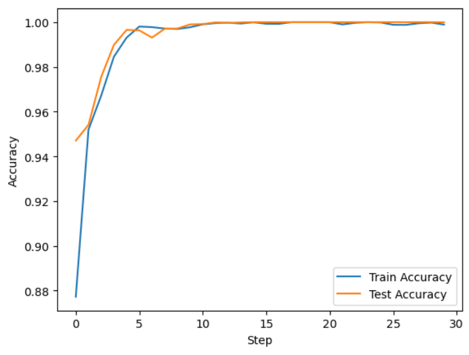

Figure 23: **Training and Testing Accuracy of RNN Model (T=10)**

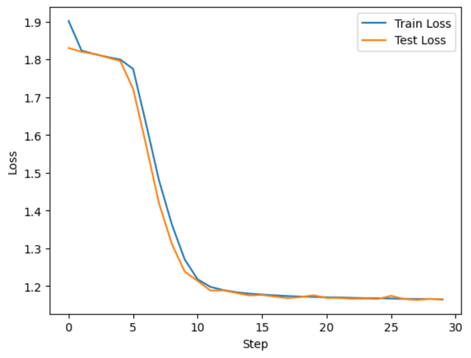

Figure 24: **Training and Testing Loss of RNN Model (T=10)**

B.3 T = 15

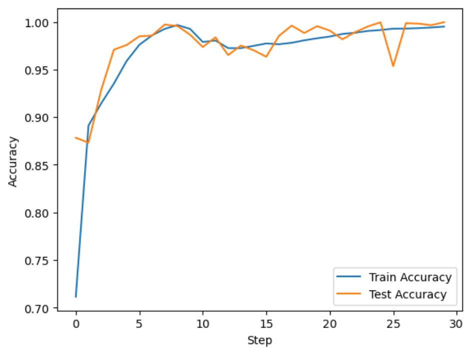

Figure 25: **Training and Testing Accuracy of RNN Model (T=15)**

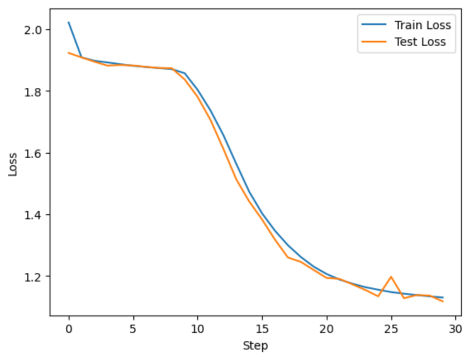

Figure 26: **Training and Testing Loss of RNN Model (T=15)**

B.4 T = 20

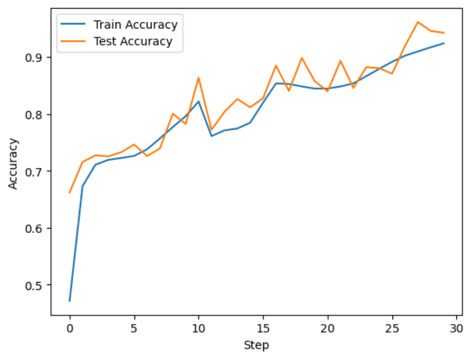

Figure 27: **Training and Testing Accuracy of RNN Model (T=20)**

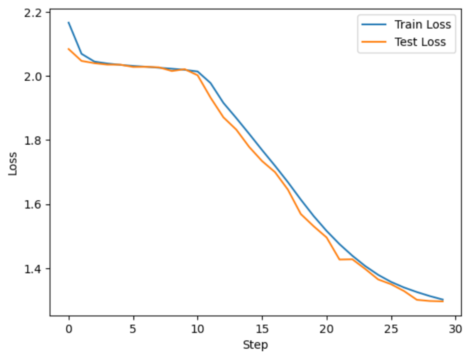

Figure 28: **Training and Testing Loss of RNN Model (T=20)**

B.5 T = 25

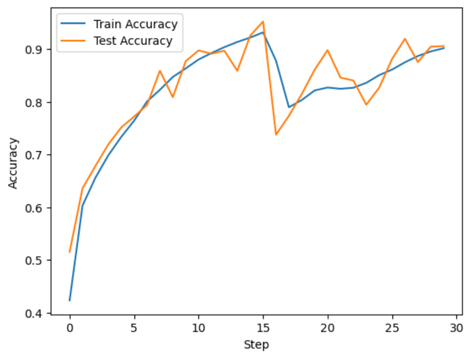

Figure 29: **Training and Testing Accuracy of RNN Model (T=25)**

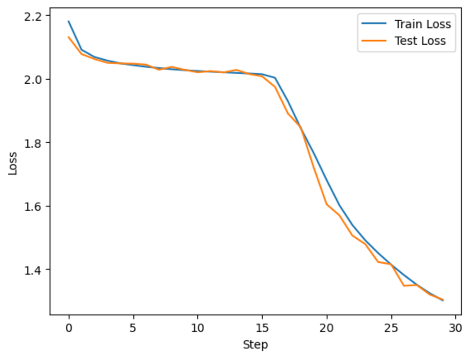

Figure 30: **Training and Testing Loss of RNN Model (T=25)**

B.6 T = 30

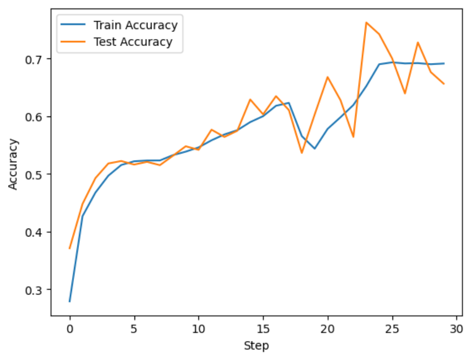

Figure 31: **Training and Testing Accuracy of RNN Model (T=30)**

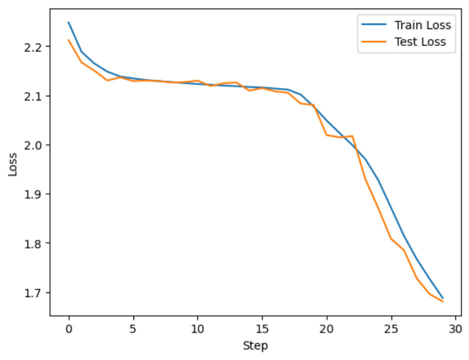

Figure 32: **Training and Testing Loss of RNN Model (T=30)**