SUSTech CS324 Deep Learning
Report for Assignment 3

Student ID:	12312710
Name:	胥熠/Brighton
Date:	2025 年 12 月 26 日

目录

1 Part I: PyTorch LSTM

1.1 Task 1

In this task, I implemented an LSTM (Long Short-Term Memory) network from scratch using PyTorch, without using `torch.nn.LSTM`. The implementation strictly follows the recurrence equations provided in the assignment.

1.1.1 Model Architecture

The core of the LSTM is the cell state $c^{(t)}$, which acts as a highway for information flow. The network utilizes three gates (Input, Forget, Output) to regulate this flow. The mathematical formulation implemented in `lstm.py` is as follows:

$$\begin{aligned}
g^{(t)} &= \tanh(W_{gx}x^{(t)} + W_{gh}h^{(t-1)} + b_g) \quad \text{(Candidate Memory)} && (1)\\
i^{(t)} &= \sigma(W_{ix}x^{(t)} + W_{ih}h^{(t-1)} + b_i) \quad \text{(Input Gate)} && (2)\\
f^{(t)} &= \sigma(W_{fx}x^{(t)} + W_{fh}h^{(t-1)} + b_f) \quad \text{(Forget Gate)} && (3)\\
o^{(t)} &= \sigma(W_{ox}x^{(t)} + W_{oh}h^{(t-1)} + b_o) \quad \text{(Output Gate)} && (4)\\
c^{(t)} &= g^{(t)} \odot i^{(t)} + c^{(t-1)} \odot f^{(t)} \quad \text{(Cell State Update)} && (5)\\
h^{(t)} &= \tanh(c^{(t)}) \odot o^{(t)} \quad \text{(Hidden State)} && (6)
\end{aligned}$$

I defined separate `nn.Linear` layers for each gate component. Since each gate requires only one bias term, I enabled `bias=True` for the input-processing linear layers and `bias=False` for the hidden-state processing layers.

Furthermore, I utilized **One-Hot Encoding** to represent the input tokens, consistent with my previous assignment. Since the digits are categorical and independent of their numerical values, One-Hot Encoding provides orthogonal features for each digit. The `input_size` is set to 10.

1.2 Task 2: Training and Evaluation

I trained the LSTM model on the provided dataset using the following hyperparameters:

- Input Dimension: 10
- Number of Classes: 10
- Hidden Units: 128
- Batch Size: 128
- Learning Rate: 0.001

- Max Epochs: 100
- Max Gradient Norm: 10.0
- Data Size: 10000
- Train Portion: 0.8

Although the max epoch is set to 100, I designed an **early stopping** mechanism, which terminates training if the validation accuracy reaches 99%. In practice, most training sessions converged in fewer than 30 epochs.

1.2.1 Performance & Comparison

After training, the model achieves the following accuracy results:

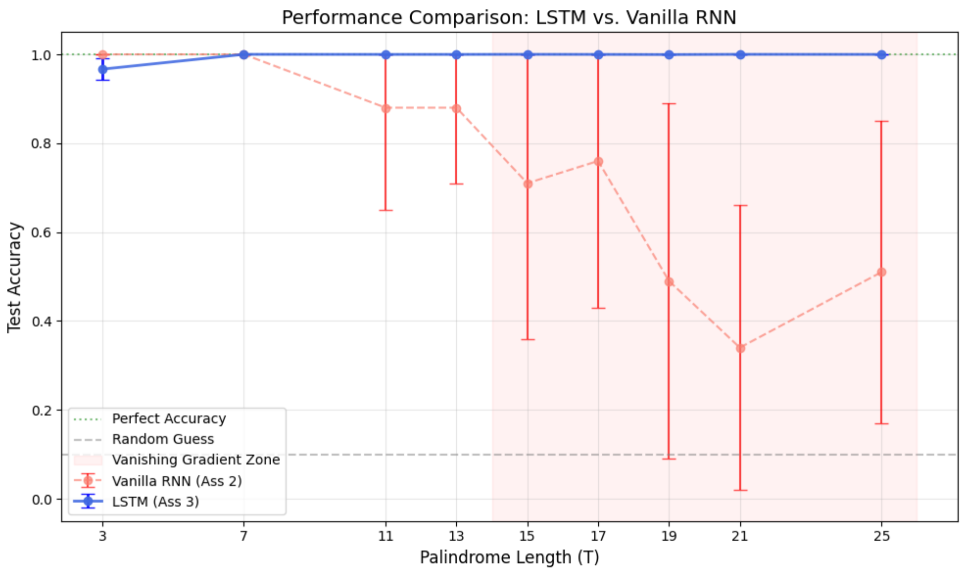


图 1: Performance Comparison: LSTM vs. Vanilla RNN

The accuracy of the RNN is derived from my previous assignment. As shown in Figure 1, the RNN model performs acceptably on short sequences (around 10). However, as the sequence length increases, the performance degrades significantly due to the vanishing gradient problem.

In contrast, the LSTM model maintains high accuracy even on long sequences, achieving **exactly 100%** accuracy on sequences of length 25.

One noticeable pattern is that the LSTM's performance did not reach 100% on sequences of length 3. For such short sequences, the possible combinations are extremely limited (only 100 combinations). Consequently, the restricted dataset size limits the gradient updates per epoch, making it harder for the complex LSTM to converge, whereas the simpler RNN benefits from its lighter structure on such small data. (Probably I guess?)

Overall, the LSTM demonstrates superior performance over the RNN, especially on longer sequences, validating its effectiveness in handling long-term dependencies.

2 Part II: Generative Adversarial Networks

2.1 Task 1: Implementation Details

In this part, I implemented a Generative Adversarial Network (GAN) using PyTorch to generate MNIST-like handwritten digit images. The GAN consists of two main components: a Generator and a Discriminator.

2.1.1 Network Architecture

To adapt to the MNIST dataset, I designed the following architectures for the Generator and Discriminator:

- **Generator (G)**:
  - **Input**: A latent vector $z \in \mathbb{R}^{100}$ sampled from a standard normal distribution $\mathcal{N}(0, 1)$.
  - **Hidden Layers**: Four linear layers with increasing dimensions ($128 \to 256 \to 512 \to 1024$), each followed by Batch Normalization and LeakyReLU activation ($\alpha = 0.2$).
  - **Output**: A linear layer mapping to 784 (28×28) dimensions, followed by a **Tanh** activation function to squash pixel values into the range $[-1, 1]$.
- **Discriminator (D)**:
  - **Input**: Flattened image vector $x \in \mathbb{R}^{784}$.
  - **Architecture**: Three linear layers with decreasing dimensions ($784 \to 256 \to 128$), each followed by LeakyReLU activation ($\alpha = 0.2$) and Dropout ($p = 0.4$) for regularization.
  - **Output**: A single scalar with **Sigmoid** activation, representing the probability that the input is real.

2.1.2 Loss Function & Optimization

The training objective follows the Minimax game:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{7}$$

I used the **Adam** optimizer with a learning rate of 0.0002 and $\beta_1 = 0.5$.

2.2 Task 2: Training Process & Dynamic Analysis

This part is particularly interesting. I **initially** followed the **recommended** structure of Generator and Discriminator ($784 \to 512 \to 256$ without Dropout). However, I found that the Discriminator quickly became too strong, leading to vanishing gradients for the Generator.

Here are some observations during training:

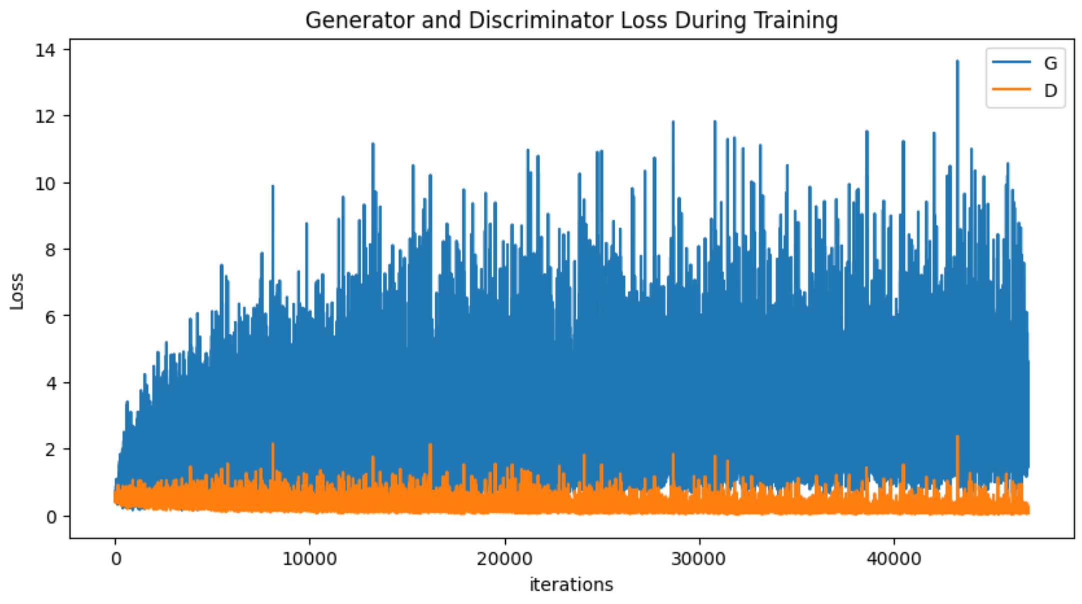


图 2: Loss Dynamics of Discriminator and Generator during Initial Training

As shown in Figure 2, the Discriminator's loss rapidly decreases to near zero, indicating it can easily distinguish real from fake images. Conversely, the Generator's loss spikes and remains high. This is very problematic because when the Discriminator is too confident, the Generator receives almost no useful gradient information to improve. The generated images after 50 epochs were **unsatisfactory**, as shown below:

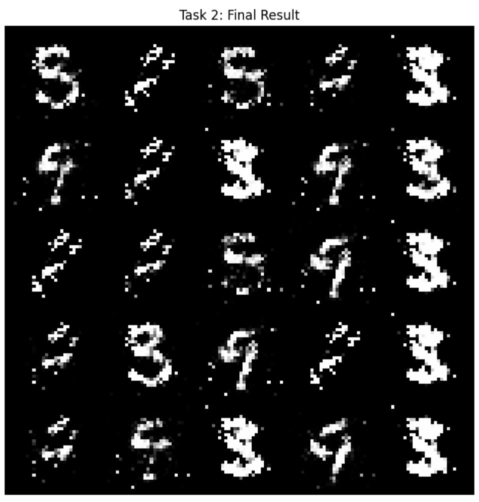


图 3: Generated Images after 50 Epochs with Initial Architecture

2.2.1 Balancing the Power of Generator and Discriminator

To address this problem, I had to decrease the Discriminator's capacity by adding Dropout layers (with $p = 0.4$) and reducing its number of neurons ($784 \to 256 \to 128$). This adjustment helps to balance the power between the Generator and Discriminator, allowing both networks to learn effectively. After adjustment as the architecture mentioned above, the training dynamics improved significantly, as shown below:

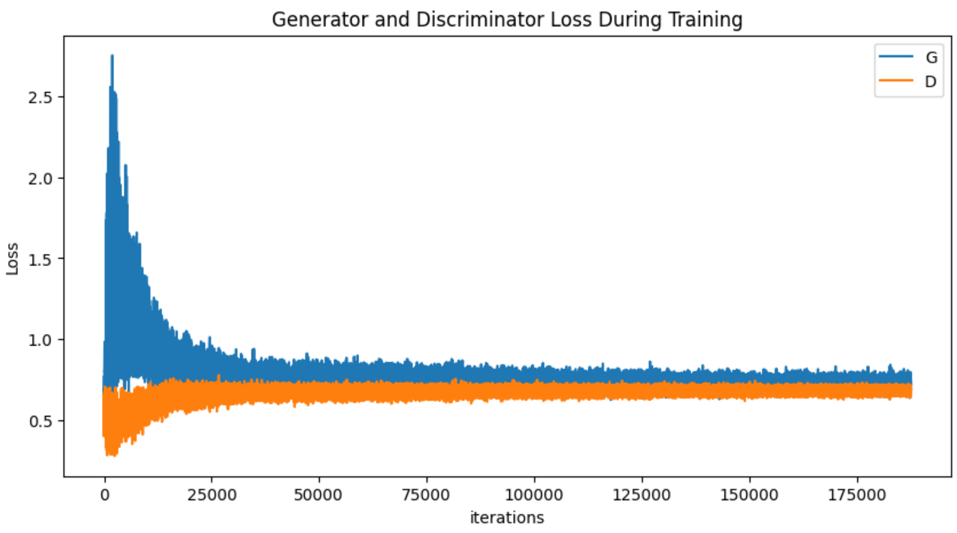
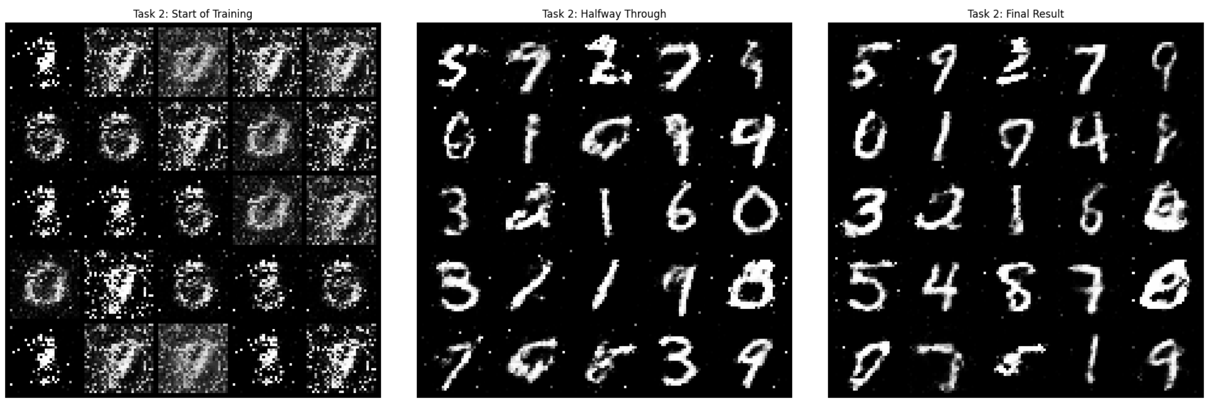


图 4: Loss Dynamics of Discriminator and Generator after Balancing

As shown in Figure 4, both losses fluctuate around a stable point. More surprisingly, the loss of both networks hovers around $\log(2) \approx 0.693$. This indicates that the Discriminator is guessing randomly (50% accuracy), suggesting that the Generator has become proficient enough to produce realistic images that can fool the Discriminator half the time.

Finally, after 200 epochs of training with the balanced architecture, the Generator produces high-quality images that closely resemble real MNIST digits:



(a) Beginning (b) Middle (c) Final

图 5: GAN Generated Images Progression in Different Training Stages

As we can observe, at the beginning (Figure 5a), the images are mostly noise with few consistent patterns. By the middle of training (Figure 5b), we can see digit-like structures with few noise. Finally, at the end of training (Figure 5c), the generated images look nicer, although some digits are still a bit weird.

2.3 Task 3: Latent Space Interpolation

To verify that the GAN has learned the underlying continuous manifold of the MNIST data rather than simply memorizing training samples, I performed a linear interpolation in the latent space.

I sampled two latent vectors z_1 and z_2 corresponding to distinct digits and generated images for $z_\alpha = (1 - \alpha)z_1 + \alpha z_2$.

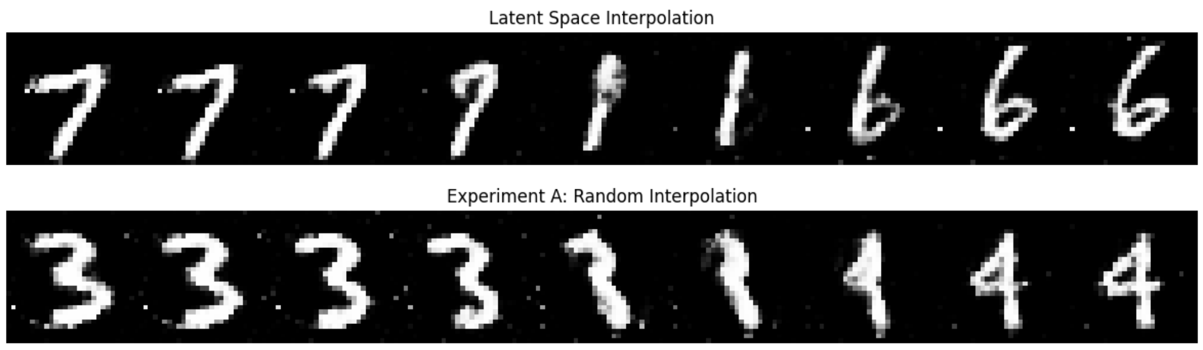

Latent Space Interpolation
Experiment A: Random Interpolation

图 6: Latent Space Interpolation

The above figure shows a smooth transition from digit `7` to digit `6` as α varies from 0 to 1. And as we can see, the process is interesting. Since both digits have similar structural components, the transition appears smooth and natural, and the intermediate images look like `1`. We can probably guess that: In GAN's latent space, `1` is located between `7` and `6`.

But... Which features correspond to this transition?

2.4 Further Exploration: Vector Arithmetic

Inspired by the property of word embeddings (e.g., $King - Man + Woman = Queen$), I investigated whether the GAN's latent space captures semantic arithmetic.

Based on the interpolation observation that `1` appears between `7` and `6`, I hypothesized that a `7` can be conceptually decomposed into a `1` (vertical stroke) plus a `roof` (top horizontal stroke). To test this, I extracted a **"Roof Feature Vector"** (z_{roof}) and injected it into a digit `6`:

$$z_{roof} = z_7 - z_1 \tag{8}$$

What does this `roof` look like? We can visualize it by generating images from z_{roof} to z_6 to see how it modifies the digit '6'.

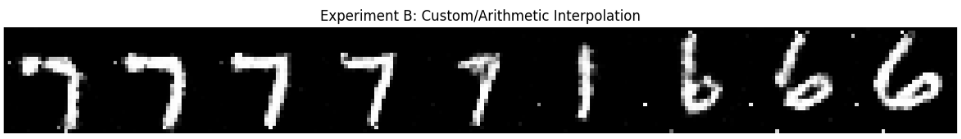


图 7: `roof` to '6' Transition

Wait... Why the roof still looks like a `7`? I get that `7` might occupy a huge space in the latent space. We have to conduct further experiments by varying the intensity of the `roof` feature.

$$z_{hybrid} = z_6 + 1.2 \cdot z_{roof} \tag{9}$$

Here is the result:

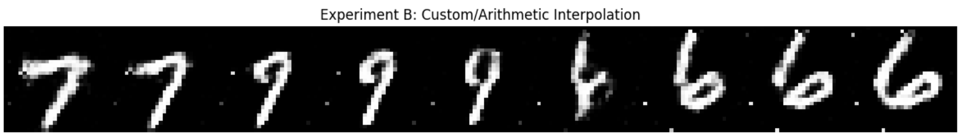


图 8: z_{hybrid} to `6` Transition

As shown in Figure 8, we observe a fascinating transformation: We even see a digit that resembles `9` emerging from the combination of `6` and the `roof` feature. That is:

$$\begin{aligned} z_9 &\approx z_6 + 0.8 \cdot z_{hybrid} \\ &\approx z_6 + 0.8 \times (z_6 + 1.2 \cdot z_{roof}) \\ &\approx z_6 + 0.8 \times (z_6 + 1.2 \times (z_7 - z_1)) \\ &\approx 1.8z_6 + 0.96z_7 - 0.96z_1 \\ &\approx 1.8(bar + 6's\, loop) + 0.96(bar + 7's\, roof) - 0.96(bar) \\ &\approx 1.8 \cdot bar + 1.8 \cdot 6's\, loop + 0.96 \cdot 7's\, roof \end{aligned} \tag{10}$$

It seems that this GAN might learn the features in a somewhat different way than we expect. This suggests that the weight of **6's loop** is higher than **7's roof**, so the final result has a loop? We don't exactly know. However, this is very interesting and worth further exploration!

3 Conclusion

In this assignment, I implemented and analyzed two fundamental deep learning architectures:

1. **LSTM**: I successfully addressed the vanishing gradient problem inherent in RNNs. The LSTM implementation achieved 100% accuracy on palindrome sequences up to length 25.
2. **GAN**: I trained a GAN to generate MNIST digits and overcame the ”Discriminator Overpowering” instability by regulating the discriminator’s capacity. Through latent space interpolation and vector arithmetic experiments (e.g., $7 - 1 + 6 \approx 9$), I demonstrated that the GAN learned a semantically meaningful data manifold, where digits are represented as compositions of structural features (strokes and loops) rather than isolated pixel maps.

Actually, this course is the most interesting one I have taken in this semester. I love doing abstract thinking, and deep learning is like a field full of imagination and creativity. Moreover, deep learning is also very practical and powerful, which can solve many real-world problems. I have already started to use deep learning in my research work, even understand this world using the philosophy of deep learning.

Thanks to Prof. Jianguo Zhang and TAs for such great course and assignments!