CS324: Deep Learning
Assignment 3

Wai Yan Kyaw (12312638)

1 Part I: PyTorch LSTM

1.1 Introduction

I implemented a Long Short-Term Memory (LSTM) network from scratch using PyTorch to solve the palindrome prediction task. A palindrome is a sequence that reads the same forwards and backwards. The specific goal was to predict the T-th digit of a palindrome given the preceding $T-1$ digits. Based on the previous assignment, I knew that standard RNNs struggle with sequence lengths $T > 20$ due to vanishing gradients. My implementation of the LSTM aims to demonstrate how gating mechanisms allow the model to preserve information over long sequences ($T = 30$).

1.2 Model Architecture and Parameters

I designed the model using a custom LSTM cell followed by a linear decoding layer. I utilized One-Hot encoding for the input digits to ensure categorical stability for the network.

1.2.1 Model Structure

The model consists of four internal gating mechanisms and a final output layer:

- **Input Gate (i):** Decides which values to update in the memory.
- **Forget Gate (f):** Decides what information to discard from the previous state.
- **Output Gate (o):** Decides what part of the current cell state to output as the hidden state.
- **Cell Candidate (g):** Creates a vector of new candidate values to be added to the state.

1.2.2 Parameters

For a hidden dimension of $D_{hid} = 128$ and an input dimension of $D_{in} = 10$, the parameters are as follows:

- **Weight Matrices ($W_{ix}, W_{fx}, W_{ox}, W_{gx}$):** Each of size 128×10.
- **Recurrent Weights ($W_{ih}, W_{fh}, W_{oh}, W_{gh}$):** Each of size 128×128.
- **Biases (b_i, b_f, b_o, b_g):** Each of size 128.
- **Forget Gate Bias Initialization:** I explicitly initialized b_f to 1.0 to ensure gradients flow across time steps at the beginning of training.

1.3 Training Procedure and Dimensionality Analysis

I used the RMSProp optimizer with a learning rate of 0.001 and applied gradient clipping at a max norm of 10.0. During the forward pass, I tracked the matrix dimensions carefully to ensure correct computation.
Let $B = 128$ (Batch Size), $D_{in} = 10$, and $D_{hid} = 128$. For each time step t:

1. **Input Processing:** The input batch $x^{(t)}$ is $(B \times D_{in}) = (128 \times 10)$.

2. **Gate Transformation:** I computed the gates by multiplying the input and previous hidden state (128×128) with their respective weights.

$$(128 \times 10) \cdot (10 \times 128) + (128 \times 128) \cdot (128 \times 128) \rightarrow (\mathbf{128 \times 128}) \tag{1}$$

3. **Cell State Update:** $c^{(t)} = g^{(t)} \odot i^{(t)} + c^{(t-1)} \odot f^{(t)}$. All matrices are (128×128), so the dimensions remain unchanged.

4. **Output Projection:** At the final step T, I projected the hidden state $h^{(T)}$ (128×128) through the output weights (128×10).

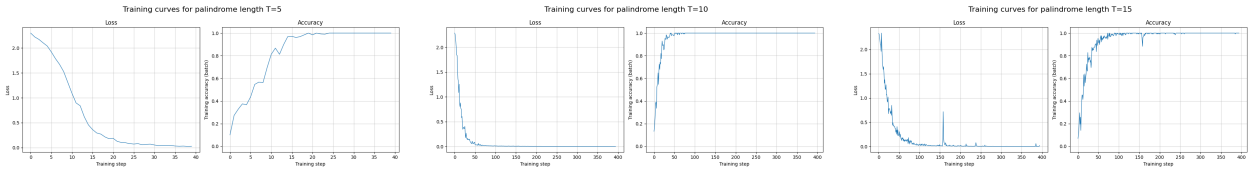
$$(128 \times 128) \cdot (128 \times 10) \rightarrow (\mathbf{128 \times 10}) \tag{2}$$

1.4 Detailed Analysis of Results

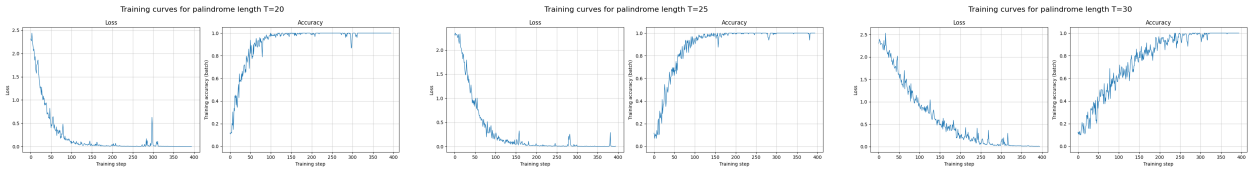
I evaluated the model on sequence lengths $T \in \{5, 10, 15, 20, 25, 30\}$.

1.4.1 Training Dynamics

As shown in the training curves below, the LSTM achieves convergence extremely fast. Even for the longest sequence $T = 30$, the loss dropped from 2.3 to near 0 within the first 200 training steps.



(a) $T = 5$ (b) $T = 10$ (c) $T = 15$

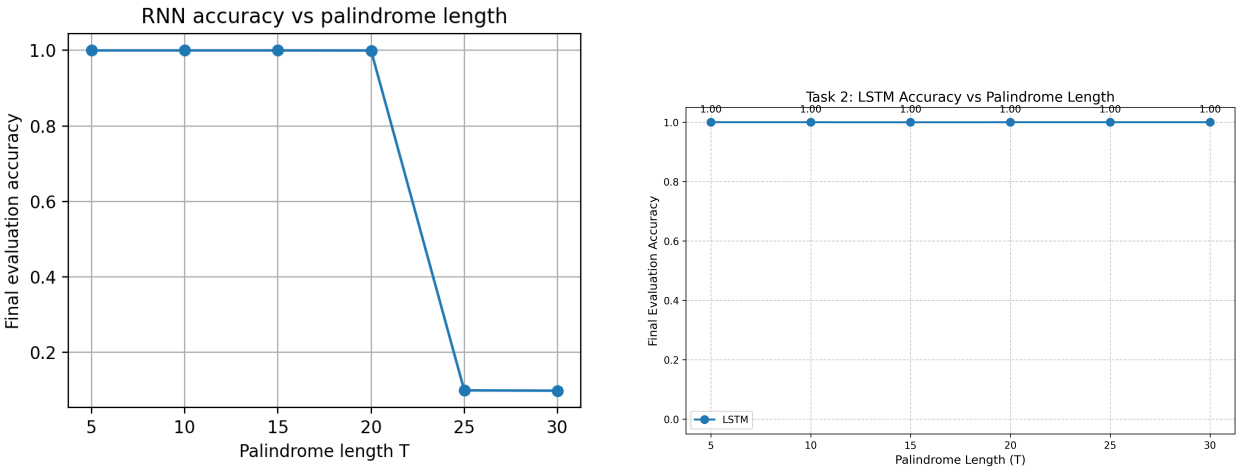


(d) $T = 20$ (e) $T = 25$ (f) $T = 30$

Figure 1: Training loss and batch accuracy curves for different palindrome lengths T. Each subfigure shows the loss (left) and accuracy (right) over training steps for a fixed T.

1.4.2 Accuracy vs. Palindrome Length

The primary objective was to see if the LSTM could outperform the Vanilla RNN on long sequences. The following plot shows the final evaluation accuracy for each length.



(a) Vanilla RNN Accuracy

(b) LSTM Accuracy

Figure 2: Comparison of Accuracy vs. Palindrome Length (T). The Vanilla RNN (a) fails for sequences $T > 20$, while the LSTM (b) maintains perfect accuracy up to $T = 30$.

Analysis: I observed that the LSTM maintained a perfect accuracy of **1.0 (100%)** across all tested lengths, including $T = 30$. This is a significant improvement over the Vanilla RNN, which dropped to random-guess levels (≈ 0.1) at $T = 30$ in the previous assignment. This success is due to the additive nature of the Cell State ($c^{(t)}$), which allows the gradient to backpropagate to the very first digit of the sequence without decaying exponentially. By initializing the Forget Gate bias to 1.0, I ensured the model started with a memory, which was essential for capturing the symmetric nature of palindromes.

2 Part II: Generative Adversarial Networks

I implemented a Generative Adversarial Network (GAN) to generate handwritten digits resembling the MNIST dataset. The training process involves a minimax game between two neural networks: the Generator (G) and the Discriminator (D).
The optimization objective is given by:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{3}$$

2.1 Task 1: Model Architecture and Implementation

I implemented the GAN using PyTorch. The architecture consists of fully connected (Linear) layers.

2.1.1 1. Generator Structure

The Generator takes a latent vector z from a standard normal distribution and maps it to the image space.

- **Input:** Latent vector $z \in \mathbb{R}^{100}$.
- **Hidden Layers:** Four linear layers with Batch Normalization and LeakyReLU activation (negative slope 0.2).
- **Output:** A vector of size 784, reshaped to $1 \times 28 \times 28$. The final activation is `Tanh` to map values to $[-1, 1]$.

Dimension Flow Calculation (Batch Size $B = 64$):

Layer	Input Shape	Operation	Output Shape
Input	$(64, 100)$	Sample $z \sim \mathcal{N}(0, I)$	$(64, 100)$
Linear 1	$(64, 100)$	$W_1x + b_1 \xrightarrow{\text{LeakyReLU}}$	$(64, 128)$
Linear 2	$(64, 128)$	$W_2x + b_2 \xrightarrow{\text{BN}} \xrightarrow{\text{LeakyReLU}}$	$(64, 256)$
Linear 3	$(64, 256)$	$W_3x + b_3 \xrightarrow{\text{BN}} \xrightarrow{\text{LeakyReLU}}$	$(64, 512)$
Linear 4	$(64, 512)$	$W_4x + b_4 \xrightarrow{\text{BN}} \xrightarrow{\text{LeakyReLU}}$	$(64, 1024)$
Linear 5	$(64, 1024)$	$W_5x + b_5 \xrightarrow{\text{Tanh}}$	$(64, 784)$
Reshape	$(64, 784)$	View	$(64, 1, 28, 28)$

Table 1: Computation flow for the Generator

2.1.2 2. Discriminator Structure

The Discriminator is a binary classifier that takes an image and outputs the probability of it being real.

- **Input:** Image tensor $(1, 28, 28)$, flattened to 784.
- **Hidden Layers:** Two linear layers with LeakyReLU.
- **Output:** Single scalar value with `Sigmoid` activation.

Dimension Flow Calculation:

Layer	Input Shape	Operation	Output Shape
Input	$(64, 1, 28, 28)$	Flatten	$(64, 784)$
Linear 1	$(64, 784)$	$W_1x + b_1 \xrightarrow{\text{LeakyReLU}}$	$(64, 512)$
Linear 2	$(64, 512)$	$W_2x + b_2 \xrightarrow{\text{LeakyReLU}}$	$(64, 256)$
Linear 3	$(64, 256)$	$W_3x + b_3 \xrightarrow{\text{Sigmoid}}$	$(64, 1)$

Table 2: Computation flow for the Discriminator

2.2 Task 2: Training Evolution Analysis

The model was trained for 50 epochs using the Adam optimizer with a learning rate of 0.0002. Then, I tracked the visual quality of generated samples at three distinct stages: start, middle, and end.

(a) Start (Epoch 0) (b) Middle (Epoch 25) (c) End (Epoch 50)

Figure 3: Visual evolution of generated samples during GAN training.

2.2.1 Analysis of Results

- **Start (Fig 3a):** At the very first batch, the weights are initialized randomly. The output is purely random noise (Gaussian static). This confirms the lack of any learned structure.
- **Middle (Fig 3b):** Halfway through training, the Generator begins to capture the spatial locality of the data. It produces blob-like shapes concentrated in the center of the frame. The digits are blurry and malformed, indicating that the Generator has learned the general position but not the fine details required to fool the Discriminator fully.
- **End (Fig 3c):** Upon termination, the generated images are distinct and recognizable digits (0-9). The strokes are sharper, and there is a diversity of classes, suggesting the model successfully avoided mode collapse.

2.3 Task 3: Latent Space Interpolation

To verify that the Generator has learned a meaningful continuous representation of the data (rather than simply memorizing training images), I performed linear interpolation in the latent space.

I sampled two random latent vectors, z_1 and z_2, and generated 7 intermediate steps using the formula:

$$z' = (1 - \alpha)z_1 + \alpha z_2, \quad \alpha \in [0, 1]$$

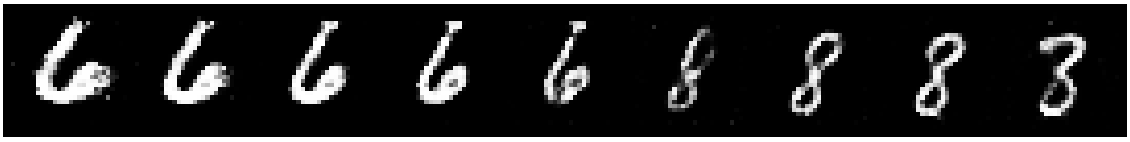


Figure 4: Linear interpolation between two latent vectors.

2.3.1 Detailed Analysis

As observed in Figure 4:

- The leftmost image is clearly a **6**.
- The rightmost image resembles an **8** (or a closed 3).
- **Transition:** Moving from left to right, we observe a smooth morphing process. The top loop of the '6' gradually curls over (steps 4-5) and eventually connects to the center (steps 6-7) to form the upper loop of the '8'.

This smooth transition confirms that the latent space is topologically continuous. If the model had merely memorized images, the interpolation would likely show a sudden jump or a superposition of two ghosts. Instead, the gradual structural change indicates the Generator understands the underlying manifold of handwritten digits.

3 Instructions to Run Code

3.1 Part 1

Run `lstm.ipynb`.

3.2 Part 2

Type command `python my_gan.py`. Then, run `report.ipynb`.