DEEP LEARNING (CS324)
ASSIGNMENT 3 REPORT

Southern University of Science and Technology

Instructor: Prof. Zhang Jianguo

Student Name: THA Sreyny

Student ID: 12113053

December 23, 2024

1. INTRODUCTION

■ Long Short-Term Memory (LSTM) RNN

Long Short-Term Memory (LSTM) networks are a specialized form of recurrent neural networks (RNNs) that excel at learning and retaining information over long sequences. Unlike standard RNNs, which struggle with the issue of long-term dependencies, LSTMs are designed to overcome this challenge. They are an advanced version of RNNs, incorporating mechanisms such as cell states and gates to effectively handle long-term dependencies in data.

■ Generative Adversarial Network (GAN)

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow and his team in 2014, are a class of machine learning frameworks focused on generating synthetic data that closely resembles real-world data. A GAN consists of two neural networks: the Generator and the Discriminator. These networks compete in a game-theoretic manner, with the Generator aiming to create realistic data, while the Discriminator attempts to distinguish between real and generated data.

2. MOTIVATION

The primary objectives of this project are as follows: The main objectives of this assignment are as follows:

1. Implement an LSTM to predict the last digit of a palindrome and then evaluate its loss and accuracy.
2. Compare the performance of the LSTM implementation with that of the Vanilla RNN from Assignment 2.
3. Develop and train a Generative Adversarial Network (GAN) on the MNIST dataset.
4. Generate 25 images from the trained GAN.
5. Generate two images from different classes using the GAN, and perform interpolation between these two digits in the latent space.

3. METHODOLOGY

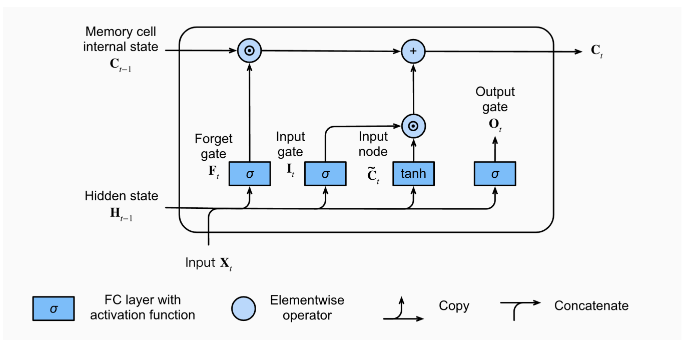
This assignment is divided into two parts:

1. PyTorch LSTM Implementation
2. Generative Adversarial Network (GAN) Implementation

■ Part I: PyTorch LSTM

In this part of the assignment, the task is to generate the last digit of a palindrome using Long Short-Term Memory (LSTM) in PyTorch. Compared to the previous implementation that used a Vanilla Recurrent Neural Network (RNN), this LSTM implementation offers enhanced features, including the use of cells and gates, which make it more powerful. The structure of an LSTM is described below.

Figure 1. LSTM Structure


Memory cell internal state $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
Output gate $\mathbf{O}_t$
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
σ
tanh
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

Source: d2l.ai

The LSTM has a more intricate architecture, consisting of multiple gates as opposed to the single neuron in a Vanilla RNN. The gates in an LSTM are as follows:

- **Input Gate (i_t)**: The input gate controls the extent to which new information from the current input (x_t) and the previous hidden state (h_{t-1}) should be allowed into the cell state. It helps determine which values should be updated in the cell state.

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

  where W_{ix} and W_{ih} are the weights for the input x_t and the hidden state h_{t-1}, respectively, and b_i is the bias term. The function σ denotes the sigmoid function, which outputs values between 0 and 1, allowing the gate to control the amount of information passed through.
- **Forget Gate (f_t)**: The forget gate controls which parts of the previous cell state (c_{t-1}) should be carried forward. It decides what to forget or retain from the past.

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

  where W_{fx} and W_{fh} are the weights for the input x_t and the hidden state h_{t-1}, respectively, and b_f is the bias term. The sigmoid function σ outputs a value between 0 and 1 for each element of the cell state c_{t-1}.
- **Output Gate (o_t)**: The output gate controls which part of the cell state should be passed on to the next hidden state.

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$$

  where W_{ox} and W_{oh} are the weights for the input x_t and the hidden state h_{t-1}, respectively, and b_o is the bias term. The output of the sigmoid function σ determines how much of the cell state (via the tanh function) is passed to the output.
- **Candidate Cell State (g_t)**: The candidate cell state represents the new information that may be added to the cell state.

$$g_t = \tanh(W_{gx}x_t + W_{gh}h_{t-1} + b_g)$$

  where W_{gx} and W_{gh} are the weights for the input x_t and the hidden state h_{t-1}, respectively, and b_g is the bias term. The tanh function ensures that the candidate values are in the range $[-1, 1]$.
- **Cell State (c_t)**: The cell state is the central component of the LSTM that carries information over different time steps. It is updated at each step depending on the input and forget gates.

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

  where $\odot$ denotes element-wise multiplication. The forget gate f_t determines how much of the previous cell state c_{t-1} is retained, and the input gate i_t controls how much of the candidate cell state g_t is added.
- **Hidden State (h_t)**: The hidden state is the output of the LSTM cell at the current time step. It is used as the input for the next time step and can also be used for the final output of the network.

$$h_t = o_t \odot \tanh(c_t)$$

  where o_t decides what part of the cell state is output, and $\tanh(c_t)$ transforms the cell state to values between $[-1, 1]$.

■ Part II: GAN

In this section, the goal is to build and train a Generative Adversarial Network (GAN) using the MNIST dataset to generate samples. A GAN consists of two neural networks: the Generator and the Discriminator, which are trained in a competitive, game-theoretic manner.

Discriminator Objective Function

The Discriminator's goal is to maximize the following objective function:

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

Generator Objective Function

The Generator's goal is to maximize the following objective function:

$$L_G = \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{2}$$

Alternative Generator Objective Function

In practice, an alternative loss function for the Generator is used to avoid vanishing gradients:

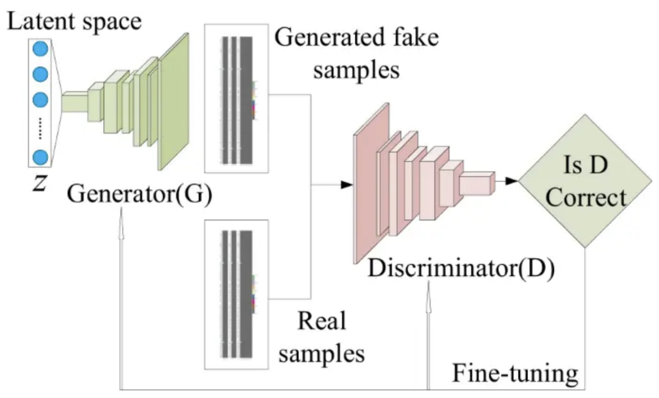
$$L_G = -\mathbb{E}_{z\sim p_z}[\log D(G(z))] \tag{3}$$

GAN Architecture (Figure 2)

Generator (G): The Generator generates synthetic data that resembles the real data. It takes random noise from a sampled distribution as input and transforms it into a data sample. The goal is to produce data that the Discriminator cannot distinguish from real data, thereby maximizing the likelihood of the Discriminator making an error.

Discriminator (D): The Discriminator's task is to differentiate between real data (from the training set) and fake data generated by the Generator. It receives a data sample and outputs a probability indicating whether the sample is real or fake. The Discriminator seeks to minimize the chance of making an incorrect classification.

Figure 2. GAN Structure



Source: (Sumit Singh, 2022)

Training Process

Step 1: Initialization Initialize both the Generator and Discriminator with random weights.

Step 2: Train the Discriminator

- Sample a batch of real data from the training set.
- Sample a batch of random noise vectors and use the Generator to produce fake data.
- Compute the loss using both real and fake data, based on the objective function (1) above.
- Perform backpropagation and update the Discriminator's weights to minimize the loss.

Step 3: Train the Generator

- Sample a batch of random noise vectors and generate fake data with the Generator.
- Compute the loss based on the Discriminator's ability to classify the fake data as real, using the objective function (3).
- Perform backpropagation and update the Generator's weights to maximize the Discriminator's classification error on the fake data.

4. EXPERIMENT

■ I. PyTorch LSTM

1) Dataset Preparation

The dataset is a palindrome dataset with a digit label, which consists of digit numbers with an input length of N. In this experiment, there are 1,000,000 samples, each with an input length of 19. The dataset is divided into a training and validation set with a ratio of 8:2.

2) Network Architecture

As described in Figure 1, each layer has an input gate, forget gate, and output gate.

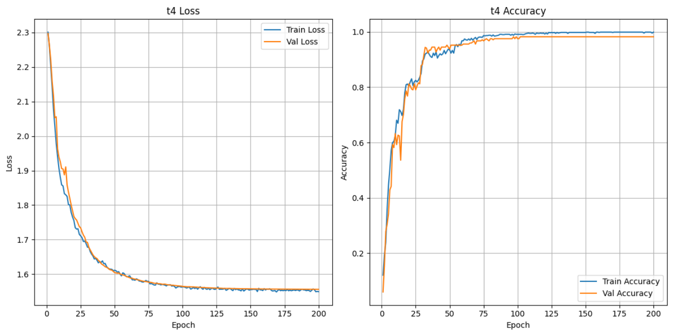
3) Hyperparameter Configuration

- Input dimension: 1
- Number of classes: 10
- Number of hidden units: 128
- Batch size: 128
- Learning rate: 0.001
- Maximum epochs: 200
- Maximum norm: 10.0
- Data size: 100,000
- Training set portion: 0.8

4. Result

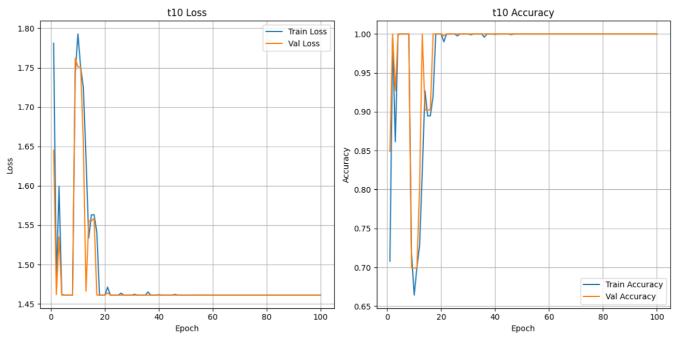
For an input length of 4, the performance result can be obtained as shown in Figure 3.

Figure 3. LSTM Result T=4



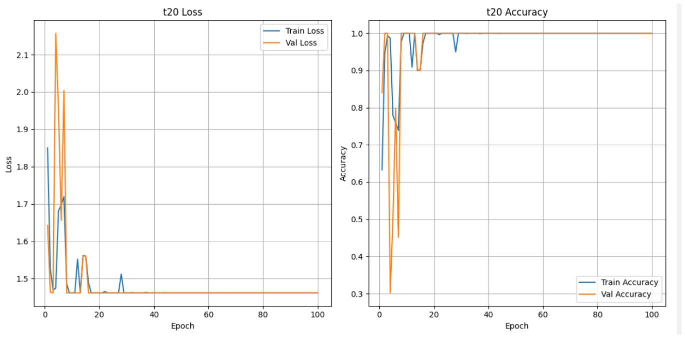
For input length 10, the result is as follow in Figure 4:

Figure 4. LSTM Result T=10



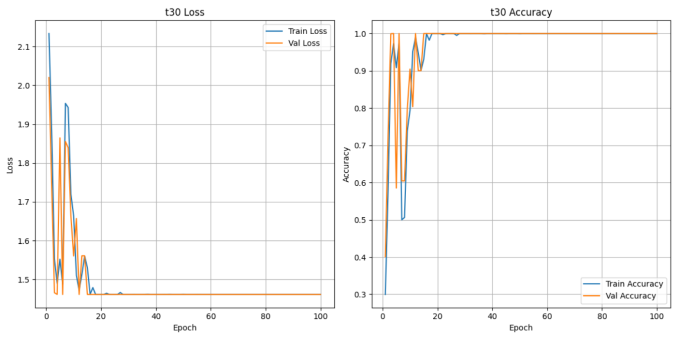
For input length 20, the result is as follow in Figure 5:

Figure 5. LSTM Result T=20



For input length 30, the result is as follow in Figure 6:

Figure 6. LSTM Result T=30



5. Compare LSTM and Vanila RNN

LSTM Model

- The training accuracy increases with input length, stabilizing at a high level (around 1.0) for lengths greater than 6.
- The validation accuracy also rises but stabilizes at a lower level (around 0.8), indicating potential overfitting as the model performs better on training data than on validation data.

Vanilla RNN Model

- The training accuracy exhibits fluctuations, peaking around an input length of 6 and dropping significantly for longer lengths.
- The validation accuracy follows a similar pattern, with significant drops after input lengths of around 6, indicating that the Vanilla RNN struggles to generalize with longer sequences.

Key Observations

- The erratic behavior of the Vanilla RNN's accuracy indicates that it is not be well-suited for longer input sequences, possibly due to issues like vanishing gradients.
- The LSTM shows better generalization compared to the Vanilla RNN, especially as input lengths increase.

Conclusion

The LSTM model demonstrates more robust performance with longer input sequences compared to the Vanilla RNN, which struggles to maintain accuracy. This suggests that LSTMs are better suited for tasks requiring the handling of longer dependencies in sequential data.

■ Part II: GAN

1. Dataset Preparation

The MNIST dataset, consisting of 70,000 grayscale images of handwritten digits (28x28 pixels), was used for training.

2. Network Architecture

- **Generator**: The Generator produces synthetic images from random noise vectors. It uses a series of linear layers with Leaky ReLU activations and Batch Normalization in the hidden layers to stabilize training. The output layer employs Tanh activation to scale the generated images to the range $[-1, 1]$.
- **Discriminator**: The Discriminator classifies images as real or fake, consisting of linear layers with Leaky ReLU activations in the hidden layers and a Sigmoid output layer to output probabilities.

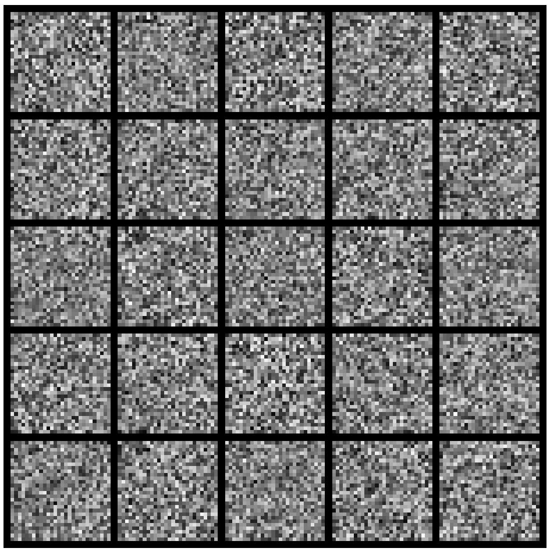
3. Hyperparameter Configuration

- Input size: 100
- Hidden size: 256
- Image size: 28x28
- Batch size: 64
- Epochs: 200
- Learning rate: 0.0001

4. Results

- **Epoch 1**: Initial generated images were rudimentary.
- **Epoch 100**: Generated images showed noticeable improvements.
- **Epoch 200**: The generator produced high-quality images resembling real digits.

Figure 7. beginning of the training



By generating 25 images from the trained GAN, the resulting images are shown in Figure 10. Additionally, two digits were interpolated in the latent space. First, latent vectors z_1 and z_2 were sampled from a standard normal distribution, and the resulting interpolation is shown in Figure 11.

5. Analysis of GAN Results

The generated images from the trained GAN (Figure 9) demonstrate a clear improvement in image quality as training progresses. At the start, the images are noisy and lack discernible patterns, but by the end of the training, the generated digits closely resemble real handwritten digits. This shows the model's capability to learn the distribution of the MNIST dataset.

Figure 8. Half way of training

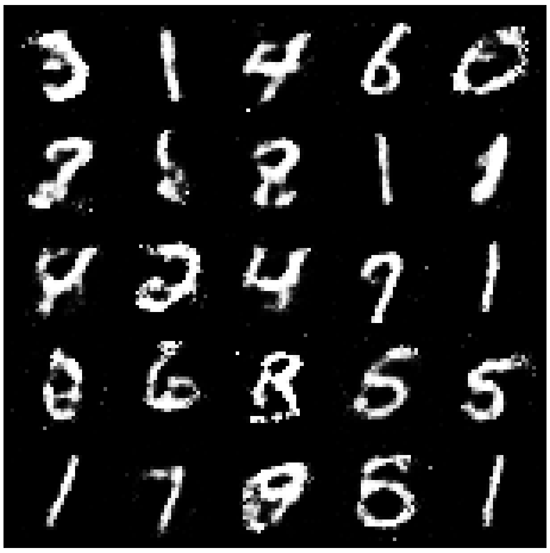
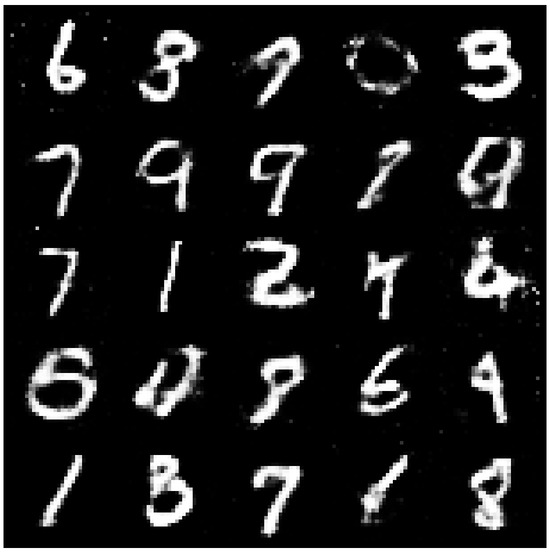


Figure 9. At the end of training



The interpolation experiment between two latent vectors, z_1 and z_2, sampled from a standard normal distribution (Figure 11), showcases the GAN's ability to smoothly interpolate between different digit classes in the latent space. The results indicate that the generator can produce coherent transitions between two distinct digits, suggesting it has learned meaningful representations in the latent space.

5. CONCLUSION

In these experiments, I examined the Long Short-Term Memory (LSTM) model, which successfully tackles the long-term dependency challenges faced by Vanilla RNNs. The findings indicated that LSTM models significantly outperformed Vanilla RNNs, especially when handling longer input sequences,

Figure 10. Sample 25 images from trained GAN

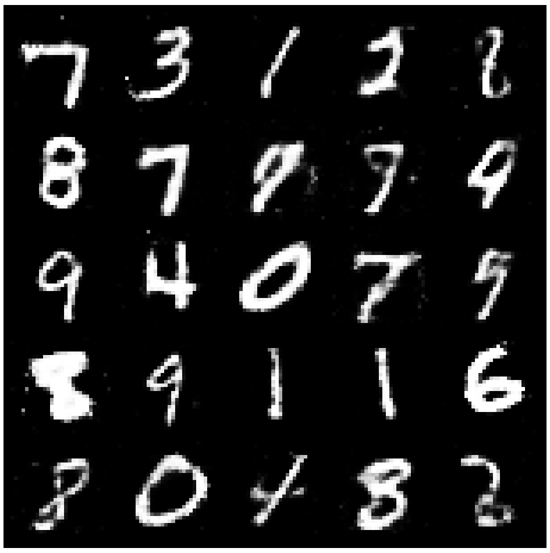
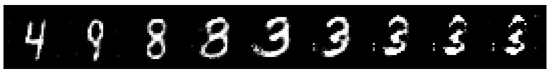


Figure 11. Interpolation from normal distribution



achieving flawless accuracy where Vanilla RNNs struggled. I also explored the application of Generative Adversarial Networks (GANs) for image generation, and the results highlighted GANs' remarkable ability to produce realistic images.