Deep Learning Assignment 3

Junyou Su 12110911

December 12, 2024

1 Instructions on how to run the code

File Structure

- Assignment_3_LSTM&GAN
  - Part 1
    - dataset.py – Script for loading and preprocessing the dataset.
    - lstm.py – LSTM model definition file.
    - p1.ipynb – Jupyter Notebook for Part 1, containing experiment steps and results analysis.
    - train.py – Script for training the LSTM model.
    - utils.py – Utility functions and helper scripts.
  - Part 2
    - data – Directory for storing data files used in Part 2.
    - images – Directory for storing generated image files.
    - images_simple_gan – Directory for storing Simple GAN-generated image files.
    - images_simple_gan.png – Simple GAN-generated image result (example 1).
    - images_simple_gan2.png – Simple GAN-generated image result (example 2).
    - images.png – General GAN-generated image results.
    - interpolation_between_1_and_8.png – Interpolation result between digits 1 and 8.
    - interpolation_between_7_and_9_simple_gan.png – Simple GAN interpolation result between digits 7 and 9.
    - mnist_generator_simple_gan.pt – Trained Simple GAN generator model file.
    - mnist_generator.pt – Trained GAN generator model file.
    - my_gan.py – Script for training the GAN model.
    - p2.ipynb – Jupyter Notebook for Part 2, containing experiment steps and results analysis.
    - simple_gan.py – Script defining and training the Simple GAN model.
- Assignment3_2024.pdf – PDF file describing the assignment requirements.

1.1 How to Run

All instructions for running the code can be found in the Jupyter notebooks: `p1.ipynb` and `p2.ipynb`. Additionally, to train the simple GAN, run: `python simple_gan.py`. To train the DCGAN, run: `python my_gan.py`.

2 The Structure of Models

2.1 Part 1 - LSTM

The LSTM model architecture and its training setup for this task are described below:

- **Model Architecture:**
  - The LSTM model is implemented using PyTorch and inherits from the `nn.Module` class.
  - It consists of the following components:
    - `i2h`: A fully connected linear layer that processes the concatenation of the input and the hidden state, mapping it to four times the hidden dimension. This layer computes the input gate, forget gate, candidate cell state, and output gate.
    - `h2o`: A fully connected linear layer that maps the hidden state to the output dimension.
    - Activation functions:
      - `tanh`: Used to apply non-linear transformations to the cell state and candidate cell state.
      - `sigmoid`: Used for the gates (input gate, forget gate, and output gate) to constrain their values between 0 and 1.
  - The model processes sequences of fixed length (`seq_length`) with an input dimension (`input_dim`), a hidden dimension (`hidden_dim`), and an output dimension (`output_dim`).
- **Forward Pass:**
  - The forward method takes a batch of input sequences (`x`) with dimensions `[batch_size, seq_length, input_dim]`.
  - The model initializes the hidden state (`h_t`) and cell state (`c_t`) as zeros with dimensions `[batch_size, hidden_dim]`.
  - For each time step in the sequence:
    - The input at the current time step and the hidden state are concatenated and passed through the `i2h` layer to compute the gates.
    - The gates are split into input gate (`i_t`), forget gate (`f_t`), candidate cell state (`g_t`), and output gate (`o_t`).
    - The cell state is updated as:
$$c_t = f_t \cdot c_t + i_t \cdot g_t$$
    - The hidden state is updated as:
$$h_t = o_t \cdot \tanh(c_t)$$
    - The hidden state is passed through the `h2o` layer to compute the output.
  - The output from the final time step is returned.

2.2 Part 2 - GAN

2.2.1 Simple GAN

The Simple GAN model is composed of two main components: the Generator and the Discriminator. The architecture and their functionalities are described below:

- **Generator:**
  - The Generator takes a latent vector `z` as input, sampled from a noise distribution (e.g., Gaussian distribution), and generates an image representation.
  - The architecture consists of a series of fully connected (`Linear`) layers with increasing dimensions, interleaved with activation functions and batch normalization.
  - The structure is as follows:
    - `Linear`: Maps `latent_dim` to 128.
    - `LeakyReLU(0.2)`: Introduces non-linearity with a slight slope for negative values to avoid dead neurons.
    - `Linear`: Maps 128 to 256.

    - `BatchNorm1d(256)`: Normalizes intermediate outputs to stabilize training.
    - `LeakyReLU(0.2)`: Activation function.
    - `Linear`: Maps 256 to 512.
    - `BatchNorm1d(512)`: Normalization layer.
    - `LeakyReLU(0.2)`: Activation function.
    - `Linear`: Maps 512 to 1024.
    - `BatchNorm1d(1024)`: Normalization layer.
    - `LeakyReLU(0.2)`: Activation function.
    - `Linear`: Maps 1024 to 784 (image dimension for a flattened 28x28 image).
    - `Tanh()`: Output non-linearity to constrain pixel values to the range $[-1, 1]$.
  - The forward pass generates an image using the sequence of layers applied to the latent vector `z`.

- **Discriminator:**
  - The Discriminator evaluates the validity of an image (real or fake) by mapping it to a scalar probability.
  - The architecture consists of fully connected (`Linear`) layers with decreasing dimensions, interleaved with activation functions.
  - The structure is as follows:
    - `Linear`: Maps 784 (flattened 28x28 image) to 512.
    - `LeakyReLU(0.2)`: Introduces non-linearity with a small slope for negative values.
    - `Linear`: Maps 512 to 256.
    - `LeakyReLU(0.2)`: Activation function.
    - `Linear`: Maps 256 to 1.
    - `Sigmoid()`: Output non-linearity to constrain the validity score to the range $[0, 1]$.
  - The forward pass computes the validity score of the input image by flattening it and passing it through the layers.

- **Training Setup:**
  - The training involves an adversarial setup where:
    - The Generator aims to produce realistic images to fool the Discriminator.
    - The Discriminator aims to distinguish between real images (from the dataset) and fake images (generated by the Generator).
  - The loss function used for both components is the Binary Cross-Entropy (BCE) loss.
  - The training alternates between optimizing the Generator and the Discriminator to minimize their respective losses.

2.2.2 DCGAN (Deep Convolutional GAN)

The DCGAN model consists of two main components: the Generator and the Discriminator, both utilizing convolutional layers to improve the quality of generated images and stabilize the training process. The architecture and functionalities are described below:

- **Generator:**
  - The Generator takes a latent vector `z` as input and generates an image using transposed convolutions.
  - The architecture is structured as follows:
    - `Linear`: Maps the latent dimension (`latent_dim`) to a feature space of size $128 \times 7 \times 7$.
    - Reshape: Converts the flattened vector into a 4D tensor with dimensions (`batch_size`, $128, 7, 7$).
    - `ConvTranspose2d`: Performs upsampling using transposed convolutions.

· `ConvTranspose2d(128, 128, kernel_size=4, stride=2, padding=1)`: Doubles the spatial resolution to 14×14.
· `BatchNorm2d(128)`: Normalizes intermediate outputs to stabilize training.
· `ReLU`: Activation function.
· `ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1)`: Further upsamples to 28×28.
· `BatchNorm2d(64)`: Normalization layer.
· `ReLU`: Activation function.
* `Conv2d(64, 1, kernel_size=7, stride=1, padding=3)`: Generates the final output image with a single channel (grayscale).
* `Tanh`: Constrains pixel values to the range $[-1, 1]$.

– The forward pass of the Generator can be described in the following steps:
* Step 1: The input latent vector z (of size `latent_dim`) is passed through the fully connected (`Linear`) layer, producing a flattened vector of size $128 \times 7 \times 7$.
* Step 2: The flattened vector is reshaped into a 4D tensor of shape (`batch_size`, $128, 7, 7$), representing feature maps.
* Step 3: The reshaped tensor is passed through the transposed convolution layers (`ConvTranspose2d`) in sequence:
· In the first transposed convolution, the spatial resolution is doubled from 7×7 to 14×14. The feature maps are normalized (`BatchNorm2d`) and activated using `ReLU`.
· In the second transposed convolution, the resolution is further increased from 14×14 to 28×28, with similar normalization and activation applied.
* Step 4: The final convolution (`Conv2d`) reduces the number of channels to 1, resulting in a grayscale image of size 28×28.
* Step 5: The `Tanh` activation function is applied to constrain the pixel values to the range $[-1, 1]$, producing the final generated image.

– Overall, the Generator maps the latent space to the image space by progressively upsampling and refining the features.

- **Discriminator:**

– The Discriminator evaluates the validity of an image (real or fake) using a series of convolutional layers.
– The architecture is structured as follows:
* `Conv2d(1, 64, kernel_size=4, stride=2, padding=1)`: Downsamples the input image from 28×28 to 14×14.
* `LeakyReLU(0.2)`: Introduces non-linearity with a small slope for negative values.
* `Conv2d(64, 128, kernel_size=4, stride=2, padding=1)`: Further downsamples to 7×7.
* `BatchNorm2d(128)`: Normalizes intermediate outputs.
* `LeakyReLU(0.2)`: Activation function.
* `Flatten`: Flattens the output into a vector.
* `Linear(128 x 7 x 7, 1)`: Maps the flattened vector to a single validity score.
* `Sigmoid`: Constrains the validity score to the range [0, 1].

– The forward pass computes the validity score for the input image.

- **Training Setup:**

– The training follows the adversarial process:
* The Generator aims to produce realistic images that can fool the Discriminator.
* The Discriminator learns to differentiate real images from fake images.

– The loss function used is Binary Cross-Entropy (BCE) loss for both the Generator and the Discriminator.

- During each training iteration:
  - The Discriminator is updated multiple times for every update of the Generator. This allows the Discriminator to learn a more stable decision boundary, ensuring it does not become too weak compared to the Generator.
  - The Generator is updated after the Discriminator has sufficiently trained, ensuring that the Generator focuses on improving image quality relative to a strong Discriminator.
- The models are trained alternately to minimize their respective losses, improving the overall quality of generated images over time.

3 Experiment

3.1 Environment Setup

- **Model:** Ubuntu 22.04 Server
- **GPU:** 8 × NVIDIA GeForce RTX 4090
- **CPU:** AMD EPYC 7542 32-Core Processor
  - **Cores:** 64 cores per socket, 2 sockets (128 threads total)
  - **Frequency:** 1.5 GHz (base) to 2.9 GHz (max)
  - **Caches:** L1: 2 MiB, L2: 32 MiB, L3: 256 MiB
  - **NUMA Nodes:** 2 (node0: CPUs 0-31, 64-95; node1: CPUs 32-63, 96-127)

3.2 Part 1

3.2.1 Experiment Setup

The configuration for the experiment is as follows:

- **Input Length:** T
- **Input Dimension:** 10
- **Number of Classes:** 10
- **Number of Hidden Units:** 128
- **Batch Size:** 128
- **Learning Rate:** 0.001
- **Maximum Epochs:** 30
- **Gradient Clipping (Max Norm):** 10.0
- **Dataset Size:** 100,000 samples
- **Training Data Portion:** 80%

3.2.2 LSTM Result

All results are provided in Appendix C, including the performance of both RNN and LSTM models for sequence lengths $T = 5, 10, 15, 20, 25, 30$.

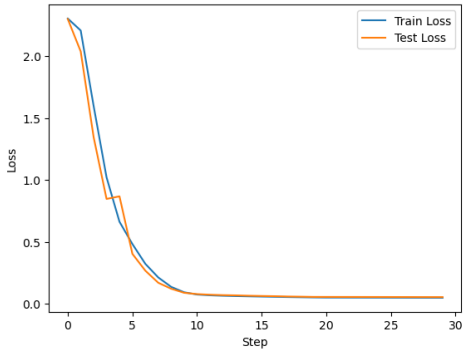


Figure 1: **Training and Testing Loss of LSTM Model (T=5)**

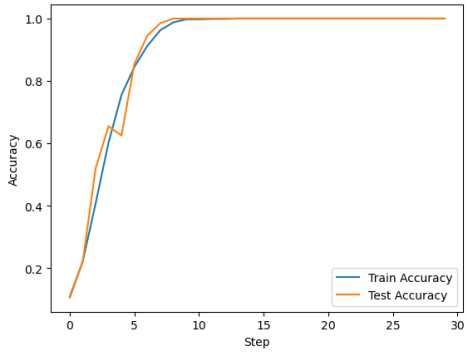


Figure 2: **Training and Testing Accuracy of LSTM Model (T=5)**

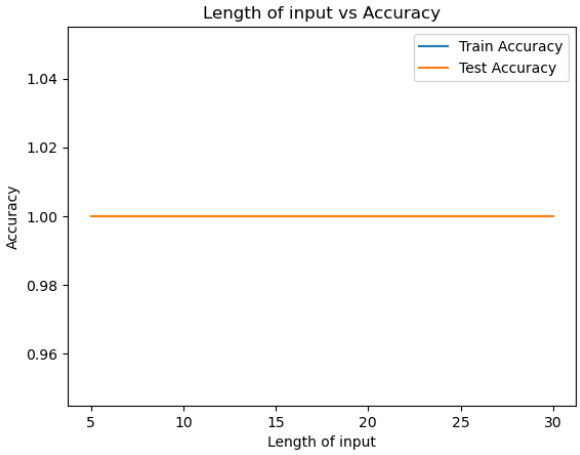


Figure 3: **Sequence Length vs. Accuracy of LSTM Model**

3.3 Part 2

3.3.1 Experiment Setup

The configuration for the experiment is as follows:

- **Number of Epochs:** 200
- **Batch Size:** 64
- **Learning Rate:** 0.0002
- **Latent Space Dimensionality:** 100
- **Save Interval:** Every 500 iterations
- **Discriminator Steps per Generator Step:** 5

3.3.2 Simple GAN

start of training

The result is illustrated in Figure 7

halfway through training

The result is illustrated in Figure 8

End of training

The result is illustrated in Figure 9

Interpolation Result

The result is illustrated in Figure 4

Figure 4: **Interpolation between 7 and 9 of Simple Gan**

3.3.3 DCGAN

start of training

The result is illustrated in Figure 10

halfway through training

The result is illustrated in Figure 11

End of training

The result is illustrated in Figure 12

Interpolation Result

The result is illustrated in Figure 5

Figure 5: **Interpolation between 1 and 8 of DCGan**

4 Analysis of Results

4.1 Part 1

4.1.1 Analysis

Using the default parameters provided, the LSTM model achieves exceptional performance on the palindrome sequence prediction task. As shown in Figure 3, both the training and test accuracy remain consistently close to 1.0 (100%) across all tested sequence lengths, ranging from $T = 5$ to $T = 30$. This demonstrates the LSTM's ability to accurately predict the T-th digit of a palindrome sequence based on the preceding $T - 1$ digits, regardless of the sequence length.

The superior performance of the LSTM model can be attributed to its ability to effectively capture and retain long-range dependencies in the input sequences. Unlike standard RNNs, which often encounter the vanishing gradient problem when processing long sequences, the LSTM's gated architecture (comprising input, forget, and output gates) enables it to maintain important information over extended sequences. This capability is particularly crucial for palindrome sequence prediction, as the task requires the model to recognize patterns and dependencies over the entire input sequence.

The consistent high accuracy achieved using the default parameters suggests that the LSTM architecture and hyperparameter choices are well-suited for this task without the need for extensive parameter tuning. This is noteworthy, as it highlights the robustness of LSTM for sequence-to-sequence

learning tasks. However, the uniformly excellent performance across all sequence lengths also raises concerns about potential overfitting. Since the dataset for this task may not be sufficiently complex or diverse, the model's high accuracy might reflect its ability to memorize patterns rather than generalize effectively.

Overall, the results align with the expectations outlined in the task description and confirm the advantages of LSTM over standard RNNs in handling long-term dependencies.

4.1.2 Compare LSTM and RNN

Figure 6 provides a comparison of the performance of LSTM and RNN on the palindrome sequence prediction task as the sequence length increases. Several critical observations can be drawn from the results:

1. **Performance with Shorter Sequences:** For shorter sequences ($T \leq 10$), both LSTM and RNN achieve high accuracy on training and validation datasets. At this stage, the simpler architecture of RNN is sufficient to model the dependencies in the sequence, and no significant difference in performance is observed between the two models.

2. **Accuracy Decline with Increasing Sequence Length:** As the sequence length increases ($T > 10$), the accuracy of RNN begins to drop noticeably, especially for validation data. In contrast, LSTM maintains a high level of accuracy for both training and validation datasets up to sequence lengths of $T = 20$. This difference arises due to the fundamental architectural improvements in LSTM, which include gates (input, forget, and output gates) that allow it to effectively capture and retain long-range dependencies. On the other hand, RNN struggles with the vanishing gradient problem, leading to a loss of important information as the sequence length grows, thereby resulting in degraded performance.

3. **Overfitting in RNN:** A clear gap between the training and validation accuracy is observed for RNN, particularly for sequence lengths in the range $T = 10$ to 20. This indicates that RNN is overfitting the training data while failing to generalize well to the validation dataset. LSTM, however, shows a much smaller gap between training and validation accuracy, demonstrating better generalization capabilities.

4. **Performance with Longer Sequences:** For very long sequences ($T \geq 25$), both LSTM and RNN experience a decline in accuracy. However, the decline is much steeper for RNN, with both its training and validation accuracy dropping below 75% at $T = 30$. LSTM, on the other hand, retains higher accuracy levels even for $T = 30$, demonstrating its robustness in handling tasks that require learning long-term dependencies.

5. **Implications of the Results:** The results confirm the superior performance of LSTM over RNN for tasks involving long-range dependencies, such as the palindrome sequence prediction task. LSTM's gated architecture enables it to mitigate the vanishing gradient problem and retain critical information across longer sequences, whereas RNN fails to do so effectively. This makes LSTM a more reliable choice for sequence-to-sequence tasks, especially when sequence lengths are large.

In summary, the comparison highlights the advantages of LSTM over RNN in terms of robustness, generalization ability, and the capacity to handle longer sequences. These characteristics make LSTM a preferred architecture for tasks that require modeling complex sequential data with long-term dependencies.

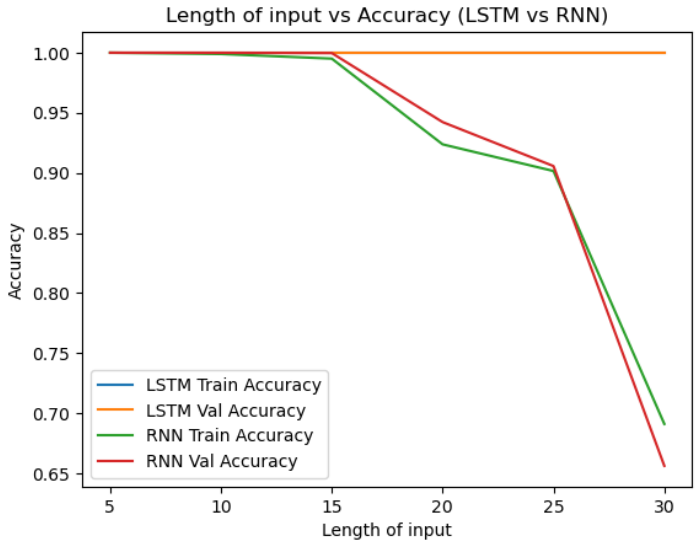


Figure 6: **LSTM vs RNN**

4.2 Part 2

4.2.1 Analysis in Different Stage

Both GAN and DCGAN share similar training structures, but the use of convolutional layers in DCGAN allows it to capture features more quickly and effectively. The differences in image quality and structure at different training stages are outlined below.

1. Early Training (Initial Stage) - **GAN:** In the early stages, the Generator produces random noise with no recognizable structure. The generated images are highly distorted, with no clear patterns or digits, as the model has not yet learned to map the latent vector to meaningful image features. The Discriminator easily distinguishes between real and fake images, providing valuable feedback to the Generator. - **DCGAN:** Due to the convolutional layers in the Generator, DCGAN is able to start capturing some basic features such as rough outlines or shapes. While the images still appear noisy, the early learning process leads to slightly more recognizable structures compared to GAN, as the convolution layers help the Generator form basic patterns of the digits.

2. Mid Training (Intermediate Stage) - **GAN:** As training progresses, the Generator begins to learn to create partially recognizable digits. The images start to show basic shapes of digits but are often blurry or incomplete. The Generator is able to produce more coherent images, but the results are still imperfect. The Discriminator can still differentiate between real and fake images, but with more difficulty. - **DCGAN:** DCGAN shows noticeable improvement by this stage. The convolutional layers allow the Generator to produce clearer and sharper features of digits. The images become more recognizable, with fewer distortions and sharper outlines compared to GAN, as the convolution layers improve the spatial hierarchy of the features generated.

3. Late Training (Final Stage) - **GAN:** In the final stages of training, the Generator has learned to produce images that closely resemble the real data distribution. The generated digits are well-formed, with minimal noise or artifacts. The images are much sharper and more coherent, showing little to no blur. The Discriminator struggles to differentiate between real and fake images as the Generator produces high-quality, realistic images. - **DCGAN:** DCGAN, owing to its convolutional architecture, generates images that are sharp, coherent, and highly realistic even more quickly than GAN. The generated digits are clearer and more precise, with the overall quality surpassing GAN's output at a faster rate due to the improved representation capabilities of the convolutional layers.

4.2.2 Compare GAN and DCGAN

In this section, we analyze the results of two generative models applied to generating handwritten digit images from the MNIST dataset: Simple GAN and DCGAN. We will discuss the experimental results, compare their performance, and highlight key differences in their architectures and training behaviors.

1. Results

- **Simple GAN:** The Simple GAN model, composed of a basic fully connected architecture, is capable of generating plausible handwritten digit images from the MNIST dataset. However, the generated images exhibit noticeable artifacts and are generally less sharp compared to the real MNIST digits. This is primarily due to the limited expressiveness of fully connected layers, which lack the ability to model the spatial structure of images, thus restricting the model's capacity to capture fine-grained image details.
- **DCGAN:** The DCGAN model significantly improves the quality of the generated images by using convolutional layers instead of fully connected ones. By employing transposed convolutions, DCGAN better preserves spatial information and is able to capture the structure of the digits more effectively. The generated images are clearer, more coherent, and appear much closer to real MNIST digits than those produced by Simple GAN.

2. Key Differences

- **Architecture:** Simple GAN uses fully connected layers, resulting in flat, unstructured images. While this approach is simpler and computationally less expensive, it lacks the capacity to learn the hierarchical structure of images, leading to poorer generation results. In contrast, DCGAN uses convolutional layers, which are better suited for image data. The transposed convolutions allow the generator to progressively upsample the latent vector into a high-resolution image, while the convolutional layers in the discriminator effectively capture spatial features and distinguish between real and fake images.
- **Training Stability:** Training GANs can be challenging due to their adversarial nature. Simple GAN tends to be less stable during training because the discriminator can become too strong initially, making it difficult for the generator to improve. DCGAN, on the other hand, benefits from techniques like batch normalization, which help stabilize training. The use of convolutional layers further contributes to stable training by making the model less prone to issues like mode collapse, allowing for more consistent improvements in image quality.
- **Image Quality:** DCGAN outperforms Simple GAN in terms of image quality. The convolutional architecture of DCGAN allows the model to better capture spatial relationships in the image, leading to sharper and more realistic outputs. Simple GAN-generated images, on the other hand, are often blurry and distorted. The batch normalization in DCGAN also helps mitigate problems such as mode collapse, ensuring more diverse and realistic images.

3. Training Process and Challenges

- Both models use Binary Cross-Entropy (BCE) loss during training. However, due to the more complex architecture of DCGAN, the training process is less susceptible to problems like vanishing gradients. In Simple GAN, the lack of spatial awareness in the architecture makes it difficult for the discriminator to distinguish real from fake images, which slows down training and results in poorer image quality.
- In DCGAN, the discriminator is better at distinguishing real images from generated ones, which helps the generator improve more quickly. This adversarial dynamic leads to more stable training, with the generated images steadily improving as training progresses.

4. Comparison Summary

- **Simple GAN:**
  - Uses fully connected layers, resulting in blurry and distorted images.
  - Less stable during training, with lower image quality.
  - Suitable for simpler tasks or when computational resources are limited.
- **DCGAN:**
  - Generates high-quality images with clearer details and better structure.
  - Training process is more stable due to the use of convolutional layers and batch normalization.
  - Better at generating realistic and coherent images, making it a superior choice for image generation tasks.

4.2.3 Interpolate Between Two Digits

This experiment evaluates the ability of GAN models to perform smooth transitions between two different digit classes in the latent space. Two images from distinct classes were sampled as the starting and ending points, and a linear interpolation was conducted with 7 steps in the latent space, resulting in 9 images, including the start and end points.

Figure 4: Simple GAN Interpolation Results (Digits 7 and 9)

Generation Quality: The interpolation between the digits "7" and "9" using Simple GAN shows distinct starting and ending images. The digit "7" and digit "9" at the endpoints are recognizable and have clear features. However, the intermediate steps reveal significant limitations. Images in frames 4 to 6 display unclear shapes and features that lack resemblance to either "7" or "9". The transition between the two digit classes appears abrupt and fails to smoothly blend the characteristics of the digits.

Latent Space Structure: The results suggest that the latent space learned by Simple GAN may not be well-structured or continuous. Instead of capturing a gradual progression from one digit class to another, the model seems to produce artifacts or ambiguous representations in the intermediate steps. This indicates that the latent space transitions in Simple GAN may be poorly aligned with semantic changes between classes.

Feature Representation: The inability of Simple GAN to preserve key features of "7" and "9" in the intermediate steps reflects a lack of consistency in the model's representation of digit characteristics. This inconsistency limits the model's capacity to interpolate smoothly and highlights gaps in its latent space learning.

Figure 5: DCGAN Interpolation Results (Digits 1 and 8)

Generation Quality: In contrast to Simple GAN, the interpolation results from DCGAN exhibit significantly smoother transitions between the digits "1" and "8". The starting image for "1" and the ending image for "8" are clear and well-defined, while the intermediate steps progressively combine the features of the two digits. For instance, in frames 4 to 6, the straight-line structure of "1" gradually transforms into the loop-like structure of "8". This smooth transition highlights the model's ability to capture and blend features effectively.

Latent Space Structure: The results demonstrate that DCGAN has learned a more continuous and structured latent space. The interpolation path in the latent space aligns closely with meaningful semantic changes, enabling the model to generate coherent intermediate representations. This indicates that DCGAN effectively maps latent vectors to corresponding digit classes with well-preserved relationships between them.

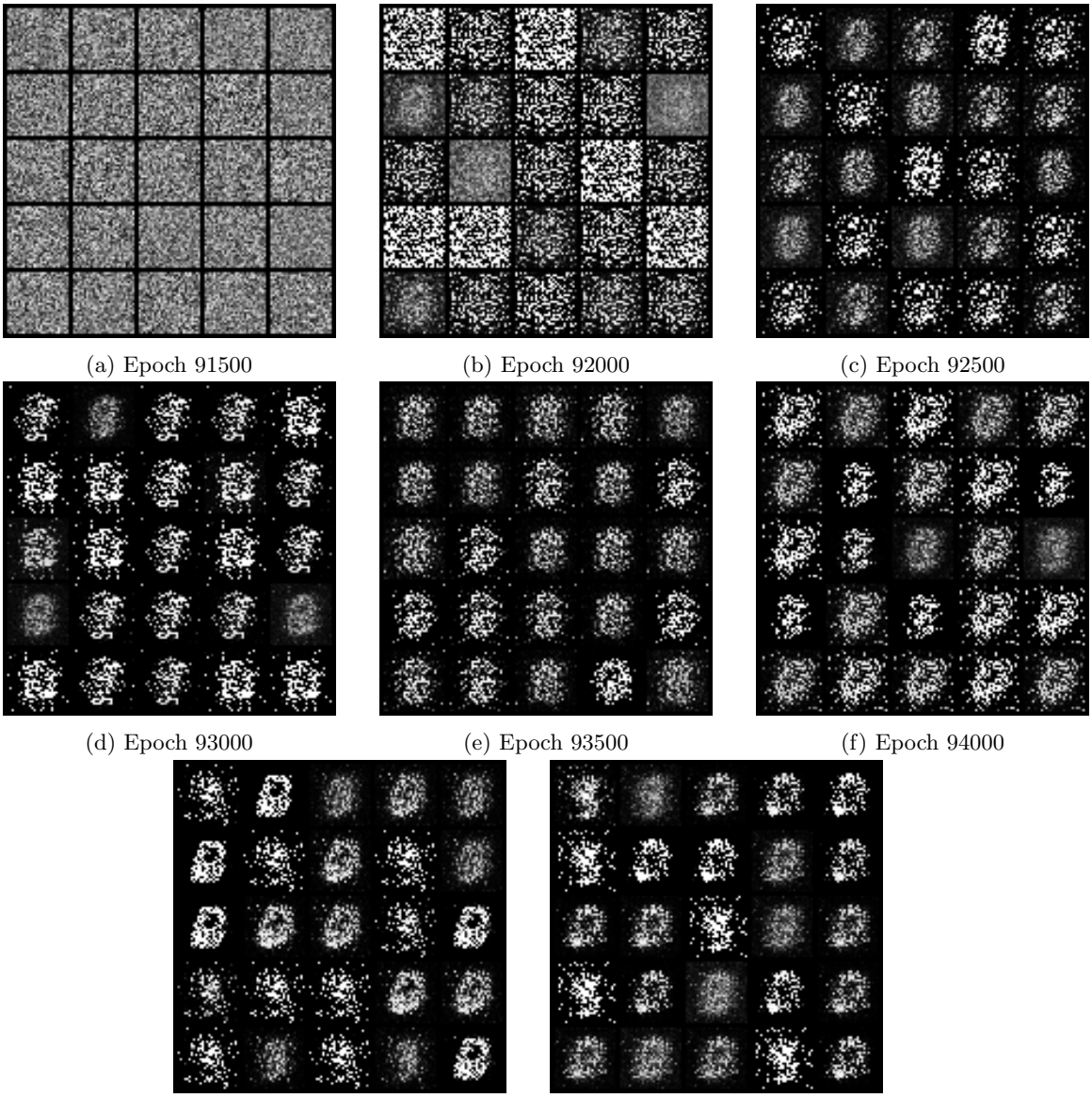
Feature Representation: The gradual blending of characteristics in DCGAN's interpolation results highlights its ability to represent digit features consistently across the latent space. Unlike Simple GAN, DCGAN maintains class-specific details throughout the interpolation process, resulting in clear and distinguishable intermediate images.

Comparison and Insights

- **Latent Space Continuity:** DCGAN clearly outperforms Simple GAN in terms of latent space continuity. While Simple GAN produces abrupt and poorly defined transitions, DCGAN delivers smooth and coherent progressions between the two digit classes. This indicates a more robust latent space representation in DCGAN.
- **Feature Preservation:** The interpolation results reveal that DCGAN captures and preserves the features of the digits more effectively than Simple GAN. The intermediate images generated by DCGAN retain meaningful characteristics from both starting and ending digit classes, whereas Simple GAN fails to maintain such consistency.
- **Overall Performance:** The experiment highlights a critical distinction between the two models. Simple GAN demonstrates a lack of structural alignment in its latent space, leading to ambiguous and low-quality interpolations. On the other hand, DCGAN achieves superior interpolation results by learning a latent space that reflects meaningful semantic relationships between digit classes.

These results underline the importance of a well-structured latent space in GAN models for generating high-quality and coherent transitions between different classes.

A Result of GAN

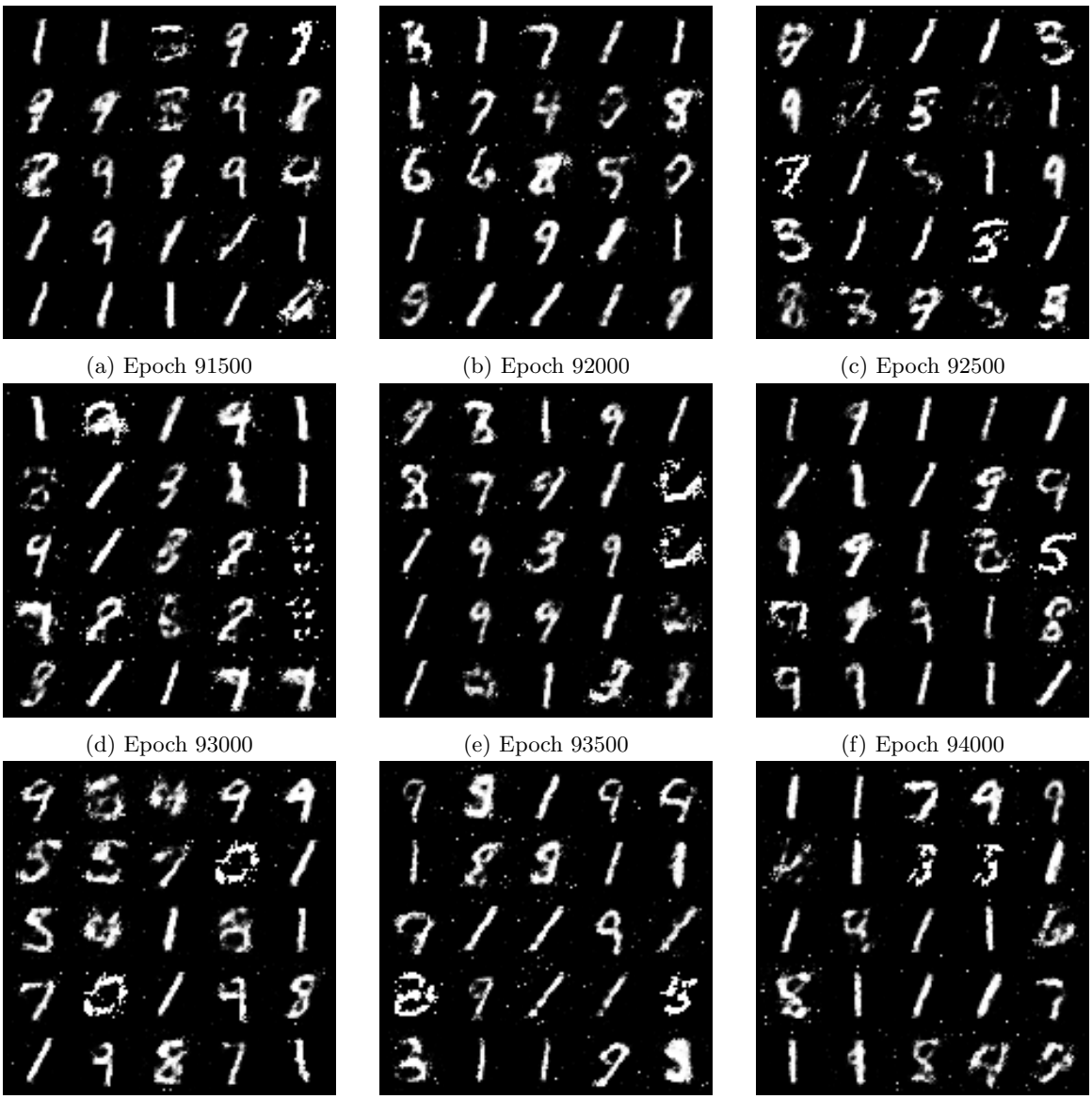


(a) Epoch 91500 (b) Epoch 92000 (c) Epoch 92500

(d) Epoch 93000 (e) Epoch 93500 (f) Epoch 94000

(g) Epoch 94500 (h) Epoch 95000

Figure 7: Beginning through training of Simple GAN

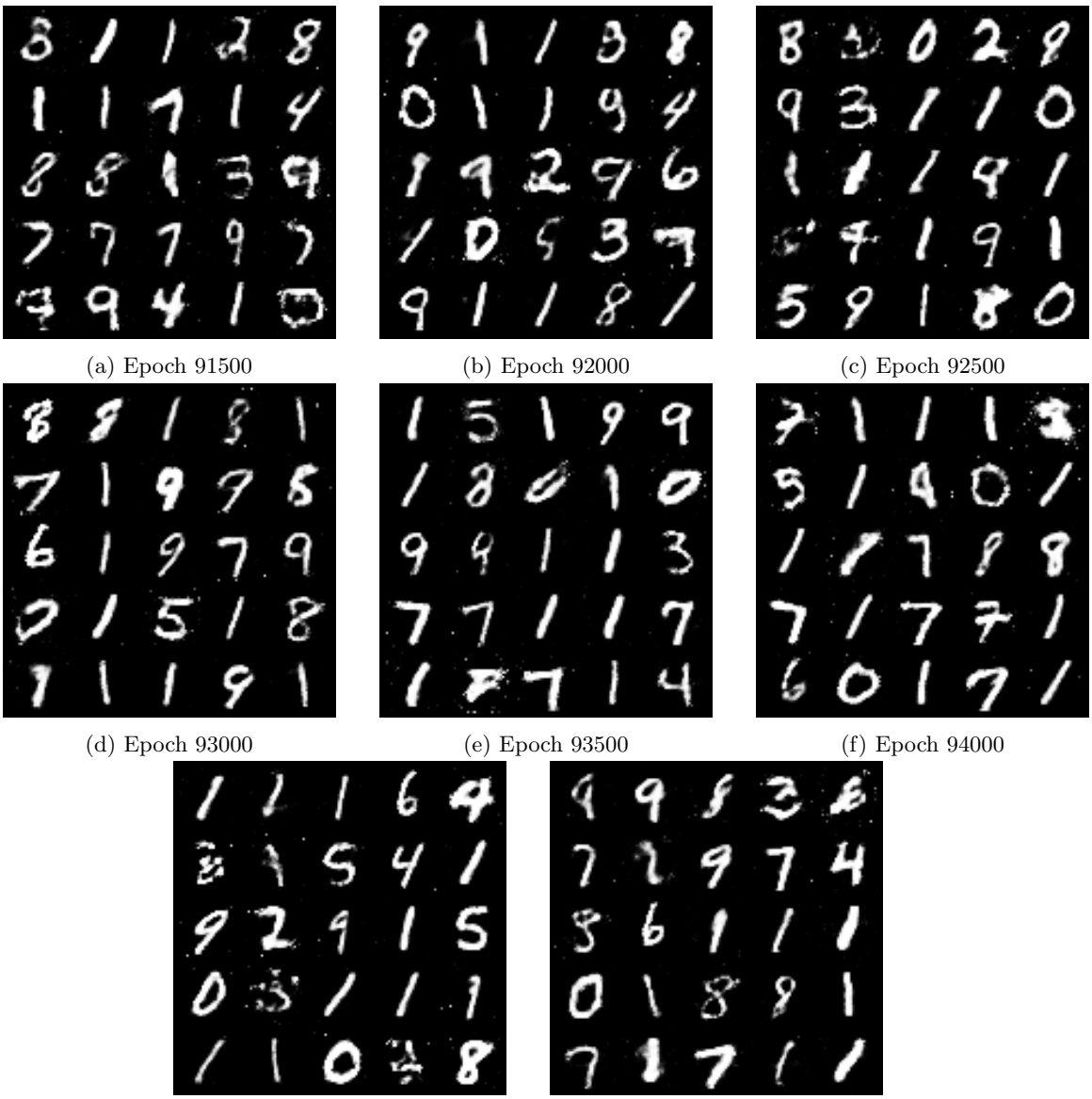


(a) Epoch 91500 (b) Epoch 92000 (c) Epoch 92500

(d) Epoch 93000 (e) Epoch 93500 (f) Epoch 94000

(g) Epoch 94500 (h) Epoch 95000 (i) Epoch 95500

Figure 8: Halfway through training of Simple GAN



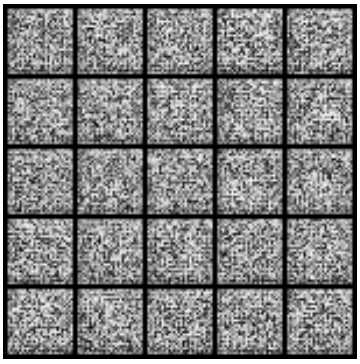
(a) Epoch 91500 (b) Epoch 92000 (c) Epoch 92500

(d) Epoch 93000 (e) Epoch 93500 (f) Epoch 94000

(g) Epoch 94500 (h) Epoch 95000

Figure 9: End through training of Simple GAN

B Result of DCGAN



(a) Epoch 91500

(b) Epoch 92000

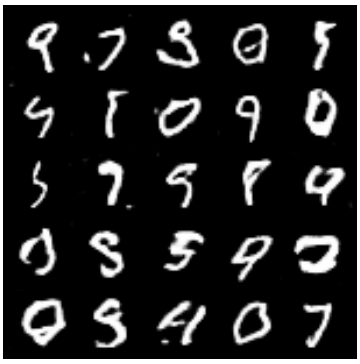
(c) Epoch 92500

(d) Epoch 93000

(e) Epoch 93500

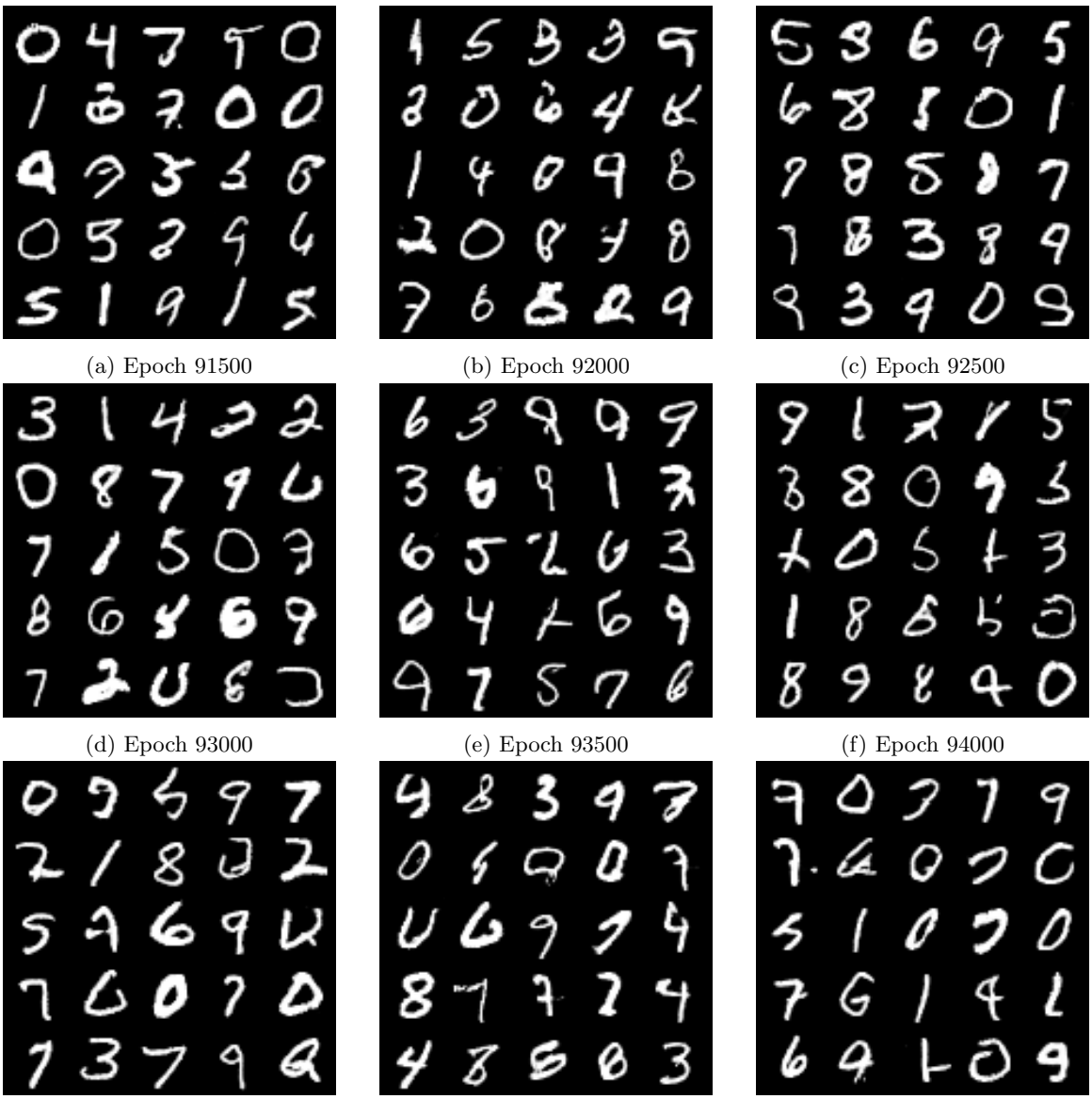
(f) Epoch 94000

(g) Epoch 94500



(h) Epoch 95000

Figure 10: Beginning through training of DCGAN



(a) Epoch 91500 (b) Epoch 92000 (c) Epoch 92500

(d) Epoch 93000 (e) Epoch 93500 (f) Epoch 94000

(g) Epoch 94500 (h) Epoch 95000 (i) Epoch 95500

Figure 11: Halfway through training of DCGAN

(a) Epoch 91500

(b) Epoch 92000

(c) Epoch 92500

(d) Epoch 93000

(e) Epoch 93500

(f) Epoch 94000

(g) Epoch 94500

(h) Epoch 95000

Figure 12: End through training of DCGAN

C Detail Results of LSTM and RNN

C.1 T = 5

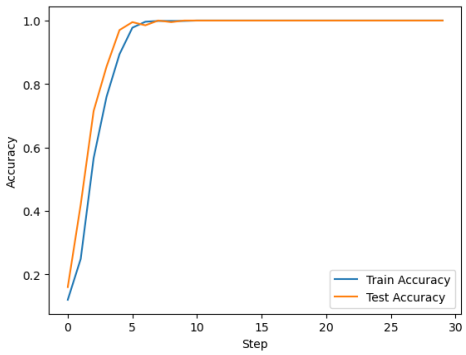


Figure 13: **Training and Testing Accuracy of RNN Model (T=5)**

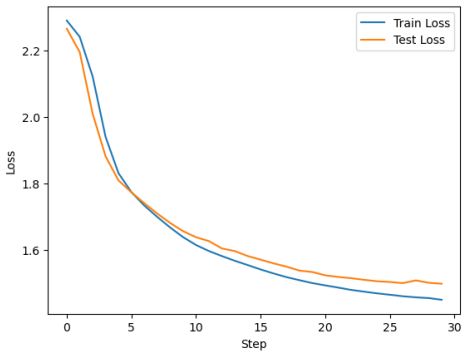


Figure 14: **Training and Testing Loss of RNN Model (T=5)**

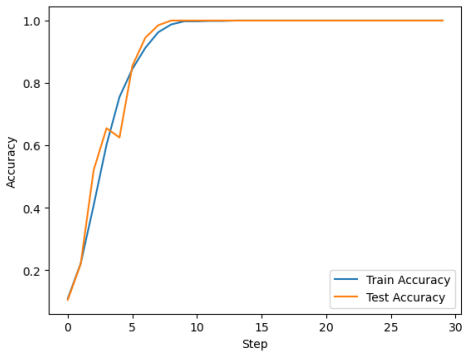


Figure 15: **Training and Testing Accuracy of LSTM Model (T=5)**

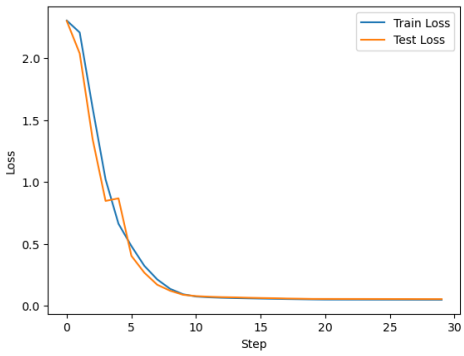


Figure 16: **Training and Testing Loss of LSTM Model (T=5)**

C.2 T = 10

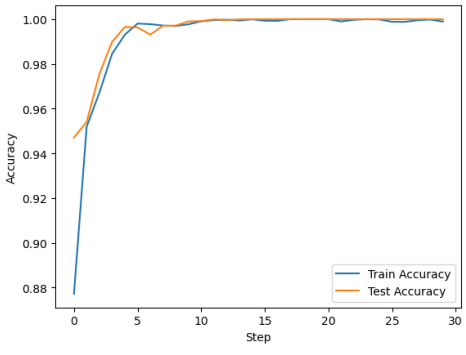


Figure 17: **Training and Testing Accuracy of RNN Model (T=10)**

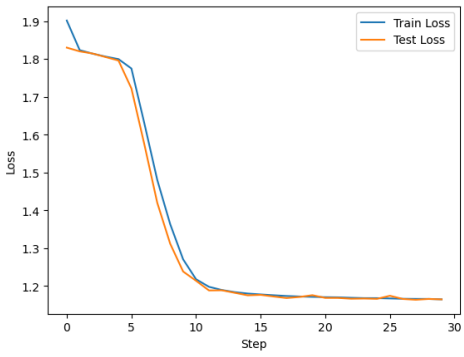


Figure 18: **Training and Testing Loss of RNN Model (T=10)**

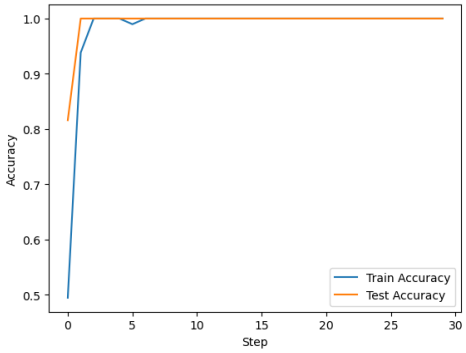


Figure 19: **Training and Testing Accuracy of LSTM Model (T=10)**

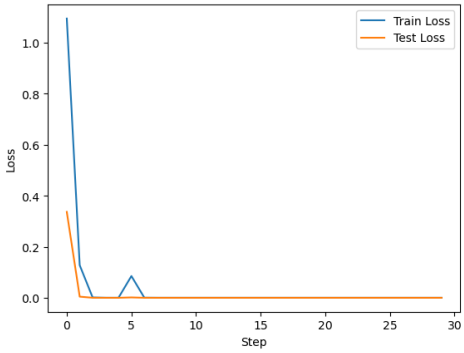


Figure 20: **Training and Testing Loss of LSTM Model (T=10)**

C.3 T = 15

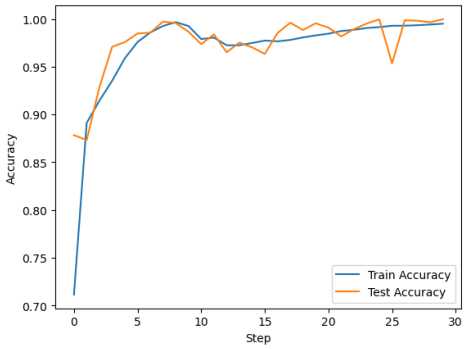


Figure 21: **Training and Testing Accuracy of RNN Model (T=15)**

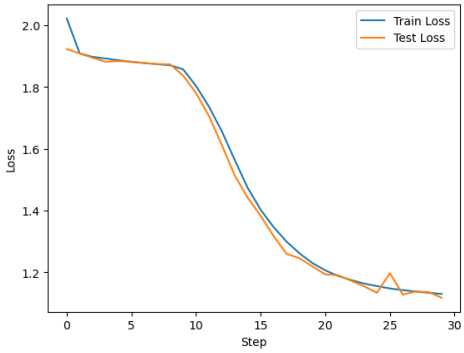


Figure 22: **Training and Testing Loss of RNN Model (T=15)**

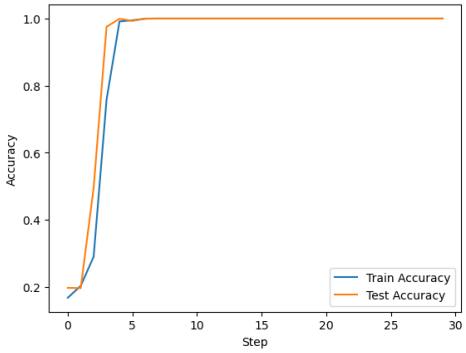


Figure 23: **Training and Testing Accuracy of LSTM Model (T=15)**

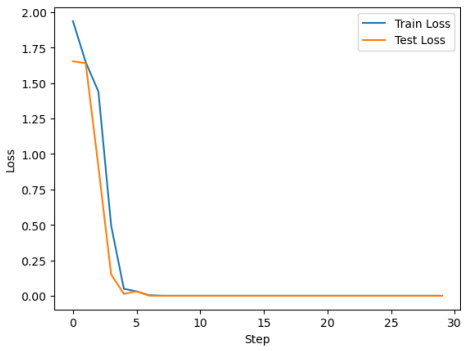


Figure 24: **Training and Testing Loss of LSTM Model (T=15)**

C.4 T = 20

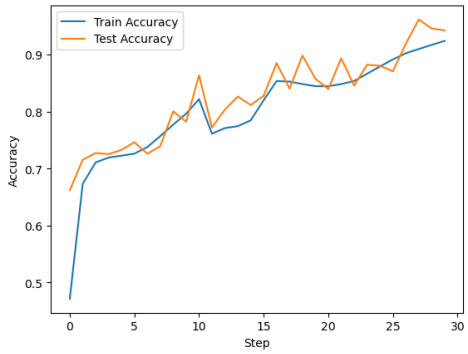


Figure 25: **Training and Testing Accuracy of RNN Model (T=20)**

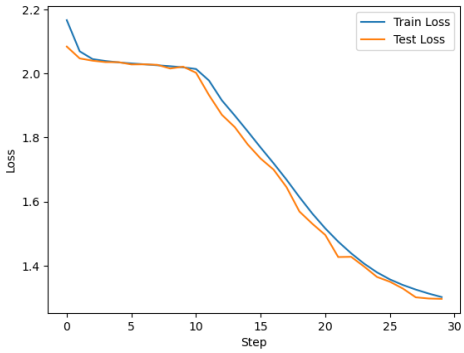


Figure 26: **Training and Testing Loss of RNN Model (T=20)**

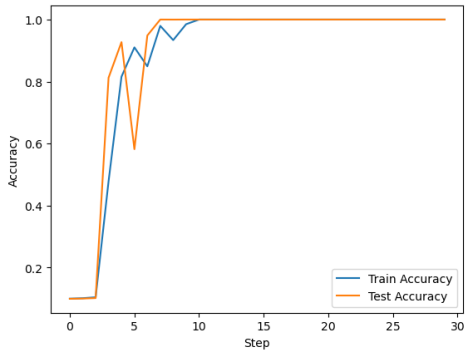


Figure 27: **Training and Testing Accuracy of LSTM Model (T=20)**

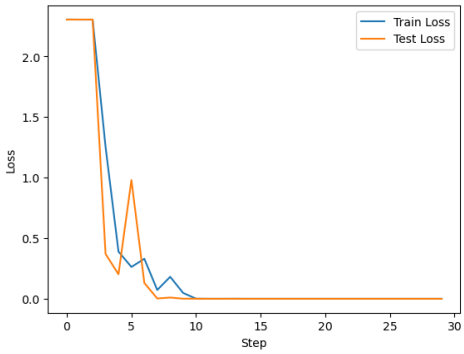


Figure 28: **Training and Testing Loss of LSTM Model (T=20)**

C.5 T = 25

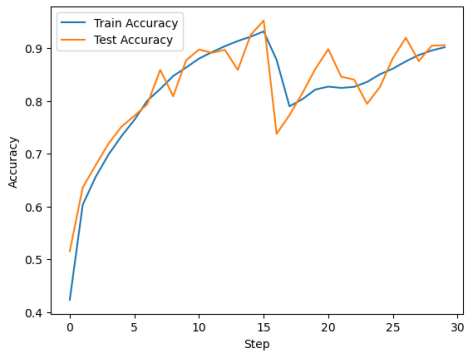


Figure 29: **Training and Testing Accuracy of RNN Model (T=25)**

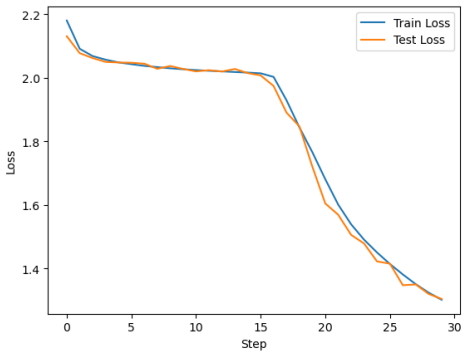


Figure 30: **Training and Testing Loss of RNN Model (T=25)**

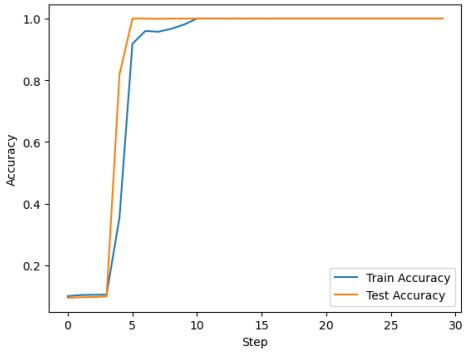


Figure 31: **Training and Testing Accuracy of LSTM Model (T=25)**

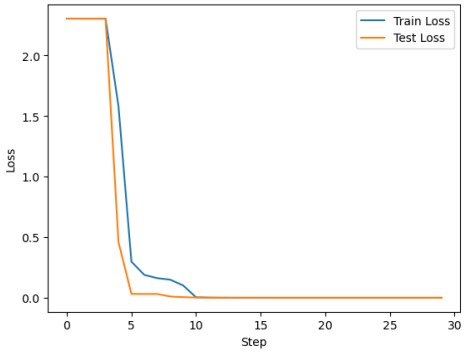


Figure 32: **Training and Testing Loss of LSTM Model (T=25)**

C.6 T = 30

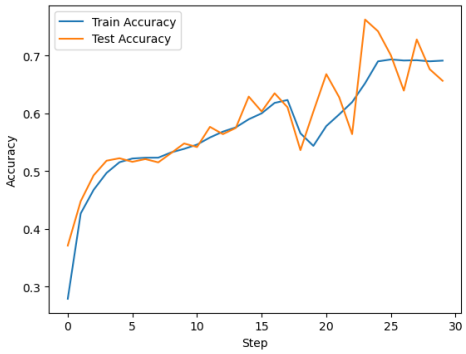


Figure 33: **Training and Testing Accuracy of RNN Model (T=30)**

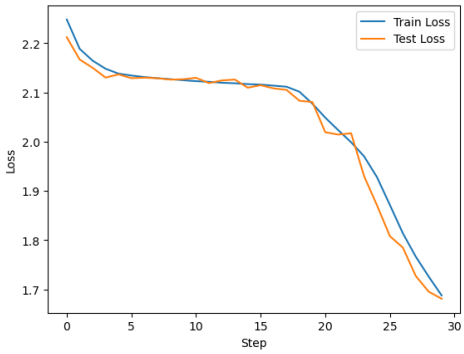


Figure 34: **Training and Testing Loss of RNN Model (T=30)**

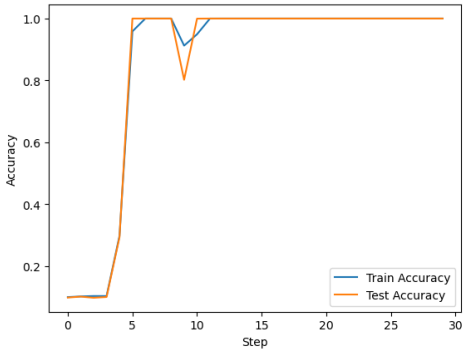


Figure 35: **Training and Testing Accuracy of LSTM Model (T=30)**

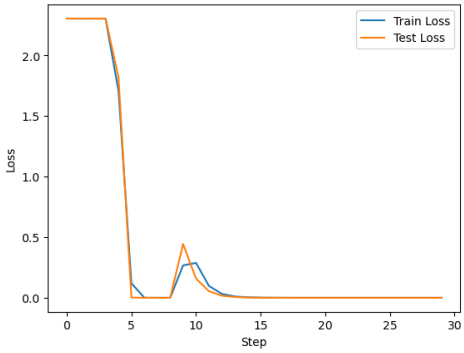


Figure 36: **Training and Testing Loss of LSTM Model (T=30)**