CS324 Assignment 3

Sihua Zhang
12410106@mail.sustech.edu.cn

***Abstract*—This assignment focuses on the implementation and evaluation of Long-Short Term Memory (LSTM) networks and Generative Adversarial Networks (GANs). The goal is to understand the strengths and weaknesses of different architectures and training techniques through hands-on experimentation and analysis.**

I. PyTorch LSTM

In this part we impletment a pytorch LSTM network to predict the last number of a palindrome sequence given all previous numbers. Formally, given a sequence of numbers $x_1, x_2, ..., x_{t-1}$, we want to predict x_t based on $x_1, x_2, ..., x_{t-1}$. The sequences are palindromic, meaning that they read the same forwards and backwards.

Compare to traditional RNNs, LSTMs are better at capturing long-term dependencies in sequences due to their unique architecture, which includes memory cells and gating mechanisms. This makes them particularly well-suited for tasks like sequence prediction, where the model needs to remember information from earlier in the sequence to make accurate predictions later on.

A. Task1

We implement a single-layer LSTM network using PyTorch bold (without torch.nn.LSTM) to predict the last number of a palindrome sequence. The network consists of an input gate, a forget gate, an output gate to achieve gating mechanisms and a cell gate to renew cell state, followed by a fully connected layer that maps the hidden state to the output. Essentially all gates mentioned above are implemented using linear layers and sigmoid functions. We use a for loop to step through time, and use RMSprop optimizer to optimize the model parameters as is requied. The loss function is cross-entropy loss. Besides, we use xavier initialization to initialize the weights of linear layers to help with convergence.

B. Task2

We train the LSTM network on palindrome sequences of length 20 (including the target number) for 5 epochs with a batch size of 128 and initial learning rate of 0.001. We adopt CosineAnnealingLR as the learning rate scheduler to adjust the learning rate during training. On a MacBook Air with M4 chip with mps backend, the training process takes less than 1 minutes. Our LSTM easily achieves 100% accuracy on the validation set after training for 3 epochs, which is sighnificantly more efficient than the traditional RNN we implemented in Assignment 2. The training and validation loss and accuracy curves are shown in Fig. 1.

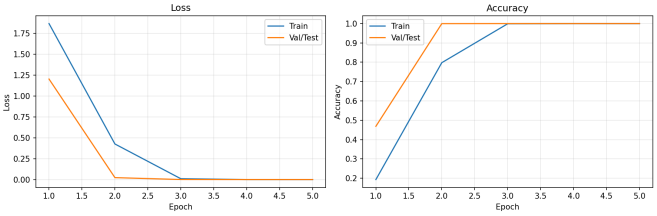

Fig. 1. Training and Validation Loss and Accuracy Curves for LSTM on Palindrome Sequences of Length 20

II. Generative Adversarial Networks

In this part we implement a Generative Adversarial Network (GAN) using PyTorch to generate images similar to those in the MNIST dataset. The GAN consists of two main components: a generator and a discriminator. The generator takes random noise as input and generates fake images, while the discriminator takes both real images from the MNIST dataset and fake images from the generator as input and tries to distinguish between them. The two networks are trained simultaneously in a minimax game, where the generator aims to produce images that can fool the discriminator, and the discriminator aims to correctly classify real and fake images. The objective function $\min_G \max_D V(D, G) =$ is defined as:

$$\mathbb{E}_{x \sim p_{\text{data}(x)}}[\log D(x)] + \mathbb{E}_{z \sim p_{z(z)}}[\log(1 - D(G(z)))] \quad (1)$$

After training, the generator should be able to produce images that closely resemble the real MNIST images.

A. Task 1

We implement the GAN using PyTorch. The generator network consists of several linear layers to upsample the input noise vector into a 28x28 image, with batch normalization and LeakyReLU activations in between. The final layer uses a Tanh activation to output pixel values in the range [−1, 1]. The discriminator network consists of several linear layers to downsample the input image, with LeakyReLU activations. We use the Binary Cross-Entropy loss (with logits) function for both networks and the Adam optimizer for training.

The loss functions for the generator and discriminator are defined as follows:

- Discriminator Loss:

$$L_D = -\log D(x) - \log(1 - D(G(z))) \quad (2)$$

- Generator Loss:

$$L_G = -\log D(G(z)) \quad (3)$$

We train the GAN using mini-batch stochastic gradient descent, alternating between updating the discriminator and

generator. This is equilavent to minimax the objective function defined above. The accuracy of the discriminator and training loss curves for both the generator and discriminator are shown in Fig. 2. The initial accuracy is almost 100%, and generator loss raise up to 8 in the first 15 epochs, indicating that the task is too easy for discriminator to distinguish from real pictures and random noises. This causes small gradient for generator to learn. Generator Loss rapiddly reduce around epoch 25 from 8 to 2, while discriminator accuracy drop from nearly 100% to around 80%. After 50 epochs, we can see that the generator loss converges and the discriminator accuracy remains around 80%. According to Nash Equilibrium theory, final accuracy should be around 50%, indicating that the discriminator is overpowered compared to the generator, and the generator cannot produce more realistic images.

To improve the expressive power of the generator and stabilize adversarial training, we replace the original multilayer perceptron (MLP)-based GAN with a Deep Convolutional GAN (DCGAN) architecture [1]. Unlike MLP-based generators, DCGAN employs convolutional and transposed convolutional layers, which better exploit the spatial structure inherent in image data through local receptive fields and weight sharing. This architectural inductive bias enables the generator to capture hierarchical visual patterns more effectively, leading to improved image fidelity and training stability. In addition, we apply Spectral Normalization (SN) to the discriminator to further stabilize training. Spectral Normalization constrains the Lipschitz constant of the discriminator by normalizing the spectral norm (i.e., the largest singular value) of each weight matrix. This prevents the discriminator from becoming overly sharp or dominant, thereby providing more informative and stable gradients to the generator during training. For optimization, we adopt the Adam optimizer for both the generator and discriminator, with learning rates tuned to balance their respective learning speeds and momentum parameters set to $b\eta_1 = 0.5$ and $b\eta_2 = 0.999$, following common practice in GAN training. These settings help reduce oscillatory behavior and promote smoother convergence. As shown in Fig. 3 ,the DCGAN exhibits more stable training dynamics compared to the MLP-based GAN. The generator loss decreases more steadily, while the discriminator accuracy converges toward 50% after approximately 200 epochs, indicating that neither network overwhelmingly dominates the adversarial game. This balance suggests that the generator is able to produce increasingly realistic samples that are difficult for the discriminator to distinguish from real images.

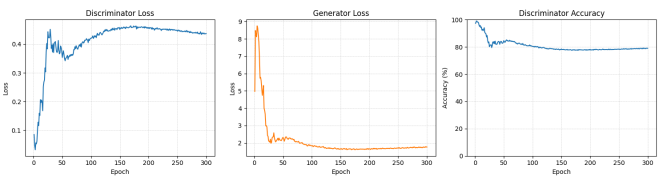

Fig. 2. Traning Curves for GAN

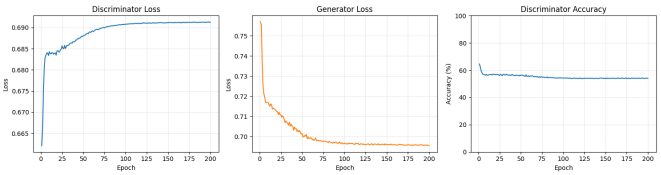

Fig. 3. Traning Curves for DCGAN

B. Task 2

We sample 25 images every 20 epochs during training to visualize the progress of the generator. Example result is shown in Fig. 4 and Fig. 5. We can observe that as training progresses, the quality of the generated images improves significantly, with the generator producing images that closely resemble real MNIST digits after 200 epochs. The DCGAN outperforms the MLP-based GAN in terms of image quality and diversity, generating clearer and more varied digits. This demonstrates the effectiveness of convolutional architectures and spectral normalization in enhancing GAN performance.

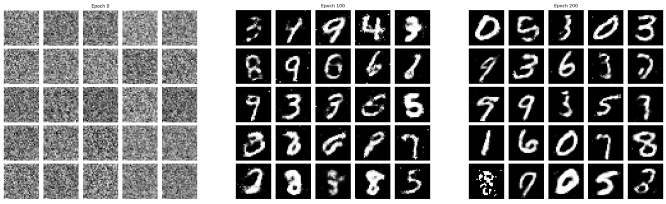

Fig. 4. Before Training (left), After 100 Epochs (middle), After 200 Epochs of GAN (right)

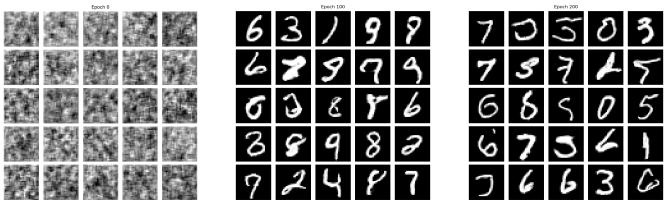

Fig. 5. Before Training (left), After 100 Epochs (middle), After 200 Epochs of GAN (right)

We provide a jupyter notebook to interactively visualize the generated images. All sampled images during training are also included in our code repository.

C. Task 3

In order to investigate whether our GAN is able to learn a continuous latent space, we perform linear interpolation in the latent space between two random noise vectors. Specifically, we generate two random noise vectors z_1 and z_2, and then create a series of intermediate vectors by linearly interpolating between them. Each intermediate vector is then fed into the generator to produce an image. The results are shown in Fig. 6. From left to right, we can observe a smooth transition between the two generated images, indicating that the GAN has successfully learned a continuous latent space where small changes in the input noise vector lead to gradual changes in the output image. This property is desirable as it

suggests that the generator has captured meaningful features of the data distribution.

Fig. 6. Interpolation in Latent Space between Two Random Noise Vectors

References

[1] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *CoRR*, 2015, [Online]. Available: https://api.semanticscholar.org/CorpusID:11758569