SUSTech CS324 Deep Learning
Report for Assignment 2

Student ID: 12312710

Name: 胥熠/Brighton

Date: 2025 年 11 月 22 日

目录

1 Part 1: PyTorch MLP

1.1 Task 1: Implementing MLP

In this task, I implemented a Multi-Layer Perceptron using PyTorch. The structure of which is the same as the NumPy Version I implemented in the Assignment 1.

1.1.1 Implementation

Unlike the NumPy version where parameters (W, b) and gradients (dW, db) are manually managed in dictionaries, PyTorch leverages the `nn.Module` class and the Autograd engine. The Autograd automatically tracks operations on tensors to compute gradients during backpropagation, which is more efficient and less error-prone.

- **Model Structure:** I used `nn.ModuleList` to dynamically store linear layers based on the input hidden unit list. At MLP ___init___ function, each layer will be appended to the ModuleList with proper dimensions.
- **Forward Pass:** In the forward method, I use a for loop to iterate through each layer in the ModuleList, applying ReLU activation after each hidden layer. The final layer outputs raw scores (logits) without activation.
- **Loss Function:** I utilized `nn.CrossEntropyLoss`, which combines `LogSoftmax` and `NLLLoss` in one single class, to compute the loss between predicted logits and true labels.
- **Optimization:** Instead of manual weight updates ($w = w - \eta \cdot dw$), I utilized `torch.optim.SGD`. Also, since I implemented momentum in the NumPy version, PyTorch's built-in momentum support in SGD was used by setting the momentum parameter.

1.1.2 Model Architecture

Both the NumPy and PyTorch implementations share the same architecture and hyperparameters:

- Input Layer: 2 neurons (for 2D input data)
- Hidden Layer 1: Two hidden layers with 20 neurons each, using ReLU activation. (where $W_1 \in [2, 20]$, $b_1 \in [20]$)
- Hidden Layer 2: Two hidden layers with 20 neurons each, using ReLU activation. (where $W_2 \in [20, 20]$, $b_2 \in [20]$)
- Output Layer: 2 neurons (for binary classification)

Where the hyperparameters are: Learning Rate: 1e-2, Momentum: 0.9, Epochs: 300, Batch Size: 64.

1.2 Task 2: Comparison with NumPy Implementation

To compare, I used three different datasets: Moons, Circles, and Blobs. For each dataset, I trained both the NumPy and PyTorch implementations of the MLP with identical hyperparameters and recorded their performance, and here are the results:

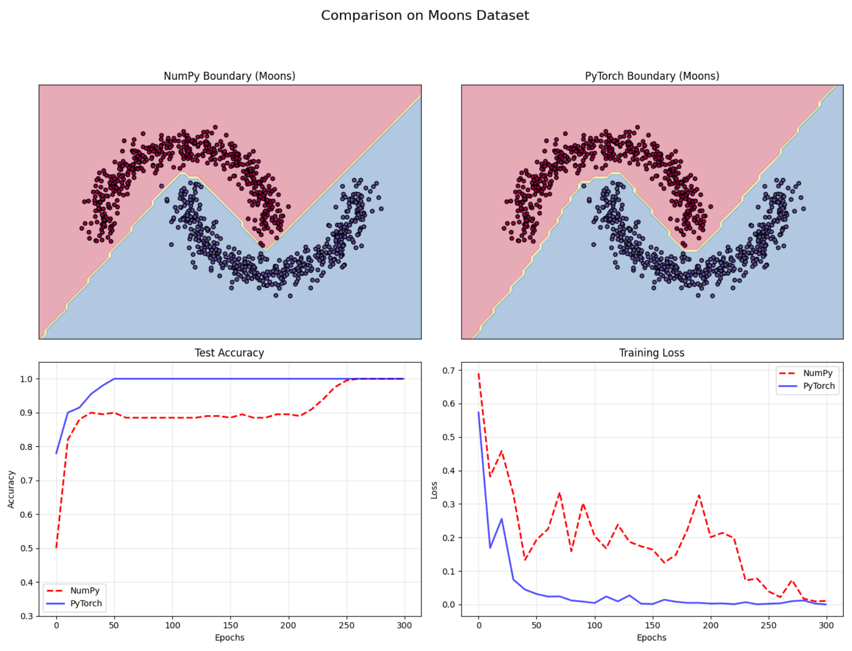


图 1: Comparison on Moons Dataset

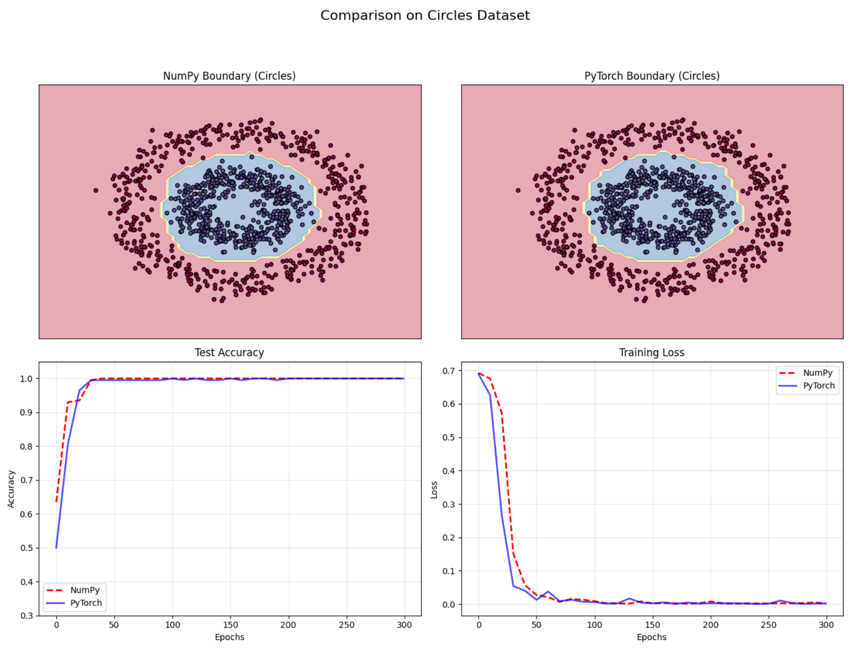


图 2: Comparison on Circles Dataset

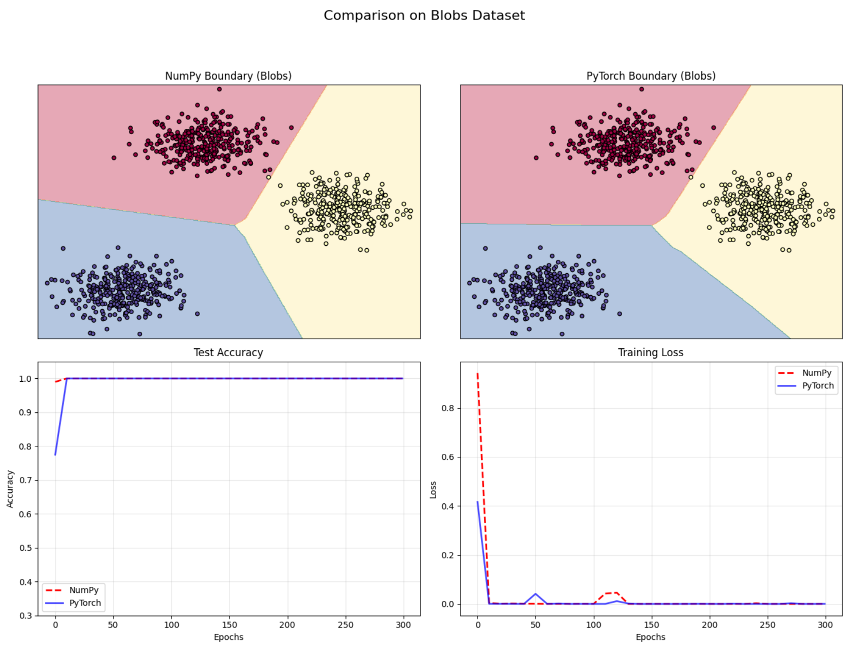


图 3: Comparison on Blobs Dataset

Actually, the results of both implementations are quite similar in terms of decision boundaries, accuracy trends, and loss curves across all datasets. PyTorch converges slightly faster but both reach comparable final accuracies.

1.2.1 PyTorch MLP in CIFAR-10 Dataset

CIFAR-10 is a more complex dataset compared to the synthetic datasets used earlier. I trained the PyTorch MLP on CIFAR-10 with the following architecture:

- Input Layer: 3072 neurons (32x32x3 flattened images)
- Hidden Layer 1: 512 neurons with ReLU activation
- Hidden Layer 2: 256 neurons with ReLU activation
- Hidden Layer 3: 128 neurons with ReLU activation
- Output Layer: 10 neurons (for 10 classes)

Using the same hyperparameters (Learning Rate: 1e-3, Adam optimizer, 20 Epochs).
Here are the detailed dimension analysis of the MLP architecture on CIFAR-10:

表 1: MLP Architecture and Dimension Analysis (CIFAR-10)

Layer Type	**Input Shape**	**Weight Matrix Shape**	**Output Shape**	**Activation**	**Parameters**
Input Flatten	$(B, 3, 32, 32)$	-	$(B, 3072)$	-	0
Hidden Layer 1	$(B, 3072)$	(3072×512)	$(B, 512)$	ReLU	1,573,376
Hidden Layer 2	$(B, 512)$	(512×256)	$(B, 256)$	ReLU	131,328
Hidden Layer 3	$(B, 256)$	(256×128)	$(B, 128)$	ReLU	32,896
Output Layer	$(B, 128)$	(128×10)	$(B, 10)$	-	1,290
				Total Parameters:	**1,738,890**

Here are the results:

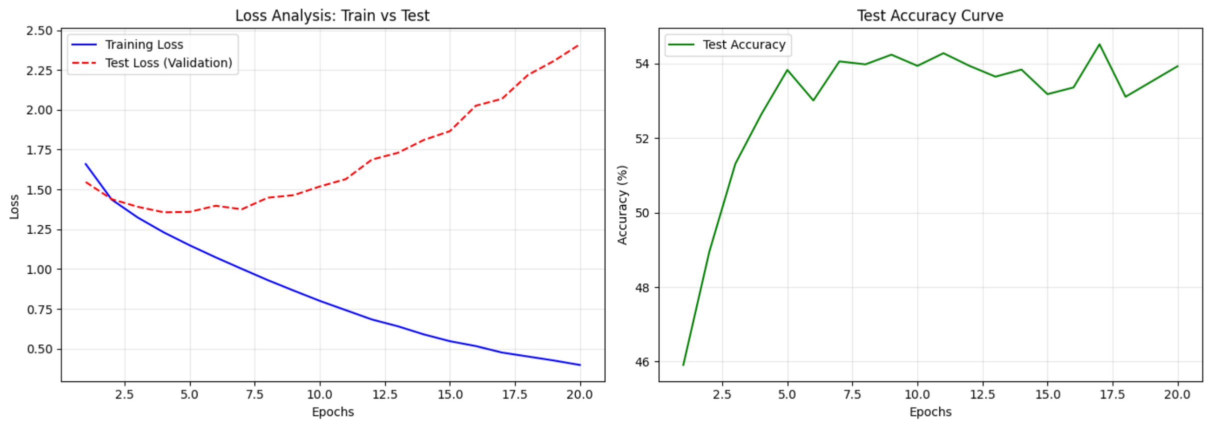


图 4: PyTorch MLP on CIFAR-10 Dataset

We can observe a new pattern that: Although the loss of train datasets decreases steadily, the test loss starts to increase at about the 8th round.

Finally, the PyTorch MLP achieved a test accuracy of around **54%** on CIFAR-10. This is a classic example of **Overfitting**, where the model learns the training data too well but loses generalization.

This indicates that MLPs, due to their fully connected nature, struggle to capture the spatial hierarchies in image data compared to architectures like CNNs.

2 Part II: PyTorch CNN

2.1 Task 1: CNN Architecture & Dimension Analysis

In this part, I implemented a reduced version of the VGG network. The network consists of three convolutional blocks followed by a fully connected classifier.

2.1.1 Detailed Model Structure

The structure, as required, follows the same as the slides give, which is a reduced VGG architecture:

- **1. conv layer**: k=3x3, s=1, p=1, in=3, out=64
- **2. maxpool**: k=3x3, s=2, p=1, in=64, out=64
- **3. conv layer**: k=3x3, s=1, p=1, in=64, out=128
- **4. maxpool**: k=3x3, s=2, p=1, in=128, out=128
- **5. conv layer**: k=3x3, s=1, p=1, in=128, out=256
- **6. conv layer**: k=3x3, s=1, p=1, in=256, out=256
- **7. maxpool**: k=3x3, s=2, p=1, in=256, out=256
- **8. conv layer**: k=3x3, s=1, p=1, in=256, out=512
- **9. conv layer**: k=3x3, s=1, p=1, in=512, out=512
- **10. maxpool**: k=3x3, s=2, p=1, in=512, out=512
- **11. conv layer**: k=3x3, s=1, p=1, in=512, out=512
- **12. conv layer**: k=3x3, s=1, p=1, in=512, out=512
- **13. maxpool**: k=3x3, s=2, p=1, in=512, out=512
- **14. linear**: in=512, out=10

表 2: Architecture and Dimension Flow of the Reduced VGG

ID	Layer Type	Kernel / Stride / Pad	Input Shape	Output Shape	Parameters
1	Conv2d	$3 \times 3/1/1$	$(3, 32, 32)$	$(64, 32, 32)$	1,792
2	MaxPool	$3 \times 3/2/1$	$(64, 32, 32)$	$(64, 16, 16)$	0
3	Conv2d	$3 \times 3/1/1$	$(64, 16, 16)$	$(128, 16, 16)$	73,856
4	MaxPool	$3 \times 3/2/1$	$(128, 16, 16)$	$(128, 8, 8)$	0
5	Conv2d	$3 \times 3/1/1$	$(128, 8, 8)$	$(256, 8, 8)$	295,168
6	Conv2d	$3 \times 3/1/1$	$(256, 8, 8)$	$(256, 8, 8)$	590,080
7	MaxPool	$3 \times 3/2/1$	$(256, 8, 8)$	$(256, 4, 4)$	0
8	Conv2d	$3 \times 3/1/1$	$(256, 4, 4)$	$(512, 4, 4)$	1,180,160
9	Conv2d	$3 \times 3/1/1$	$(512, 4, 4)$	$(512, 4, 4)$	2,359,808
10	MaxPool	$3 \times 3/2/1$	$(512, 4, 4)$	$(512, 2, 2)$	0
11	Conv2d	$3 \times 3/1/1$	$(512, 2, 2)$	$(512, 2, 2)$	2,359,808
12	Conv2d	$3 \times 3/1/1$	$(512, 2, 2)$	$(512, 2, 2)$	2,359,808
13	MaxPool	$3 \times 3/2/1$	$(512, 2, 2)$	$(512, 1, 1)$	0
-	**Flatten**	-	$(512, 1, 1)$	(**512**)	0
14	Linear	-	(512)	(10)	5,130

2.2 Task 2: Results Analysis & Optimization

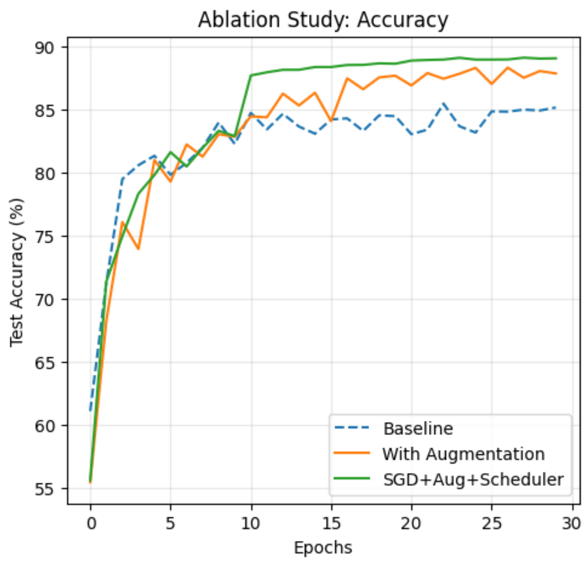
Following the above structure, at the end of training, the reduced VGG CNN achieved a test accuracy of around **84%** on CIFAR-10, which is a substantial improvement over the MLP's performance. However, there is a noticeable gap in training speed compared to the MLP, likely due to the increased model complexity and number of parameters.

I use **(pure reduced VGG with Adam, 84%)** as **Baseline**.

However, **84%** cannot satisfy the primary need, so we can consider some optimization methods as: Data Augmentation & Learning Rate Scheduling.

Although CNN is translation invariant, it is not rotation or scale invariant. Therefore, adding random rotations, crops, and flips to the training images can help the model generalize better. Also, at the beginning of training, a higher learning rate allows for faster convergence. As training progresses, reducing the learning rate helps fine-tune the weights and avoid overshooting minima.

What's better, the PyTorch itself provides convenient APIs to implement these two techniques. And here are the results:



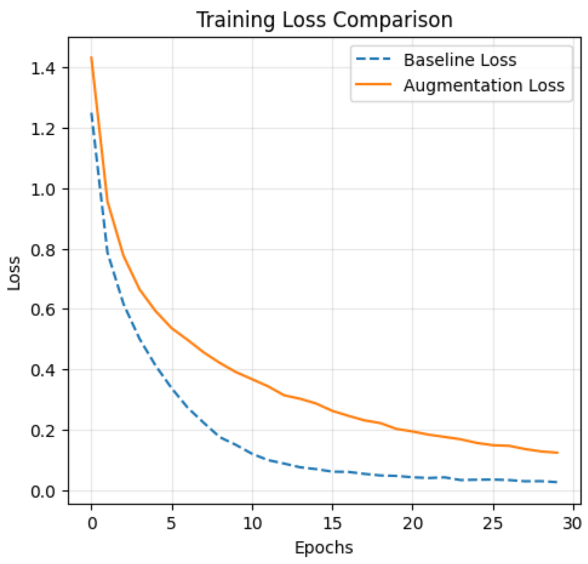


图 5: Different Reduced VGG on CIFAR-10 Dataset

As we can see from the figure above, both techniques can help improving the final accuracy. The Augmentation mainly helps reduce overfitting, while the Learning Rate Scheduler helps achieve a lower final loss.

By adding the Augmentation, we can observe that the training loss is higher than baseline, but performs better at accuracy, which powerfully illustrates its ability to prevent overfitting.

The scheduler brings a smoother accuracy curve, and the highest accuracy reaches **89%** when combining both techniques, which is a significant boost from the original **84%**.

3 Part III: PyTorch RNN

3.1 Task 1: Vanilla RNN Implementation

In this part, I implemented a RNN to solve the Palindrome Prediction task. The network follows the recurrence relation:

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

$$o^{(t)} = W_{ph}h^{(t)} + b_o$$

$$\hat{y}^{(t)} = \text{Softmax}(o^{(t)})$$

Where $h^{(t)}$ is the hidden state at time step t, $x^{(t)}$ is the input at time step t, and $o^{(t)}$ is the output logits.

In my implementations, I used 128 neurons in the hidden layer(for better memorization), and the input size is 10 (digits 0-9). The weight matrices and biases are as follows:

表 3: RNN Matrix Dimensions and Parameter Count per Time Step

Operation / Layer	**Input Variable**	**Weight Shape**	**Output Variable**	**Output Shape**
Input Transform	$x^{(t)} \in \mathbb{R}^{B\times 10}$	$W_{hx} \in \mathbb{R}^{10\times 128}$	$A \in \mathbb{R}^{B\times 128}$	$(128, 128)$
Hidden Recurrence	$h^{(t-1)} \in \mathbb{R}^{B\times 128}$	$W_{hh} \in \mathbb{R}^{128\times 128}$	$B \in \mathbb{R}^{B\times 128}$	$(128, 128)$
Activation	$A + B + b_h$	-	$h^{(t)} \in \mathbb{R}^{B\times 128}$	$(128, 128)$
Output Projection	$h^{(t)} \in \mathbb{R}^{B\times 128}$	$W_{ph} \in \mathbb{R}^{128\times 10}$	$o^{(t)} \in \mathbb{R}^{B\times 10}$	$(128, 10)$

In my implementation, the training strategy is given as:

- **Optimizer:** RMSProp (Learning Rate = 0.001).
- **Gradient Clipping:** Applied `clip_grad_norm_(max_norm=10)` to dW. Since W_{hh} is multiplied recursively T times, clipping is essential to prevent the 'Exploding Gradient' phenomenon where gradients tend to infinity.
- **One-Hot Encoding:** Since we are going to predict digits (0-9), and every digit is independent with **NO** impact of their value, so one-hot encoding is very appealing here, since it provides Orthogonality between different digits.

3.2 Task 2: Accuracy vs. Palindrome Length

I trained the RNN on palindromes of varying lengths $T \in \{3, 7, 11, 13, 15, 17, 19, 21, 25\}$. The relationship between sequence length and final accuracy is shown in Figure below:

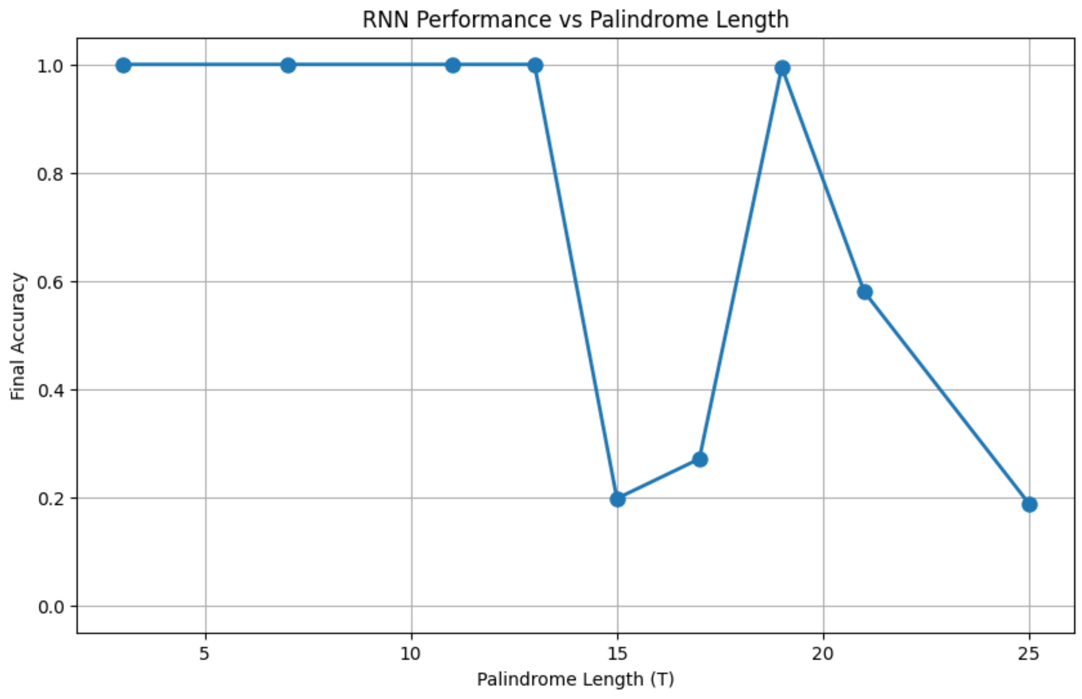


图 6: RNN Accuracy vs. Palindrome Length

Well, This very first result is very weird, it collapses at $T = 15$, however, it bounces back at $T = 19$. This is because of the 'lucky' initialization. So I re-run the experiments 5 times each and take the average, and finally get a more reasonable result as below:

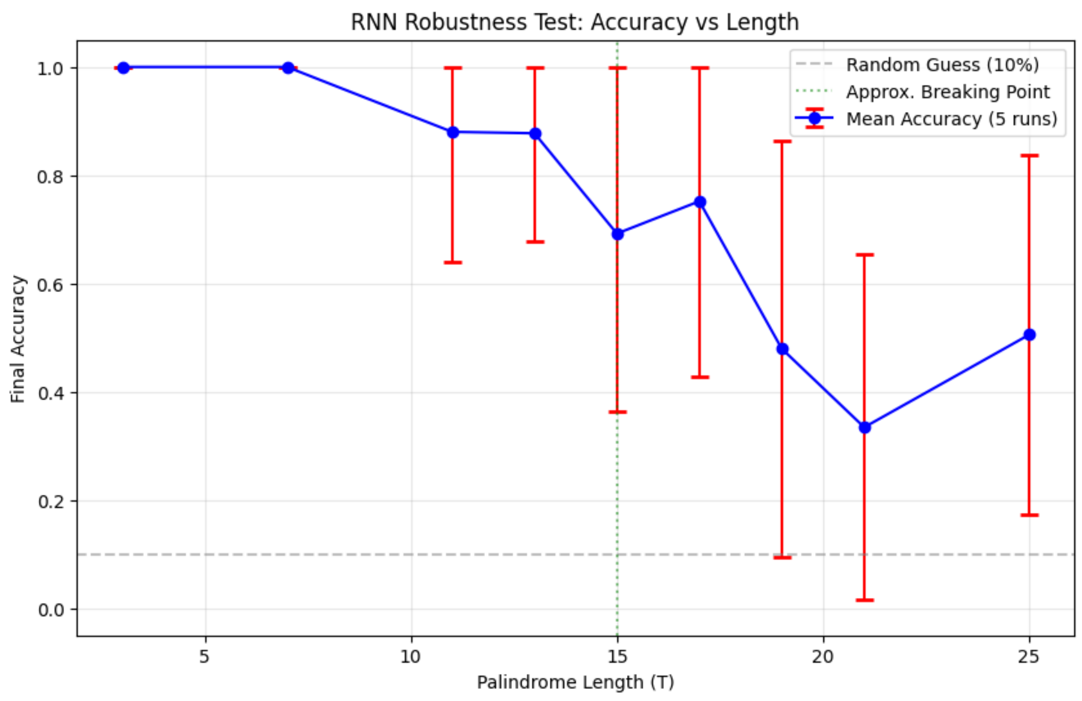


图 7: RNN Average Accuracy vs. Palindrome Length (5 runs)

3.2.1 Phenomenon Analysis: The Vanishing Gradient

The results demonstrate a clear pattern regarding the RNN's ability to learn palindromes of varying lengths:

As the palindrome length T increases, the RNN's accuracy first goes steady, soon decreases significantly, particularly beyond $T = 15$. This drop in performance is primarily due to the **Vanishing Gradient** problem inherent in training RNNs over long sequences.

During Backpropagation Through Time (BPTT), the gradient for the first input depends on the repeated multiplication of the Jacobian of the hidden state transition:

$$\frac{\partial Loss}{\partial h^{(0)}} \propto \prod_{t=1}^{T} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} = \prod_{t=1}^{T} W_{hh}^{T} \cdot \text{diag}(\tanh'(z^{(t)}))$$

Since the derivative of tanh is always ≤ 1 (and often close to 0 for saturated neurons), this product decays exponentially with T.

As T increases, the gradient signal becomes too weak to update the weights responsible for storing the initial digits, causing the 'memory loss', thus, it cannot effectively learn long-term dependencies. Then the accuracy drops, I will solve this problem in the next assignment using LSTM/GRU.