INDENG 290 - HomeWork 4

Loïc Jannin

November 2023

1 Introduction

The fundamental idea behind GANs involves a minimax game between these two neural networks. The generator aims to create synthetic data resembling the true data distribution, while the discriminator tries to distinguish between real and fake data. As training progresses, the generator learns to produce data that is indistinguishable from the real data, while the discriminator gets better at telling the difference.

This adversarial process results in the improvement of both models, with the generator continuously refining its output to deceive the discriminator, and the discriminator becoming more adept at distinguishing real from synthetic data.

The elegance of GANs lies in their ability to generate realistic data, making them applicable in various domains, including image generation, data augmentation, and more.

2 Two-dimensional sin data

In this section, we aim to generate two-dimensional data resembling a sinusoidal distribution. The input to our GAN is illustrated in the following distribution in figure 1. Starting from noise, we train our GAN to replicate this distribution.

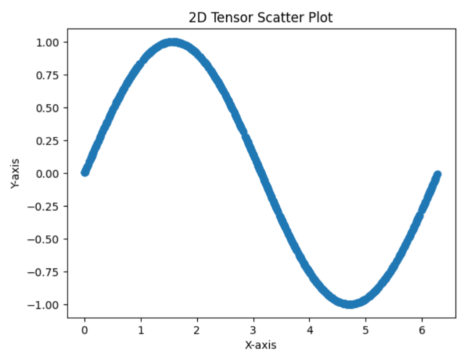


Figure 1: Input distribution

2.1 Neural Network Architecture

I began with a conventional architecture, employing one hidden layer with a dimension of 16 for each neural network. This choice served as a starting point, but the model exhibited poor performance. The generated data remained stagnant, clustering without learning the sinusoidal shape. Consequently, after conducting further tests and research, and considering the lengthy training time on my computer, I decided to adopt an architecture similar to that of the professor's.

2.1.1 Generator Architecture

The Generator is structured as follows:

- Input: Two-dimensional input
- **Layers:**
  - Fully connected layer (`nn.Linear`) with 2 input nodes and 16 output nodes followed by a ReLU activation function.
  - Fully connected layer with 16 input nodes and 32 output nodes followed by a ReLU activation function.
  - Fully connected layer with 32 input nodes and 2 output nodes (for a two-dimensional output).

2.1.2 Discriminator Architecture

The Discriminator is structured as follows:

- Input: Two-dimensional input
- **Layers:**
  - Fully connected layer (`nn.Linear`) with 2 input nodes and 256 output nodes followed by a ReLU activation function.
  - Dropout layer (`nn.Dropout`) with a dropout rate of 0.3 to reduce overfitting.
  - Fully connected layer with 256 input nodes and 128 output nodes followed by a ReLU activation function.
  - Dropout layer with a dropout rate of 0.3.
  - Fully connected layer with 128 input nodes and 64 output nodes followed by a ReLU activation function.
  - Dropout layer with a dropout rate of 0.3.
  - Fully connected layer with 64 input nodes and 1 output node followed by a sigmoid activation function to represent probabilities.

2.2 Data Preparation

Data were prepared in the same way as in the teacher's code :

The training dataset (`train_data`) consists of 1024 data points organized in 2 dimensions. To create this synthetic dataset using PyTorch, the first dimension (`train_data[:, 0]`) multiplies 2π with random values generated from a uniform distribution. The second dimension (`train_data[:, 1]`) represents the sine values of the first dimension.

The data is formatted into a DataLoader to facilitate efficient processing during training. A batch size of 32 is used, and the dataset is wrapped in a TensorDataset to ensure compatibility with the DataLoader.

2.3 Hyperparameters

After different tests, the hyperparameters chosen are :

- Generator Learning rate: 0.0001
- Discriminator Learning rate: 0.0002 . Higher learning rates were putting the GAN in collapse mode. Or showing instability in the error over epochs.
- Num of epochs: 1000

- Batch Size: 32
- Loss function for discriminator: Binary Cross Entropy, suitable for discriminator training.
- Optimizer Types: Adam.

The change in learning rate provides better stability of the losses over the epochs. With the same learning rates of 0.001, the losses tended to diverge.

2.4 Progression of Training loss

As expected, the training loss for both neural networks converges to around 0.5, which is an ideal scenario in GAN training.

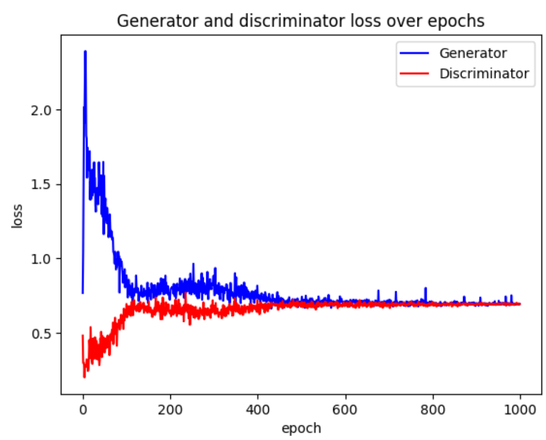


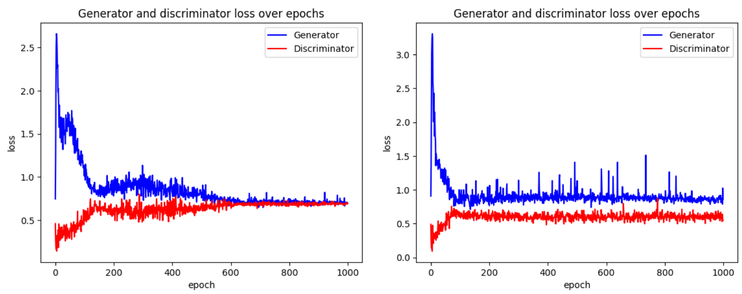
Figure 2: Losses through epochs

2.5 Noise Dimension Experiment

I attempted to modify the noise dimension from two to one, which impacted the stability of the solution. The results with a latent space dimension of 1 are similar to what we have with a latent space dimension of 2. With a dimension 3, the results are not optimal. (figure 3.a 3.b)

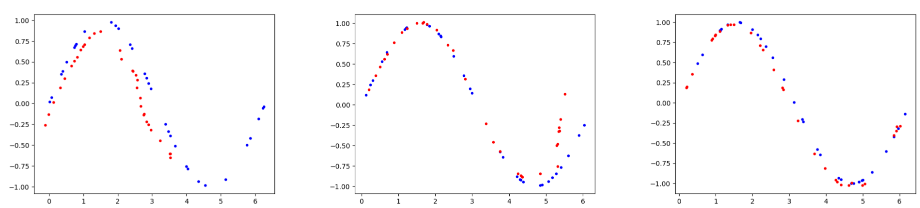
2.6 Decision to Stop Training

Graphically, we might consider stopping the training when both the generator and discriminator losses stabilize around convenient values, approximately 0.5 for each. However, it's evident that even when the losses appear stable, the generated data don't exhibit a sinusoidal pattern. Hence, we refrain from prematurely halting the training.



(a) Latent space dimension 1 (b) Latent space dimension 3

Figure 3: Comparison of responses in different latent space dimensions



(a) Generated data for epoch = 100 (b) Generated data for epoch = 500 (c) Generated data for epoch = 800

Figure 4: Plots of Generated Data

3 Two-dimensional circular data

In this section, we aim to generate two-dimensional data resembling a circle distribution. The input to our GAN is illustrated in the following distribution in figure 1. Starting from noise, we train our GAN to replicate this distribution.

3.1 Neural Network Architecture

Generator: The generator is a neural network composed of four linear layers with ReLU activation functions between them. It starts with an input dimension of two and sequentially expands to 16, then 32, followed by 64, and finally contracts back to two dimensions.

Discriminator: The discriminator is designed as a neural network consisting of four linear layers with ReLU activations. Similar to the generator, it begins with an input dimension of two and gradually narrows down the representation through intermediate layers of 256, 128, and 64 neurons. Dropout layers, also with a dropout rate of 0.3, are used after each linear layer to mitigate overfitting. The final linear layer outputs a single value via a sigmoid activation

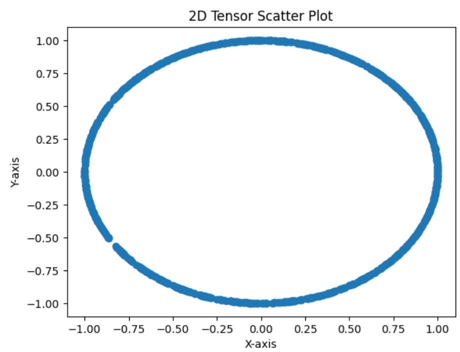


Figure 5: Input distribution

function, indicating the probability that the input data is real (from the true data distribution) rather than generated.

3.2 Data Preparation

The data preparation for this dataset involves generating a set of 1024 data points distributed uniformly on a unit circle. This is achieved by first generating random angles, ranging from 0 to 2π, using the formula $2 \times \pi \times torch.rand(1024)$. These angles represent the polar coordinates.

Next, these polar coordinates are converted to Cartesian coordinates using the equations:

$$x = radius \times \cos(\theta)$$

$$y = radius \times \sin(\theta)$$

where $radius$ is set to 1. The resulting x and y values represent the Cartesian coordinates of points uniformly distributed on the unit circle. Finally, these coordinates are stored in the train data tensor, where each row contains the x and y coordinates of a data point.

3.3 Hyperparameters

We adjusted the learning rates to enhance convergence speed. Specifically, we set the discriminator's learning rate to 0.002 and the generator's to 0.001.

3.4 Progression of Training Loss

The plotted progression of training loss across epochs indicates rapid stabilization, converging to an optimal value near 0.5.

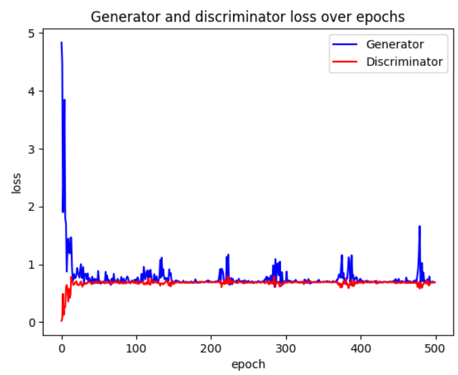
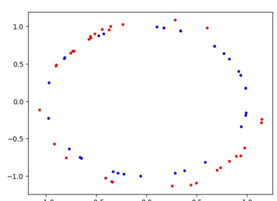


Figure 6: Losses through epochs

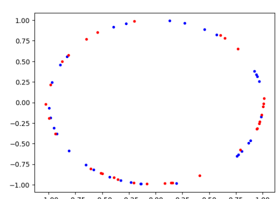
3.5 Decision to Stop Training

Here, it could be due to the GAN architecture being more optimized for generating a circular distribution, resulting in rapid convergence of losses. The generated data closely resemble the training data quite early in the training process. The graphs below illustrate that, for mid-epochs, there's little discrepancy between the generated distribution and the original one, unlike in the initial generation task.

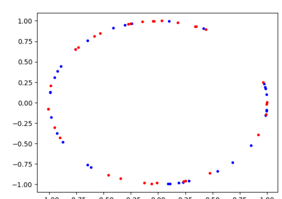
Here we can stop the training early. A condition to stop the training early would be a convergence in mean and standard deviation of the losses.



(a) Generated data for epoch = 100



(b) Generated data for epoch = 500

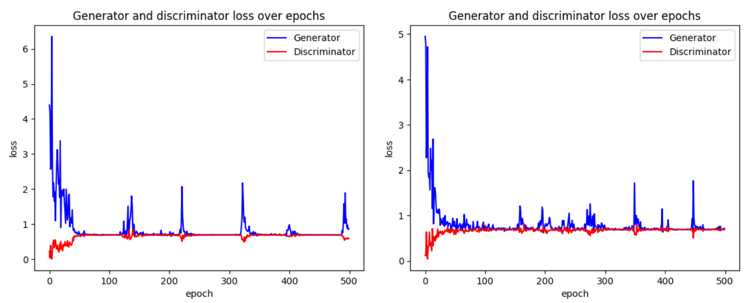


(c) Generated data for epoch = 800

Figure 7: Plots of Generated Data

3.6 Noise Dimension Experiment

I attempted to modify the noise dimension from two to one, which impacted the stability of the solution. Once again, the generator loss exhibits oscillations, more pronounced than before. As for the dimension 3, the response is overall the same as dimension 2.



(a) Latent space dimension 1 (b) Latent space dimension 3

Figure 8: Comparison of responses in different latent space dimensions

4 Two-dimensional polar coordinate

In this section, we aim to generate two-dimensional data resembling a certain pattern distribution. The input distribution to our GAN is illustrated in the following distribution in figure 1. Starting from noise, we train our GAN to replicate this distribution.

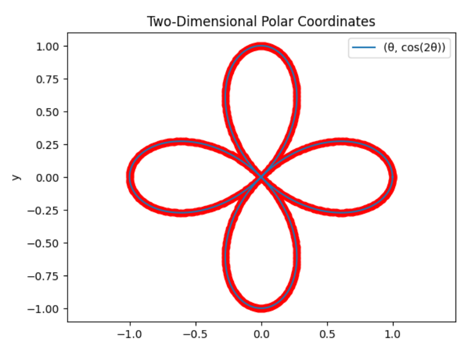


Figure 9: Input distribution

4.1 Neural Network Architecture

Generator Architecture The generator comprises five fully connected layers. It starts with an input layer of two neurons and utilizes ReLU activation functions throughout. The succeeding layers consist of 16, 32, 64, and 32 neurons, respectively, before outputting a two-dimensional result.

Discriminator Architecture The discriminator is structured with five fully connected layers. These layers include 256 neurons in the input layer, followed by layers with 128, 64, and 1 neuron(s), respectively. Each layer is activated by ReLU functions, with dropout layers integrated between them to address overfitting. The final layer implements a sigmoid activation function, providing probability-based outputs.

Complexity was added to the NNs to grasp the growing complexity of the data wee seek to generate.

4.2 Data Preparation

The initial dataset is generated to contain 1024 data points in a two-dimensional space. These data points are organized to form a circular distribution. This is achieved by creating a set of polar coordinates where the angle (θ) varies from 0 to 2π, generating 1000 points evenly distributed between these values. The radial coordinate (r) is determined using $r = \cos(2\theta)$, which results in the circular distribution. These polar coordinates are then converted into Cartesian coordinates (x and y) using the trigonometric relations: $x = r \times \cos(\theta)$ and $y = r \times \sin(\theta)$.

The resulting Cartesian coordinates (x and y) represent the data points, forming a nice drawing distribution when plotted in a two-dimensional space. After generating this dataset, it is formatted into a DataLoader with a batch size of 32 to facilitate efficient processing during the training of the neural network model.

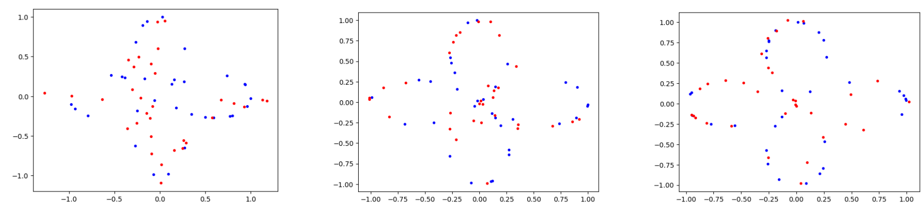
4.3 Hyperparameters

For enhanced visualization and better pattern resolution, the batch size was increased to 64 from the previous value of 32. This modification allows for a clearer visualization of the more intricate pattern. Additionally, the adjustment of the learning rates was made to further stabilize the loss throughout the epochs. The learning rate for the discriminator (lr_d) was set to 0.0002, while the learning rate for the generator (lr_g) was adjusted to 0.0001. These changes were implemented specifically to improve the stability of the loss during training.

Aside from these modifications, the remaining hyperparameters are consistent with the previously used values.

4.4 Progression of Training loss and Decision to Stop Training

The losses across epochs and various plots of generated data are provided below. It's noticeable that the losses quickly converge. However, similar to the initial part of the homework, the generated images are not sufficiently accurate. Despite stable losses, the generated images remain inaccurate even after 500 epochs.



(a) Generated data for epoch = 100

(b) Generated data for epoch = 500

(c) Generated data for epoch = 800

Figure 10: Plots of Generated Data

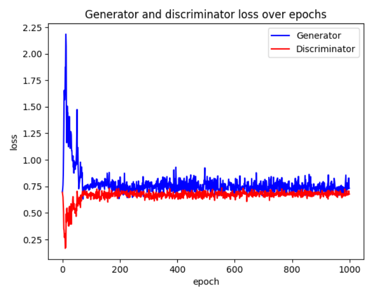
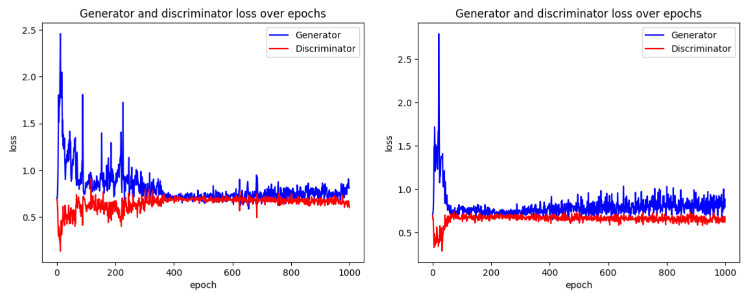


Figure 11: Losses through epochs

4.5 Noise Dimension Experiment

Once again, the results are quite similar, but the latent space of dimension 3 produces less stable outcomes.



(a) Latent space dimension 1 (b) Latent space dimension 3

Figure 12: Comparison of responses in different latent space dimensions

5 Synthetic time series

In this section, our aim was to model time series using GANs. We utilized Amazon data spanning from 2015 to 2019 as our training set, segmenting the closing prices into 60-day periods to feed into the algorithm. This segment was notably more intricate than the others due to the complexity of the data. Formatting the loader and adapting the algorithm to this unique data format required considerable effort. Moreover, the intricacy of the data demanded a more sophisticated architecture, leading to extensive training times. Perhaps I didn't optimize my algorithm correctly, as each training loop took around 6 to 10 minutes on my older computer. Consequently, I couldn't fully explore the best hyperparameters. Despite this constraint, I made efforts to enhance my generation. Notably, the difference between the outputs from the initial and final training loops proved satisfactory.

5.1 Neural Network Architecture

Generator Architecture:

- Input Layer: Receives an input of size N representing the slicing window of the time series data.
- Hidden Layers:
  - Consists of linear layers followed by Rectified Linear Unit (ReLU) activations, and dropouts.
  - Successive linear layers gradually expand and contract the network width: `Linear(2N)`, `Linear(4N)`, `Linear(4N)`, `Linear(8N)`, `Linear(4N)`, `Linear(4N)`, `Linear(2N)`.
- Output Layer: Produces an output of size N representing the generated time series data.

Discriminator Architecture:

- Input Layer: Accepts input of size N, similar to the Generator's input layer.
- Hidden Layers:
  - Consists of linear layers followed by Rectified Linear Unit (ReLU) activations, and dropouts.
  - Successive linear layers gradually expand and contract the network width: `Linear(2N)`, `Linear(4N)`, `Linear(4N)`, `Linear(2N)`, `Linear(N)`,
- Output Layer: Produces a single output representing the probability that the input data is real (i.e., from the original time series data).

5.2 Data Preparation

Downloading Data: Using yfinance, it fetches historical stock data for Amazon ("AMZN") from January 1, 2015, to January 1, 2020.

Extracting Closing Volumes: Selects the 'Close' prices from the downloaded stock data and stores them in `closing_volumes`.

Choosing a Slicing Window: Sets $N = 60$, representing the size of the slicing window for the time series data.

Creating Input Sequences: Iterates through the closing volume data and creates input sequences of length N by sliding a window across the closing volume time series. Scales each input sequence data using `MinMaxScaler` to normalize it within the range of -1 to 1. This range is chosen because the ReLU activation function is most effective in this range. Reshapes and converts the scaled data into a format suitable for further processing in the algorithm.

Converting to PyTorch Tensors: Converts the scaled input sequences into PyTorch tensors (`train_data`) of type `torch.float32`.

Creating DataLoader: Creates a PyTorch DataLoader (`train_loader`) to handle batching and shuffling of the `train_data` tensor. The DataLoader is configured to have a batch size of 32, shuffling the data at each epoch, and dropping the last incomplete batch.

5.3 Hyperparameters

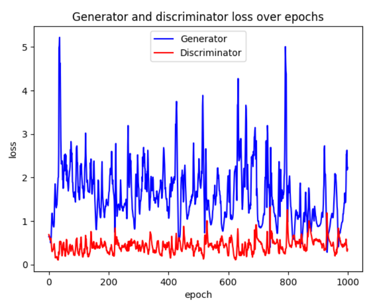
- Generator Learning rate: 0.0001
- Discriminator Learning rate: 0.0002 . Higher learning rates were putting the GAN in collapse mode, or producing unstable results in the losses over epochs. Lower learning rates were producing same results.
- Num of epochs: 1000, I tried with 500 to start but the algorithm seemed like it needed the 1000 to converge.
- Batch Size: 128. Tried different values, honestly couldn't spot the difference with 32, 64 or 128, so this one was chosen to have a faster training loop.

- Loss function for discriminator: Binary Cross Entropy, suitable for discriminator training.
- Optimizer Types: Adam.

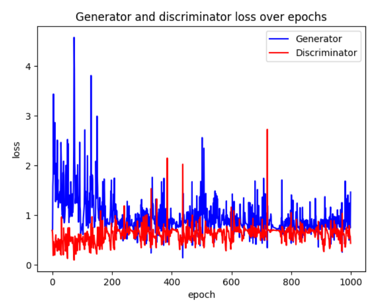
5.4 Progression of Training

5.4.1 Losses

The plot depicting losses over epochs reveals instability, particularly evident in the generator's loss. It's unclear whether this instability arises from the hyperparameters or insufficient complexity in the architecture to capture the dataset's intricacies. Unfortunately, computational limitations hinder my ability to experiment with new code. However, by expanding both the width and depth of the generator and discriminator architectures, I managed to enhance the GAN's performance, as illustrated in Figure a and Figure b.



(a) Results before improving the architecture

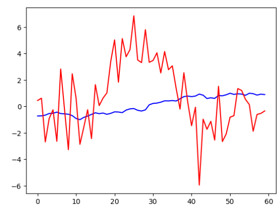


(b) Results after improving the architecture

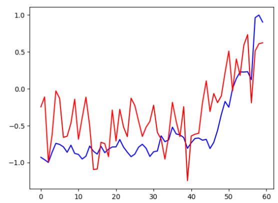
Figure 13: Overall Caption for Both Images

5.4.2 Accuracy of generated data

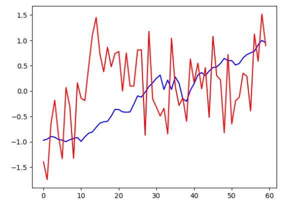
Throughout the entire training process, generated data were consistently plotted, revealing a significant observation: the model's performance is unsatisfactory. While I didn't anticipate the generated data to perfectly match the vast and diverse distribution of the dataset, the outcome is notably challenging. The generated data exhibit excessive noise and erratic patterns. At times, the model accurately captures the overall trend, particularly during periods of increasing time series. However, it struggles when the data remains stable or shows a decreasing trend over the slicing period. This could be attributed to Amazon's stock, which has generally shown significant growth over recent years, influencing the generation to mimic this trend. Nevertheless, the generated data lack smoothness, indicating a potential issue of overfitting and training on noise. Perhaps penalizing the generator for modeling noisy data could be a viable approach to improve its performance.



(a) Generated data for flat tendency



(b) Generated data increasing trend

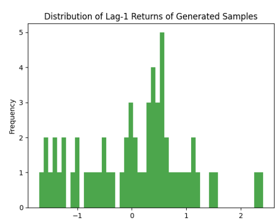


(c) Generated data increasing trend but too noisy

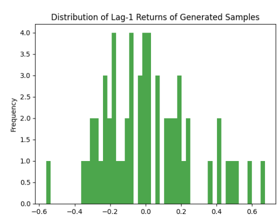
Figure 14: Plots of Generated Data

5.4.3 A stylized fact: returns distribution.

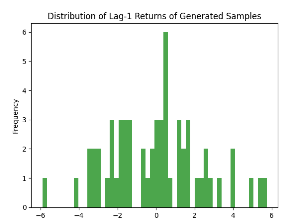
During training, we plotted the lag 1 return distribution. Ideally, the returns should conform to a normal distribution with a mean of 0. Gradually, we observe this trend taking shape. Although the generated data may not visually resemble the initial distribution entirely, the return distribution seems to exhibit the correct statistical properties.



(a) Returns distribution epoch = 100



(b) Returns distribution epoch = 500



(c) Returns distribution epoch = 900

Figure 15: Plots of Generated Data

6 Code

Once again, I've uploaded all the code to my GitHub in an .ipynb format for easy compilation. However, to follow the instructions, I'll also paste the code here. Please find the code for GANs for time series generation in the following GitHub Repository.

6.1 Part 1

```

# Import the necessary libs for the homework
import torch
from torch import nn
import matplotlib.pyplot as plt
import numpy as np
import math
import pandas as pd

# Generate the initial dataset :
train_len = 1024
train_data = torch.zeros(train_len, 2)
train_data[:,0] = 2 * math.pi * torch.rand(train_len)
train_data[:,1] = torch.sin(train_data[:,0])

# Plots the content of the tensor to make sure it's what i
    want
plt.scatter(train_data[:,0], train_data[:,1])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('2D Tensor Scatter Plot')
plt.show()

# Format the initial data into a loader to make a more
    efficient code :
batch_size = 32                              #
    --------------------------------------------------------
     Batch size

# Wrap the tensor in a TensorDataset to assure compatibility
     in the code :
train_dataset = torch.utils.data.TensorDataset(train_data)

# Create a DataLoader using the dataset :
train_loader = torch.utils.data.DataLoader(train_dataset,
    batch_size=batch_size, shuffle=True)

# Let's start with the generator as a simple neural network
class Generator(nn.Module):

```

```
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(2, 16), # Input is two dimensional
            nn.ReLU(),
            nn.Linear(16, 32),
            nn.ReLU(),
            nn.Linear(32, 2),  # Output is two dimensional
        )

    def forward(self, x):
        output = self.model(x)
        return output

# Build the discriminator as a NN
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(2, 256), #the input is two-dimensional
            nn.ReLU(),
            nn.Dropout(0.3), #droput layers reduce overfitting
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 1),
            nn.Sigmoid(),#sigmoid activation to represent probability
        )

    def forward(self, x):
        output = self.model(x)
        return output

    # Training loop :

discriminator = Discriminator()
generator = Generator()
gen_loss_vector = []
discr_loss_vector = []
num_epoch_vector = []


optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=2*lr)
```

```python
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr)

for epoch in range(num_epochs):
    for idx, real_data_set in enumerate(train_loader):
        real_data_set = real_data_set[0]
        # Preparing the real data to train the discriminator:
        real_data_label = torch.ones(batch_size,1)

        # Preparing the fake data to train the discriminator:
        noise_data_set = torch.randn((batch_size, 2))
        fake_data_set = generator(noise_data_set)
        fake_data_label = torch.zeros(batch_size, 1)

        # Creating the training samples set:
        training_data_set = torch.cat((real_data_set, fake_data_set))

        # Creating the training labels set:
        training_labels_set = torch.cat((real_data_label, fake_data_label))

        # Train the discriminator:
        discriminator.zero_grad()
        output_discriminator = discriminator(training_data_set)
        loss_discriminator = loss_function(
            output_discriminator, training_labels_set)
        loss_discriminator.backward()
        optimizer_discriminator.step()

        # Initialising the data for the discriminator:
        noise_data_set = torch.randn((batch_size, 2))

        # Train the generatot:
        generator.zero_grad()
        output_generator = generator(noise_data_set)

        # We use the discriminator output to back propagate:
        output_discriminator_generated  = discriminator(output_generator)
        loss_generator = loss_function(
            output_discriminator_generated, real_data_label)
        # We put label = 1 so that the error we want to minimize is the distance between our generated data and the label 1

        loss_generator.backward()
```

```
        optimizer_generator.step()

        # prepares data for loss plot afterwise:
        if idx == batch_size-1:
            gen_loss_vector.append(float(loss_generator))
            discr_loss_vector.append(float(loss_discriminator))
            num_epoch_vector.append(epoch)

        # Show loss
        if epoch % 100 == 0 and idx == batch_size - 1:
            print(f"Epoch: {epoch} Loss D.: {loss_discriminator}")
            print(f"Epoch: {epoch} Loss G.: {loss_generator}")


            generated_samples_for_plotting = output_generator.detach()

            # Plot real samples in blue
            plt.plot(real_data_set[:, 0], real_data_set[:, 1], ".", color='blue')

            # Plot generated samples in red
            plt.plot(generated_samples_for_plotting[:, 0], generated_samples_for_plotting[:, 1], ".", color='red')

            plt.show()

    # Plot the functions on the same graph
    plt.plot(num_epoch_vector, gen_loss_vector, label='Generator', color = 'b')  # Plot sine function with label
    plt.plot(num_epoch_vector, discr_loss_vector, label='Discriminator', color = 'r')  # Plot cosine function with label
    plt.legend()  # Show legend with function labels
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.title('Generator and discriminator loss over epochs')
    plt.show()
```

Listing 1: pb 1

6.2 Part 2

```
# Import the necessary libs for the homework
import torch
from torch import nn
import matplotlib.pyplot as plt
import numpy as np
import math
import pandas as pd


# Generate the initial dataset :
train_len = 1024
train_data = torch.zeros(train_len, 2)

# Generate random angles
theta = 2 * math.pi * torch.rand(train_len)

# Convert polar coordinates to Cartesian coordinates
radius = 1
x = radius * torch.cos(theta)
y = radius * torch.sin(theta)

train_data[:,0] = x
train_data[:,1] = y

# Plots the content of the tensor to make sure it's what i
    want
plt.scatter(train_data[:,0], train_data[:,1])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('2D Tensor Scatter Plot')
plt.show()


# Format the initial data into a loader to make a more
    efficient code :
batch_size = 32                          #
    -------------------------------------------------------
     Batch size

# Wrap the tensor in a TensorDataset to assure compatibility
     in the code :
train_dataset = torch.utils.data.TensorDataset(train_data)

# Create a DataLoader using the dataset :
train_loader = torch.utils.data.DataLoader(train_dataset,
    batch_size=batch_size, shuffle=True)

# Let's start with the generator as a simple neural network
class Generator(nn.Module):
```

```python

     def __init__(self):
         super().__init__()
         self.model = nn.Sequential(
             nn.Linear(3, 16), # Input is 3 dimensional
             nn.ReLU(),
             nn.Linear(16, 32),
             nn.ReLU(),
             nn.Linear(32, 64),
             nn.ReLU(),
             nn.Linear(64, 32),
             nn.ReLU(),
             nn.Linear(32, 2),  # Output is two dimensional
         )

     def forward(self, x):
         output = self.model(x)
         return output

# Build the discriminator as a NN
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(2, 256), #the input is two-dimensional
            nn.ReLU(),
            nn.Dropout(0.3), #droput layers reduce overfitting
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 1),
            nn.Sigmoid(), #sigmoid activation to represent probability
        )

    def forward(self, x):
        output = self.model(x)
        return output

# Sets the parameters
lr = 0.001
loss_function = nn.BCELoss()
num_epochs = 500

# Training loop :

```

```python
discriminator = Discriminator()
generator = Generator()
gen_loss_vector = []
discr_loss_vector = []
num_epoch_vector = []


optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=2*lr)
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr)

for epoch in range(num_epochs):
    for idx, real_data_set in enumerate(train_loader):
        real_data_set = real_data_set[0]
        # Preparing the real data to train the discriminator:
        real_data_label = torch.ones(batch_size,1)

        # Preparing the fake data to train the discriminator:
        noise_data_set = torch.randn((batch_size, 3))
        fake_data_set = generator(noise_data_set)
        fake_data_label = torch.zeros(batch_size, 1)

        # Creating the training samples set:
        training_data_set = torch.cat((real_data_set, fake_data_set))

        # Creating the training labels set:
        training_labels_set = torch.cat((real_data_label, fake_data_label))

        # Train the discriminator:
        discriminator.zero_grad()
        output_discriminator = discriminator(training_data_set)
        loss_discriminator = loss_function(
            output_discriminator, training_labels_set)
        loss_discriminator.backward()
        optimizer_discriminator.step()

        # Initialising the data for the generator:
        noise_data_set = torch.randn((batch_size, 3))

        # Train the generatot:
        generator.zero_grad()
        output_generator = generator(noise_data_set)

        # We use the discriminator output to back propagate:
```

```python
        output_discriminator_generated  = discriminator(output_generator)
        loss_generator = loss_function(
            output_discriminator_generated, real_data_label)
        # We put label = 1 so that the error we want to minimize is the distance between our generated data and the label 1

        loss_generator.backward()
        optimizer_generator.step()

        # prepares data for loss plot afterwise:
        if idx == batch_size-1:
            gen_loss_vector.append(float(loss_generator))
            discr_loss_vector.append(float(loss_discriminator))
            num_epoch_vector.append(epoch)

        # Show loss
        if epoch % 100 == 0 and idx == batch_size - 1:
            print(f"Epoch: {epoch} Loss D.: {loss_discriminator}")
            print(f"Epoch: {epoch} Loss G.: {loss_generator}")


            generated_samples_for_plotting = output_generator.detach()

            # Plot real samples in blue
            plt.plot(real_data_set[:, 0], real_data_set[:, 1], ".", color='blue')

            # Plot generated samples in red
            plt.plot(generated_samples_for_plotting[:, 0], generated_samples_for_plotting[:, 1], ".", color='red')

            plt.show()

# Plot the functions on the same graph
plt.plot(num_epoch_vector, gen_loss_vector, label='Generator', color = 'b')  # Plot sine function with label
plt.plot(num_epoch_vector, discr_loss_vector, label='Discriminator', color = 'r')  # Plot cosine function with label
plt.legend()  # Show legend with function labels
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('Generator and discriminator loss over epochs')
```

```
plt.show()
```

Listing 2: pb 1

6.3 Part 3

```
# Import the necessary libs for the homework
import torch
from torch import nn
import matplotlib.pyplot as plt
import numpy as np
import math
import pandas as pd

# Generate the initial dataset :
train_len = 1024
train_data = torch.zeros(train_len, 2)

# Generate polar coordinates
theta = torch.linspace(0, 2 * math.pi, train_len)  # 1000 points between 0 and 2
r = torch.cos(2 * theta)

# Convert to Cartesian coordinates
x = r * torch.cos(theta)
y = r * torch.sin(theta)

train_data[:,0] = r * torch.cos(theta)
train_data[:,1] = r * torch.sin(theta)

# Plot the points
plt.plot(x, y, label='(  , cos(2  ))')
plt.scatter(x, y, color='red')  # Scatter plot to highlight individual points
plt.title('Two-Dimensional Polar Coordinates')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.axis('equal')
plt.show()

# Format the initial data into a loader to make a more efficient code :
batch_size = 32                              # --------------------------------------------------------- Batch size

# Wrap the tensor in a TensorDataset to assure compatibility in the code :
```

```python
train_dataset = torch.utils.data.TensorDataset(train_data)

# Create a DataLoader using the dataset :
train_loader = torch.utils.data.DataLoader(train_dataset,
    batch_size=batch_size, shuffle=True)

# Let's start with the generator as a simple neural network
class Generator(nn.Module):

     def __init__(self):
         super().__init__()
         self.model = nn.Sequential(
             nn.Linear(2, 16), # Input is two dimensional
             nn.ReLU(),
             nn.Linear(16, 32),
             nn.ReLU(),
             nn.Linear(32, 64),
             nn.ReLU(),
             nn.Linear(64, 32),
             nn.ReLU(),
             nn.Linear(32, 2),  # Output is two dimensional
         )

     def forward(self, x):
         output = self.model(x)
         return output

# Build the discriminator as a NN
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(2, 256), #the input is two-dimensional
            nn.ReLU(),
            nn.Dropout(0.3), #droput layers reduce
                overfitting
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 1),
            nn.Sigmoid(),#sigmoid activation to represent
                probability
        )

    def forward(self, x):
        output = self.model(x)
        return output
```

```

# Training loop :

discriminator = Discriminator()
generator = Generator()
gen_loss_vector = []
discr_loss_vector = []
num_epoch_vector = []


optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr_d)
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr_g)

for epoch in range(num_epochs):
    for idx, real_data_set in enumerate(train_loader):
        real_data_set = real_data_set[0]
        # Preparing the real data to train the discriminator :
        real_data_label = torch.ones(batch_size,1)

        # Preparing the fake data to train the discriminator :
        noise_data_set = torch.randn((batch_size, 2))
        fake_data_set = generator(noise_data_set)
        fake_data_label = torch.zeros(batch_size, 1)

        # Creating the training samples set:
        training_data_set = torch.cat((real_data_set, fake_data_set))

        # Creating the training labels set:
        training_labels_set = torch.cat((real_data_label, fake_data_label))

        # Train the discriminator:
        discriminator.zero_grad()
        output_discriminator = discriminator(training_data_set)
        loss_discriminator = loss_function(
            output_discriminator, training_labels_set)
        loss_discriminator.backward()
        optimizer_discriminator.step()

        # Initialising the data for the discriminator:
        noise_data_set = torch.randn((batch_size, 2))

        # Train the generatot:
        generator.zero_grad()
```

```
        output_generator = generator(noise_data_set)

        # We use the discriminator output to back propagate:
        output_discriminator_generated  = discriminator(output_generator)
        loss_generator = loss_function(
            output_discriminator_generated, real_data_label)
        # We put label = 1 so that the error we want to minimize is the distance between our generated data and the label 1

        loss_generator.backward()
        optimizer_generator.step()

        # prepares data for loss plot afterwise:
        if idx == 1:
            gen_loss_vector.append(float(loss_generator))
            discr_loss_vector.append(float(loss_discriminator))
            num_epoch_vector.append(epoch)

        # Show loss
        if epoch % 100 == 0 and idx == 1:
            print(f"Epoch: {epoch} Loss D.: {loss_discriminator}")
            print(f"Epoch: {epoch} Loss G.: {loss_generator}")


            generated_samples_for_plotting = output_generator.detach()

            # Plot real samples in blue
            plt.plot(real_data_set[:, 0], real_data_set[:, 1], ".", color='blue')

            # Plot generated samples in red
            plt.plot(generated_samples_for_plotting[:, 0], generated_samples_for_plotting[:, 1], ".", color='red')

            plt.show()
```

Listing 3: pb 1

6.4 Part 4

```
# Import the necessary libs for the homework
import torch
```

```python
from torch import nn
import matplotlib.pyplot as plt
import numpy as np
import math
import pandas as pd
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler



# Import Amazon data from yfinance
stock_data = yf.download("AMZN", start="2015-01-01", end="2020-01-01")

# Extract closing volumes
closing_volumes = stock_data['Close']

# Choose the slicing window
N = 60

# Create a tensor containing the slicing windows of closing volumes
input_sequences = []
for i in range(len(closing_volumes) - N):
    input_sequence = closing_volumes.iloc[i:i+N].values

    # Scale the data for better convergence of the algrithm.
    input_array = np.array(input_sequence).reshape(-1, 1)
    scaler = MinMaxScaler(feature_range=(-1, 1)) # Scale between -1 and 1 because its the area where ReLu is the most efficient
    scaled_data = scaler.fit_transform(input_array)
    scaled_data_list = []

    # Using a list format for algorithm compatibility after
    for data in scaled_data:
        scaled_data_list.append(data[0])

    input_sequences.append(scaled_data_list)

train_data = torch.tensor(input_sequences, dtype=torch.float32)

# Convert your data to a TensorDataset using torch.TensorDataset for loader
train_dataset = torch.utils.data.TensorDataset(train_data)

batch_size = 32

```

```python
# Create a DataLoader with shuffle=True for shuffling at each epoch
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True, drop_last=True)

# Designing the architecture of the GAN
class Generator(nn.Module):

     def __init__(self):
         super().__init__()
         self.model = nn.Sequential(
             nn.Linear(N, 2*N), # Input is a N slicing window
             nn.ReLU(),
             nn.Linear(2*N, 4*N),
             nn.ReLU(),
             nn.Dropout(0.3), #droput layers reduce overfitting
             nn.Linear(4*N, 4*N),
             nn.ReLU(),
             nn.Dropout(0.3), #droput layers reduce overfitting
             nn.Linear(4*N, 8*N),
             nn.ReLU(),
             nn.Dropout(0.3), #droput layers reduce overfitting
             nn.Linear(8*N, 4*N),
             nn.ReLU(),
             nn.Dropout(0.3), #droput layers reduce overfitting
             nn.Linear(4*N, 4*N),
             nn.ReLU(),
             nn.Linear(4*N, 2*N),
             nn.ReLU(),
             nn.Linear(2*N, N),  # Output is N slicing window
         )

     def forward(self, x):
         output = self.model(x)
         return output

# Build the discriminator as a NN
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(N, 2*N), #the input is lenght N
            nn.ReLU(),
```

```
            nn.Dropout(0.3), #droput layers reduce overfitting
            nn.Linear(2*N, 4*N),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(4*N, 4*N),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(4*N, 2*N),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(2*N, N),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(N, 1),
            nn.Sigmoid(), # sigmoid activation to represent probability
        )

    def forward(self, x):
        output = self.model(x)
        return output

from tqdm import tqdm
# Training loop :
num_epochs = 1000
discriminator = Discriminator()
generator = Generator()
gen_loss_vector = []
discr_loss_vector = []
num_epoch_vector = []


optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr_d)
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr_g)


for epoch in tqdm(range(num_epochs)):

    for index, batch in enumerate(train_loader):
        real_data_set = batch[0]

        # Preparing the real data to train the discriminator :
        real_data_label = torch.ones(batch_size,1)

        # Preparing the fake data to train the discriminator :
```

```python
        noise_data_set = torch.randn((batch_size, N))
        fake_data_set = generator(noise_data_set)
        fake_data_label = torch.zeros(batch_size, 1)

        # Creating the training samples set:
        training_data_set = torch.cat((real_data_set, fake_data_set))

        # Creating the training labels set:
        training_labels_set = torch.cat((real_data_label, fake_data_label))

        # Train the discriminator:
        discriminator.zero_grad()
        output_discriminator = discriminator(training_data_set)
        loss_discriminator = loss_function(
            output_discriminator, training_labels_set)
        loss_discriminator.backward()
        optimizer_discriminator.step()

        # Initialising the data for the gznzrator:
        noise_data_set = torch.randn((batch_size, N))

        # Train the generatot:
        generator.zero_grad()
        output_generator = generator(noise_data_set)
        output_discriminator_generated = discriminator(output_generator)
        loss_generator = loss_function(output_discriminator_generated, real_data_label)
        # We put label = 1 so that the error we want to minimize is the distance between our generated data and the label 1
        loss_generator.backward()
        optimizer_generator.step()

        # prepares data for loss plot afterwise:
        if index == 0:
            gen_loss_vector.append(float(loss_generator))
            discr_loss_vector.append(float(loss_discriminator))
            num_epoch_vector.append(epoch)


        # Show loss
        if epoch % 50 == 0 and index == 0:
            print(f"Epoch: {epoch} Loss D.: {loss_discriminator}")
```

```python
            print(f"Epoch: {epoch} Loss G.: {loss_generator}")

            # Let's plot the first time series of the batch in blue
            time_steps = []
            for time in range(len(real_data_set[0])):
                time_steps.append(time)
            plt.plot(time_steps,real_data_set[0],"-",color='blue')

            generated_samples_for_plotting = output_generator.detach()[0]

            # Plot generated samples in red
            plt.plot(time_steps, generated_samples_for_plotting, "-", color='red')
            plt.show()

            # let's plot the lag 1 return distribution
            # Calculate lag-1 returns
            lag_1_returns = generated_samples_for_plotting[1:] - generated_samples_for_plotting[:-1]

            # Plot the distribution of lag-1 returns
            plt.hist(lag_1_returns, bins=50, color='green', alpha=0.7)
            plt.xlabel('Lag-1 Returns')
            plt.ylabel('Frequency')
            plt.title('Distribution of Lag-1 Returns of Generated Samples')
            plt.show()
```
Listing 4: pb 1