CS324 · Deep Learning

# Assignment 2

THA SREYNY / 12113053

November 28, 2024

## 1 Introduction

The purpose of this assignment is to implement a Multi-Layer Perceptron (MLP) architecture using PyTorch and compare its performance with a previously implemented NumPy version. The report documents the implementation steps, experimental setup, and results for the following tasks:

- Task 1: Implement the MLP architecture and training procedure in PyTorch.
- Task 2: Train and compare the NumPy and PyTorch MLP implementations on the same dataset.
- Task 3: Train a PyTorch MLP on the CIFAR-10 dataset to achieve the highest possible accuracy.

## 2 Part I: PyTorch MLP

### 2.1 Task 1: PyTorch MLP Implementation

#### 2.1.1 Implementation

The MLP architecture was implemented as a subclass of `torch.nn.Module`, and the training procedure was implemented using PyTorch's training utilities. The architecture consists of the following components:

- Input layer matching the feature dimensions of the dataset.
- One or more hidden layers with ReLU activation functions.
- Output layer with softmax activation for classification.

#### 2.1.2 MLP Implementation (`numpy_mlp.py`)

```
class MLPN(object):
    def __init__(self, n_inputs, n_hidden, n_classes):

        self.layers = []
        self.layers.append(Linear(n_inputs, n_hidden[0]))
        self.layers.append(ReLU())
        for index in range(1, len(n_hidden)):
            self.layers.append(Linear(n_inputs[index - 1], n_hidden[index]))
            self.layers.append(ReLU())
        self.layers.append(Linear(n_hidden[-1], n_classes))
        self.layers.append(SoftMax())

        self.loss_function = CrossEntropy()

    def forward(self, x):
        for y in self.layers:
            out = y.forward(out)

        return out

    def backward(self, dout):
        for y in reversed(self.layers):
            dout = y.backward(dout)
```

#### 2.1.3 MLP Implementation (pytorch_mlp.py)

```
class MLP(nn.Module):
    def __init__(self, n_inputs, n_hidden, n_classes):
        self.model = nn.Sequential(
            nn.Linear(n_inputs, n_hidden[0]),
            nn.ReLU(),
            nn.Linear(n_hidden[0], n_classes)
        )

    def forward(self, x):
        out = self.model(x)
        return out
```

#### 2.1.4 Training Procedure (pytorch_train_mlp.py)

The training loop includes:

- Forward pass.
- Loss computation using `torch.nn.CrossEntropyLoss`.
- Backpropagation and optimization using `torch.optim.Adam`.

## 2.2 Task 2: Training and Comparison of NumPy and PyTorch MLPs

### 2.2.1 Experimental Setup

We used the `make_moons` dataset from Scikit-learn to generate training and testing data. Both MLP implementations (NumPy and PyTorch) were trained on the same dataset using the same hyperparameters:

- Learning rate: 0.01
- Number of epochs: 1000
- Batch size: 1000

### 2.2.2 Results

The performance of the two implementations was compared using classification accuracy. Both implementations achieved similar accuracy rates. A visualization of the training result for both models.

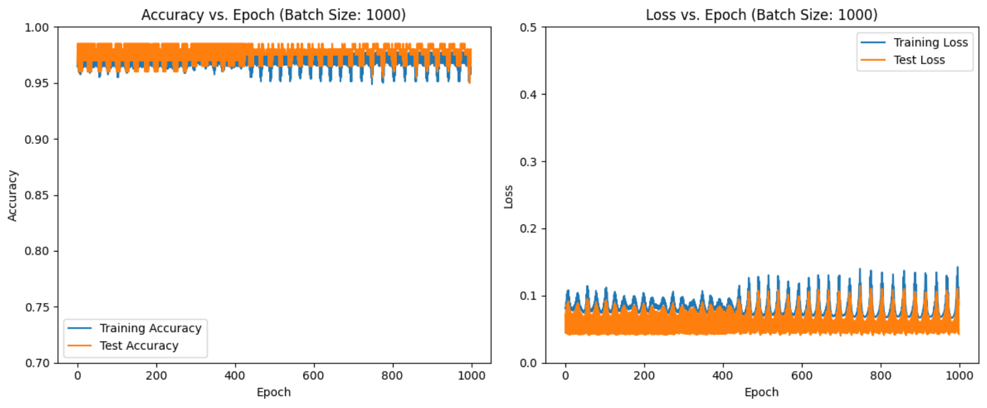

Figure 1: Training and validation loss curves for the CIFAR-10 MLP by numpy.

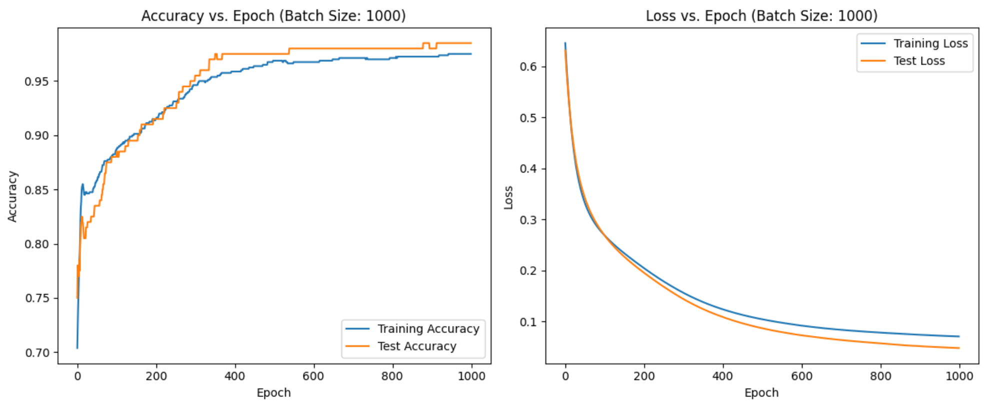

Figure 2: Training and validation loss curves for the CIFAR-10 MLP by pytorch.

### 2.2.3 Conclusion

The PyTorch implementation of the MLP outperforms the NumPy version in terms of speed and stability. PyTorch converges faster, with smoother accuracy and loss curves, thanks to optimized features like GPU support, advanced optimizers (e.g., Adam), and automatic differentiation. In contrast, the NumPy implementation shows slower convergence and more erratic training behavior due to the lack of such optimizations.

### 2.2.4 Pytorch model with other datasets

**Datasets**

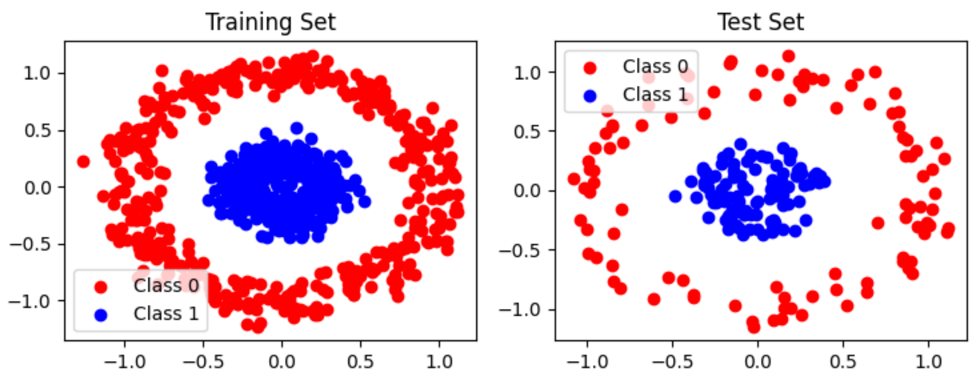

Figure 3: Dataset of make circles.

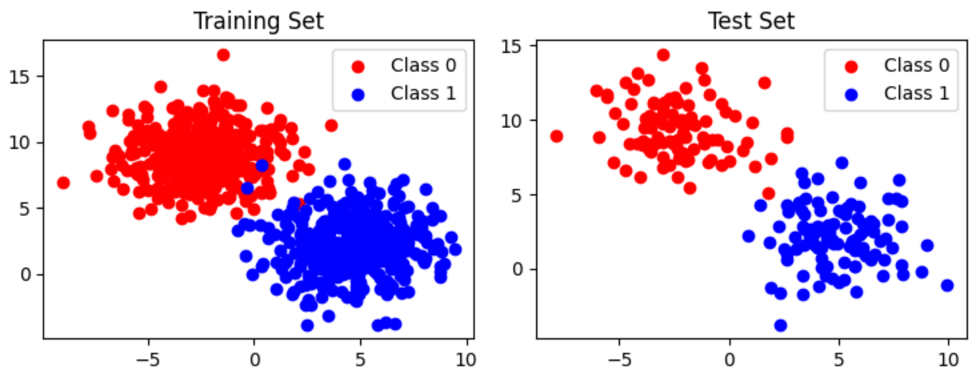

Figure 4: Dataset of make blobs

### 2.2.5 Results

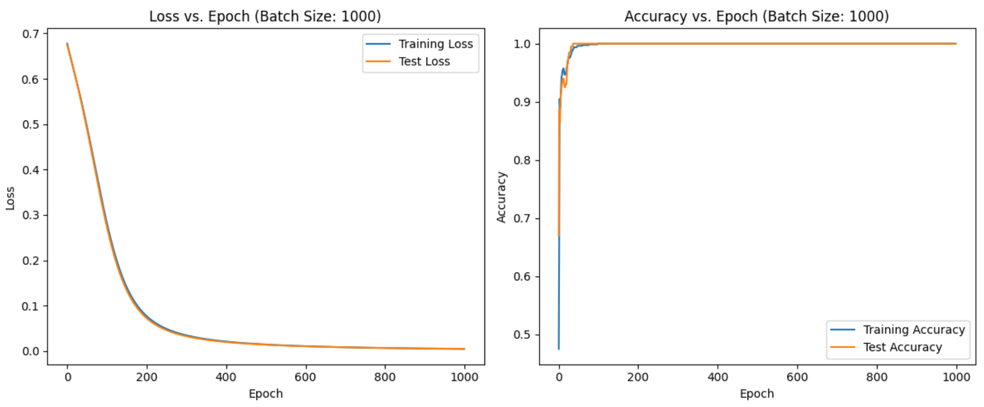

Figure 5: Training and validation loss curves for the make circles MLP by pytorch

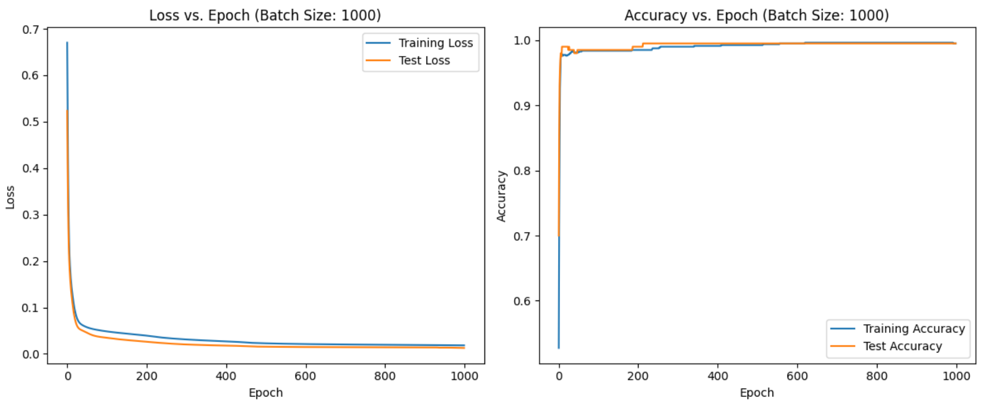

Figure 6: Training and validation loss curves for the make blobs MLP by pytorch

### 2.2.6 Conclusion

The implementation and evaluation of the Multi-Layer Perceptron (MLP) on the `make_moon`, `make_circle`, and `make_blobs` datasets demonstrate its versatility and effectiveness in handling diverse classification problems.

- **Make Moon Dataset:** The MLP successfully learns the non-linear decision boundaries required for this dataset, showcasing its ability to model complex patterns with curved separations.
- **Make Circle Dataset:** The model effectively classifies concentric circular regions by leveraging its hidden layers to capture the radial symmetry of the data.
- **Make Blobs Dataset:** On this dataset, the MLP efficiently separates clusters, highlighting its suitability for problems involving well-defined clusters with linear or slightly non-linear boundaries.

Overall, the MLP proves to be a robust and adaptable architecture capable of achieving high accuracy across datasets with varying levels of complexity and separability. This performance came from careful tuning of hyperparameters such as the number of hidden layers, neurons, learning rate, and regularization techniques. These findings underline the importance of understanding the data's characteristics to obtain an optimal MLP model.

### 2.3 Task 3: PyTorch MLP on CIFAR-10 Dataset

#### 2.3.1 Objective

Train a PyTorch MLP on the CIFAR-10 dataset to achieve the highest possible accuracy using appropriate architectures, optimization, and regularization techniques.

#### 2.3.2 Implementation

- Dataset loading: `torchvision.datasets.CIFAR10` was used to load the training and test sets.
- Preprocessing: Images were normalized and converted to tensors.
- MLP architecture: A deeper network with dropout layers was used to prevent overfitting.

#### 2.3.3 Results

Training and validation of Accuracy and loss curves are shown in Figure 7.

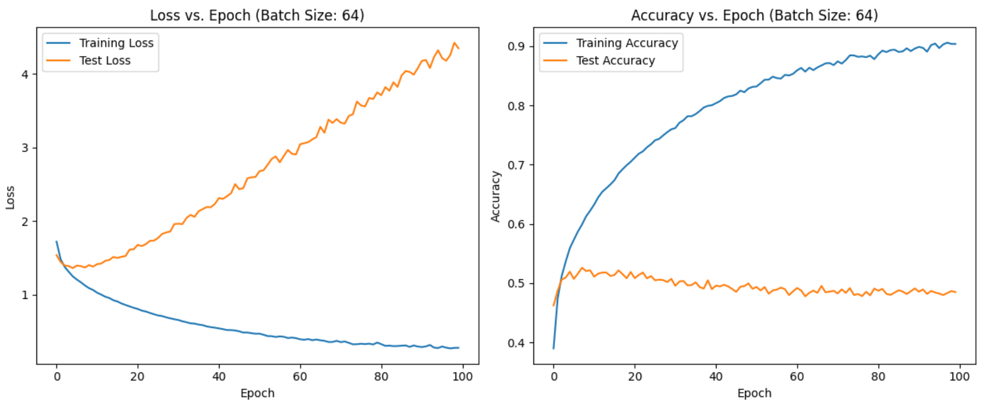

Figure 7: Training and validation of Accuracy and loss curves for the CIFAR-10 MLP using pytorch.

#### 2.3.4 Conclusion

The low performance of Multi-Layer Perceptrons (MLPs) on the CIFAR-10 dataset is primarily due to several key factors. CIFAR-10 images, being 32x32 pixels, present a high-dimensional input space that MLPs struggle to navigate effectively, as they lack the ability to exploit spatial hierarchies inherent in images. Unlike Convolutional Neural Networks (CNNs), which can capture local patterns and relationships, MLPs treat each input feature independently, leading to challenges in distinguishing the diverse classes present in the dataset. Additionally, MLPs are prone to overfitting, particularly when they have high capacity and are not properly regularized. Their shallow architecture compared to deeper networks also limits their ability to learn complex features, while issues related to optimization, such as vanishing gradients, can hinder effective training.

## 3 Part II: PyTorch CNN

### 3.1 Introduction to Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are specialized for handling image data. These networks take images as input and classify them into one of several possible output categories. A CNN is structured with a series of layers, where each layer processes the data before passing it to the next. The input image traverses through all the layers to be accurately assigned to a specific category. Compared to standard neural networks, CNNs can identify complex features within images that might otherwise go unnoticed.

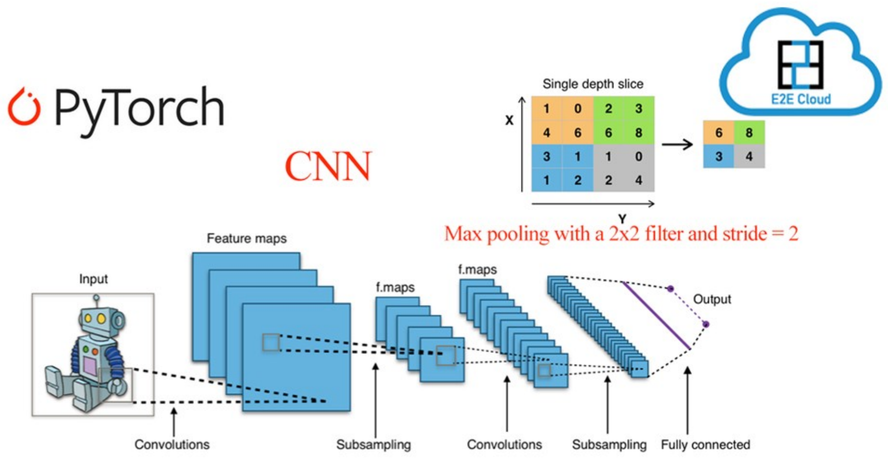

Figure 8: CNN architecture. Source: [1]

The design of a typical neural network can be summarized as follows:

1. The model is initialized with an optimizer, and a corresponding loss function is defined.
2. The network comprises layers, which are trained to establish the desired functionality within the model.
3. The training process runs for a set number of iterations (epochs), during which training and validation losses are recorded.
4. Over time, the validation loss decreases as the model adjusts and improves its performance.
5. After training, the results are plotted to provide a visual representation of the training and validation performance, enabling an evaluation of their consistency.
6. The model is validated by measuring its accuracy on the dataset. Achieving an accuracy above 70% is generally considered a strong outcome.

## 3.2 Task 1: CNN Implementation

### 3.2.1 Objective

The goal of this task is to implement a CNN architecture and training procedure using PyTorch by completing the files `cnn_model.py` and `cnn_train.py`. The architecture follows a reduced version of the VGG network.

The CNN architecture consists of:

- Convolutional layers with ReLU activation and max pooling.
- Fully connected layers with dropout for regularization.
- Output layer for classification.

The model was trained using both Adam and SGD optimizers with the following hyperparameters:

- Learning rate: 0.001 (Adam) and 0.01 (SGD).
- Number of epochs: 100.
- Batch size: 64.

#### 3.2.2 Implementation

```
class CNN(nn.Module):

  def __init__(self, n_channels, n_classes):

    super(CNN, self).__init__()
    self.model = nn.Sequential(
      nn.Conv2d(in_channels=n_channels, out_channels=32, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(32),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),

      nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(64),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),

      nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(128),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),

      nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(256),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),

      nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(512),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2, stride=2),


      nn.Flatten(),
      nn.Linear(512, 128),
      nn.ReLU(),
      nn.Linear(128, n_classes)
    )

  def forward(self, x):
    out = self.model(x)
    return out
```

## 3.3 Task 2: Performance Analysis

### 3.3.1 Accuracy and Loss Curves

The training and validation accuracy and loss curves for both optimizers are shown in Figures 9 and 10.

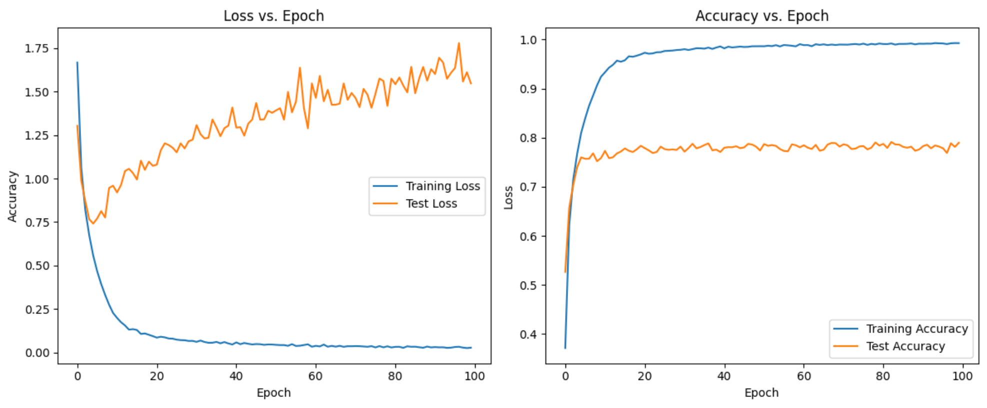

Figure 9: Training and validation curves for the CNN using the Adam optimizer.

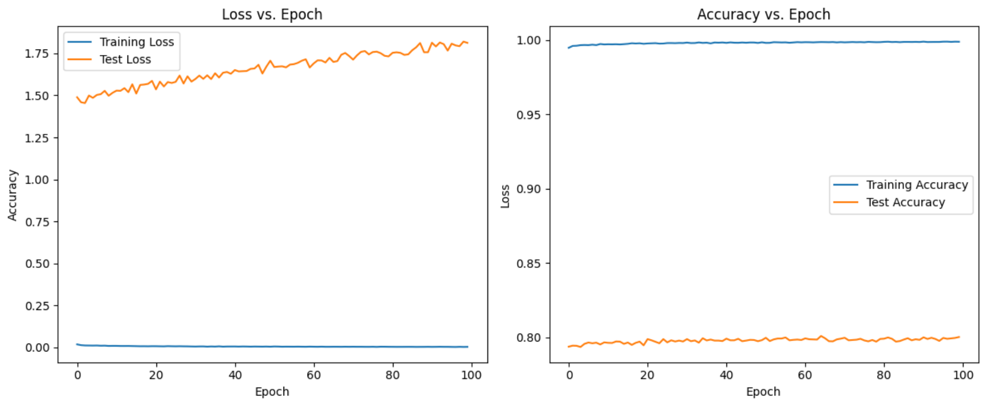

Figure 10: Training and validation curves for the CNN using the SGD optimizer.

### 3.3.2 Comparison of Optimizers

- **Adam**: Faster convergence with smoother loss curves but prone to overfitting.
- **SGD**: Slower convergence but better generalization on the validation set.

## 4 Part III PyTorch RNN

Recurrent Neural Networks (RNNs) are designed to address the limitations of traditional neural networks, which process inputs and outputs independently. For tasks such as predicting the next word in a sentence, understanding context from previous inputs is essential. RNNs resolve this issue by introducing a feedback mechanism, where the output of one step is used as input for the next.

This architecture enables RNNs to maintain context through a *hidden state*, also referred to as the *memory state*, which retains important information from prior steps in a sequence. By reusing the same parameters across all steps, RNNs ensure consistency and reduce model complexity compared to traditional neural networks. These features make RNNs particularly effective for tasks involving sequential data.

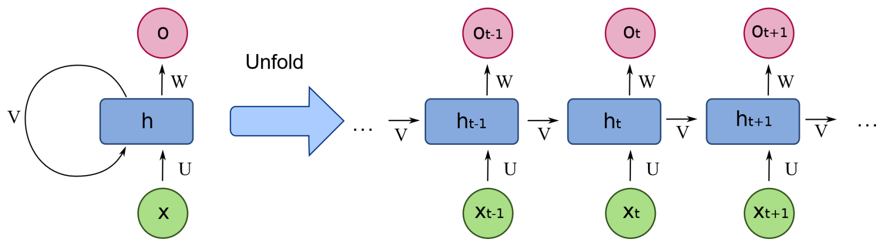

Figure 11: RNN architecture. Source: [2]

In this part, we implemented a Recurrent Neural Network (RNN) to predict the last digit of a palindrome. Palindromes are sequences that remain the same when read forward and backward (e.g., 123454321, 1111, 4224). The provided dataset class, `PalindromeDataset`, generates palindrome numbers for training and testing the model. The RNN was designed following the structure defined by the equations:

$$h^{(t)} = \tanh(W_{hh}h^{(t-1)} + W_{hx}x^{(t)} + b_h), \tag{1}$$

$$o^{(t)} = (W_{ph}h^{(t)} + b_o), \tag{2}$$

$$\tilde{y}^{(t)} = \text{softmax}(o^{(t)}). \tag{3}$$

Here, the hidden state $h^{(0)}$ is initialized to a vector of zeros. The task focuses on predicting the last digit of a palindrome number, optimizing the cross-entropy loss only at the final time step $T$:

$$\mathcal{L} = -\sum_{k=1}^{K} y_k \log(\tilde{y}_k^{(T)}), \tag{4}$$

where $K = 10$ represents the 10 possible digit classes, and $y_k$ is a one-hot encoding vector.

## 4.1 Task 1: Manual RNN Implementation

The RNN was implemented without using `torch.nn.RNN` or `torch.nn.LSTM`. Instead, a custom RNN was developed in the file `vanilla_rnn.py`, using the skeleton provided in `train.py`.

### Forward Pass

For the forward pass, a `for` loop was used to compute the hidden states $h^{(t)}$ and output logits $o^{(t)}$ for each time step $t$. The equations (1), (2), and (3) were implemented explicitly using matrix multiplications and activation functions.

### Backward Pass

The backward pass leveraged PyTorch's automatic differentiation to compute gradients. The RMSProp optimizer was used to update the weights and biases, ensuring stable training.

## 4.2 Task 2: Performance Analysis

Using the custom RNN from Task 1, the network was trained to predict the $T$-th digit of palindromes given the first $T - 1$ digits. The performance was evaluated for varying palindrome lengths $T$.

## Results

A Jupyter Notebook was used to plot the prediction accuracy against the palindrome length. The accuracy was observed to decrease as the palindrome length increased, demonstrating the RNN's limitations in retaining information over long sequences. With $T = 5$, near-perfect accuracy was achieved, but performance degraded for $T > 10$ due to the limited memory of the RNN.

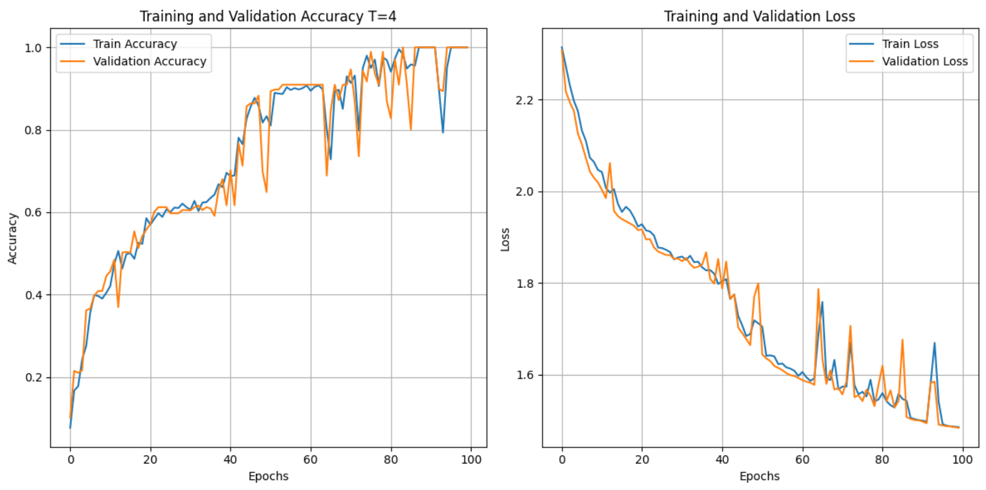

Figure 12: Training and validation curves for the RNN with T=4

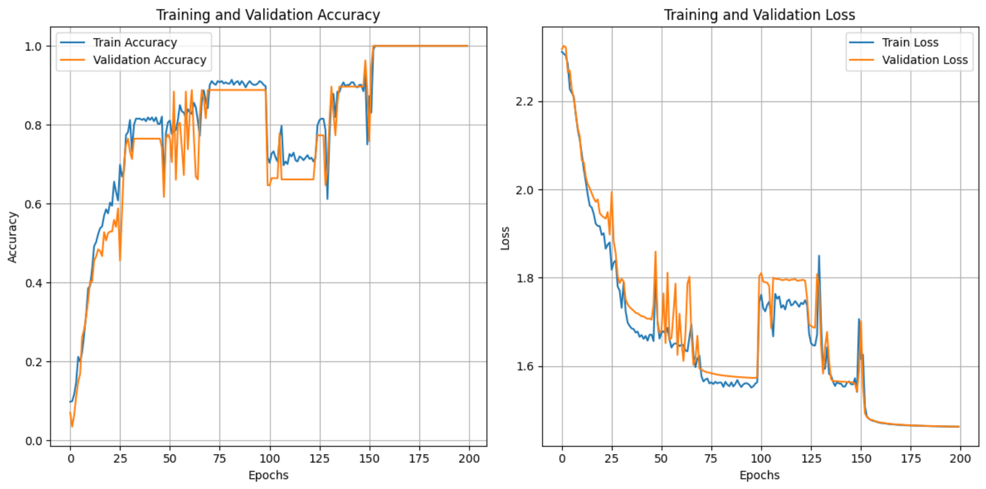

Figure 13: Training and validation curves for the RNN with T=5

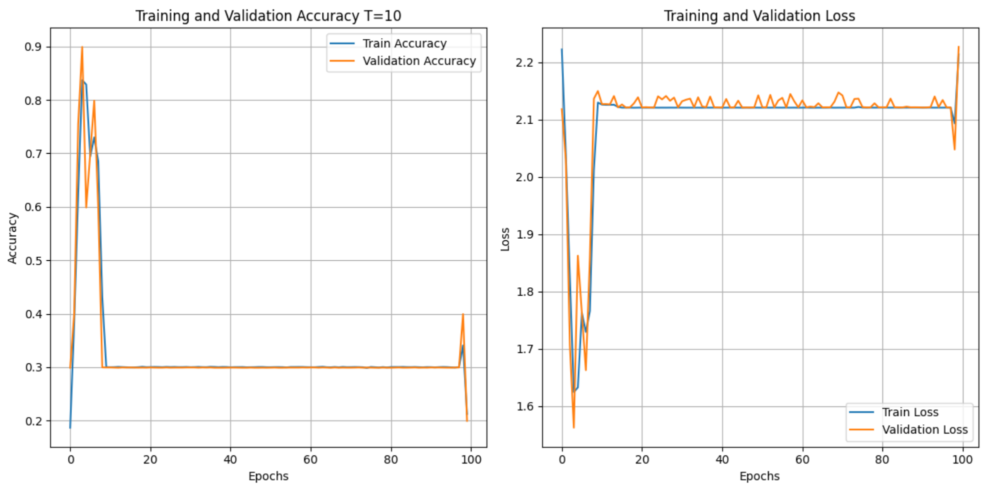

Figure 14: Training and validation curves for the RNN with T=10

## Discussion

These results align with the theoretical limitations of RNNs discussed in the lecture. As sequence length increases, the vanishing gradient problem hampers the network's ability to learn long-term dependencies. Future work could involve using architectures like LSTMs or GANs to address these issues.

## References

[1] Shivanii Raina, *Implementing CNN in PyTorch with Custom Dataset and Transfer Learning*. October 4, 2022 Available:`https://www.linkedin.com/pulse/implementing-cnn-pytorch-custom-dataset-transfer-learning-raina`.

[2] Understanding Pytorch vanilla RNN architectures. Available:`https://stackoverflow.com/questions/57122727/understanding-pytorch-vanilla-rnn-architectures`.