

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.

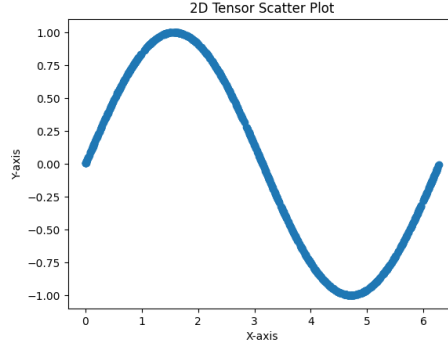


Figure 1: Input distribution

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.

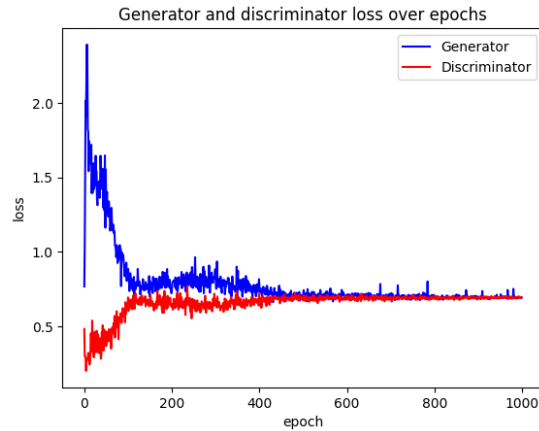


Figure 2: Losses through epochs

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.

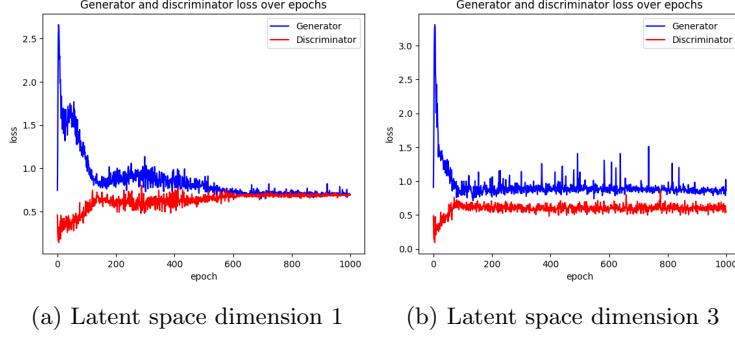


Figure 3: Comparison of responses in different latent space dimensions

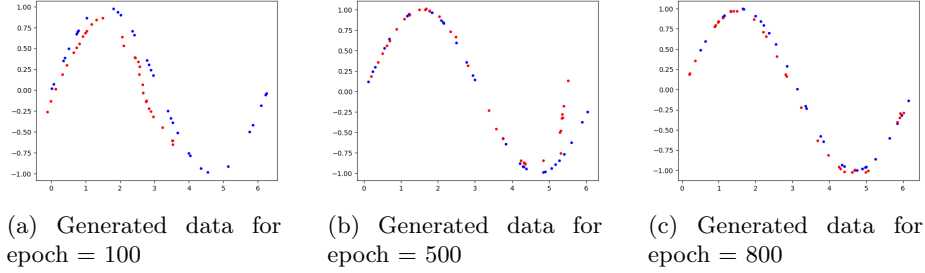


Figure 4: Plots of Generated Data

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

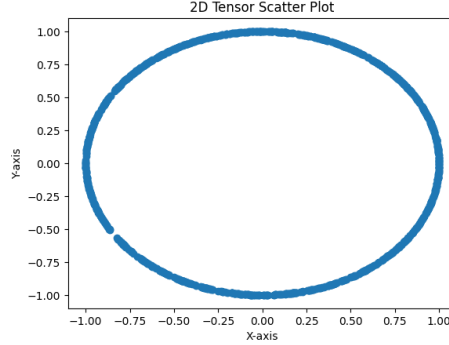


Figure 5: Input distribution

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 $2 \times \pi \times \text{torch.rand}(1024)$. These angles represent the polar coordinates.

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

$$x = \text{radius} \times \cos(\theta)$$

$$y = \text{radius} \times \sin(\theta)$$

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.

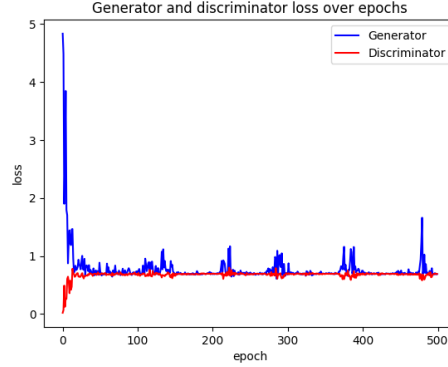


Figure 6: Losses through epochs

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.

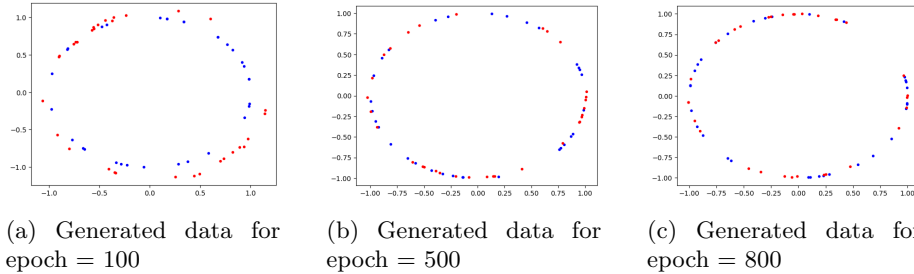


Figure 7: Plots of Generated Data

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.

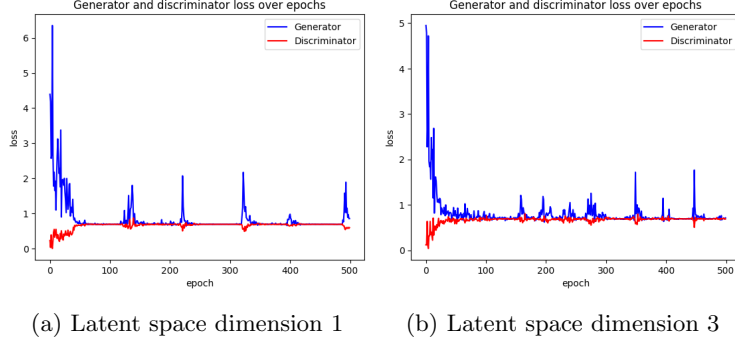


Figure 8: Comparison of responses in different latent space dimensions

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.

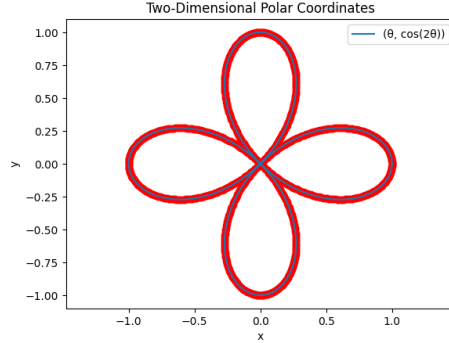


Figure 9: Input distribution

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 we 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.

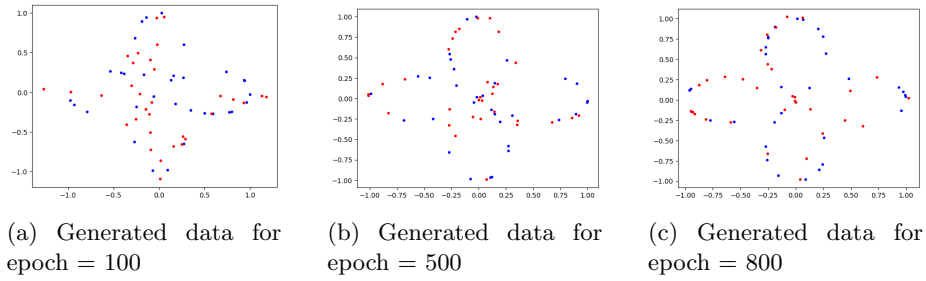


Figure 10: Plots of Generated Data

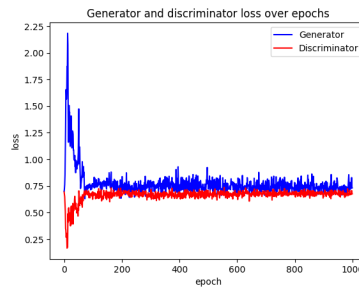


Figure 11: Losses through epochs

4.5 Noise Dimension Experiment

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

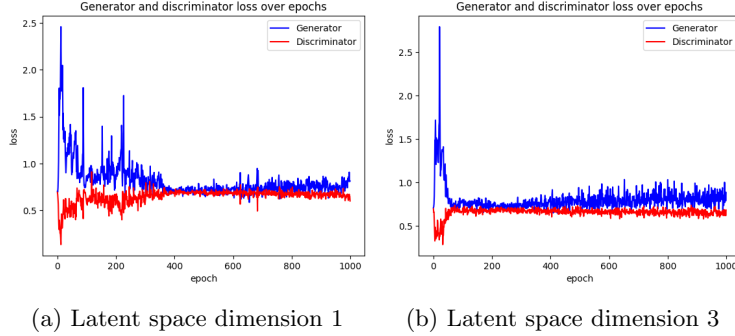


Figure 12: Comparison of responses in different latent space dimensions

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

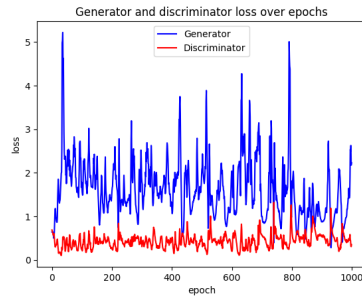
- Generator Learning rate: 0.0001
- Discriminator Learning rate: 0.0002 . Higher learning rates were putting the GAN in collapse mode, or producing unstable results in the losses over epochs. Lower learning rates were producing same results.
- Num of epochs: 1000, I tried with 500 to start but the algorithm 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.

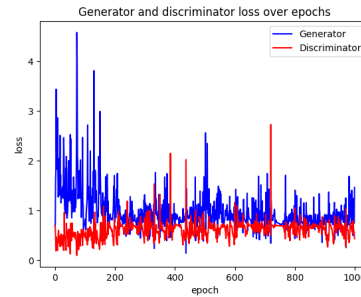
5.4 Progression of Training

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.

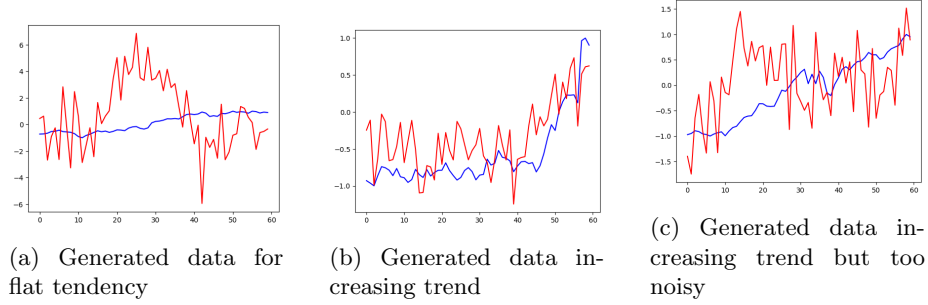


Figure 14: Plots of Generated Data

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.

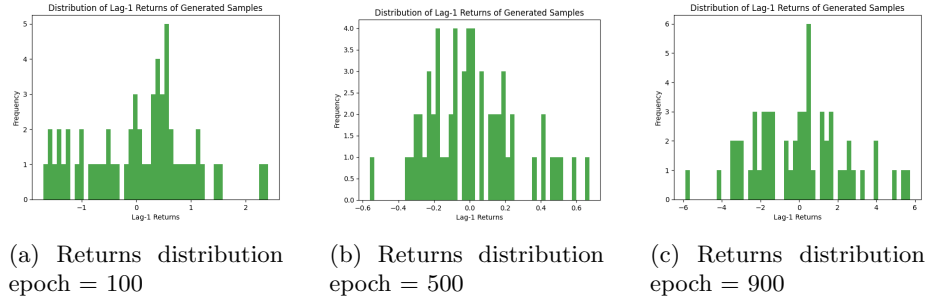


Figure 15: Plots of Generated Data

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

```
1
2 # Import the necessary libs for the homework
3 import torch
4 from torch import nn
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import math
8 import pandas as pd
9
10 # Generate the initial dataset :
11 train_len = 1024
12 train_data = torch.zeros(train_len, 2)
13 train_data[:,0] = 2 * math.pi * torch.rand(train_len)
14 train_data[:,1] = torch.sin(train_data[:,0])
15
16 # Plots the content of the tensor to make sure it's what i
    want
17 plt.scatter(train_data[:,0], train_data[:,1])
18 plt.xlabel('X-axis')
19 plt.ylabel('Y-axis')
20 plt.title('2D Tensor Scatter Plot')
21 plt.show()
22
23 # Format the initial data into a loader to make a more
    efficient code :
24 batch_size = 32                                     #
    -----
    Batch size
25
26 # Wrap the tensor in a TensorDataset to assure compatibility
    in the code :
27 train_dataset = torch.utils.data.TensorDataset(train_data)
28
29 # Create a DataLoader using the dataset :
30 train_loader = torch.utils.data.DataLoader(train_dataset,
    batch_size=batch_size, shuffle=True)
31
32 # Let's start with the generator as a simple neural network
33 class Generator(nn.Module):
34
```

```

35     def __init__(self):
36         super().__init__()
37         self.model = nn.Sequential(
38             nn.Linear(2, 16), # Input is two dimensional
39             nn.ReLU(),
40             nn.Linear(16, 32),
41             nn.ReLU(),
42             nn.Linear(32, 2), # Output is two dimensional
43         )
44
45     def forward(self, x):
46         output = self.model(x)
47         return output
48
49 # Build the discriminator as a NN
50 class Discriminator(nn.Module):
51     def __init__(self):
52         super().__init__()
53         self.model = nn.Sequential(
54             nn.Linear(2, 256), #the input is two-dimensional
55             nn.ReLU(),
56             nn.Dropout(0.3), #dropout layers reduce
57                             #overfitting
58             nn.Linear(256, 128),
59             nn.ReLU(),
60             nn.Dropout(0.3),
61             nn.Linear(128, 64),
62             nn.ReLU(),
63             nn.Dropout(0.3),
64             nn.Linear(64, 1),
65             nn.Sigmoid(),#sigmoid activation to represent
66                             #probability
67         )
68
69     def forward(self, x):
70         output = self.model(x)
71         return output
72
73 # Training loop :
74 discriminator = Discriminator()
75 generator = Generator()
76 gen_loss_vector = []
77 discr_loss_vector = []
78 num_epoch_vector = []
79
80 optimizer_discriminator = torch.optim.Adam(discriminator.
81     parameters(), lr=2*lr)

```



```

81 optimizer_generator = torch.optim.Adam(generator.parameters
    (), lr=lr)
82
83 for epoch in range(num_epochs):
84     for idx, real_data_set in enumerate(train_loader):
85         real_data_set = real_data_set[0]
86         # Preparing the real data to train the discriminator
            :
87         real_data_label = torch.ones(batch_size,1)
88
89         # Preparing the fake data to train the discriminator
            :
90         noise_data_set = torch.randn((batch_size, 2))
91         fake_data_set = generator(noise_data_set)
92         fake_data_label = torch.zeros(batch_size, 1)
93
94         # Creating the training samples set:
95         training_data_set = torch.cat((real_data_set,
            fake_data_set))
96
97         # Creating the training labels set:
98         training_labels_set = torch.cat((real_data_label,
            fake_data_label))
99
100        # Train the discriminator:
101        discriminator.zero_grad()
102        output_discriminator = discriminator(
            training_data_set)
103        loss_discriminator = loss_function(
104            output_discriminator, training_labels_set)
105        loss_discriminator.backward()
106        optimizer_discriminator.step()
107
108        # Initialising the data for the discriminator:
109        noise_data_set = torch.randn((batch_size, 2))
110
111        # Train the generatot:
112        generator.zero_grad()
113        output_generator = generator(noise_data_set)
114
115        # We use the discriminator output to back propagate:
116        output_discriminator_generated = discriminator(
            output_generator)
117        loss_generator = loss_function(
118            output_discriminator_generated, real_data_label)
119        # We put label = 1 so that the error we want to
            minimize is the distance between our generated
            data and the label 1
120
121        loss_generator.backward()

```

```

122     optimizer_generator.step()
123
124     # prepares data for loss plot afterwise:
125     if idx == batch_size-1:
126         gen_loss_vector.append(float(loss_generator))
127         discr_loss_vector.append(float(
128             loss_discriminator))
129         num_epoch_vector.append(epoch)
130
131     # Show loss
132     if epoch % 100 == 0 and idx == batch_size - 1:
133         print(f"Epoch: {epoch} Loss D.: {
134             loss_discriminator}")
135         print(f"Epoch: {epoch} Loss G.: {loss_generator}
136             ")
137
138     generated_samples_for_plotting =
139         output_generator.detach()
140
141     # Plot real samples in blue
142     plt.plot(real_data_set[:, 0], real_data_set[:,
143         1], ".", color='blue')
144
145     # Plot generated samples in red
146     plt.plot(generated_samples_for_plotting[:, 0],
147         generated_samples_for_plotting[:, 1], ".",
148         color='red')
149
150     plt.show()
151
152     # Plot the functions on the same graph
153     plt.plot(num_epoch_vector, gen_loss_vector, label='
154         Generator', color = 'b') # Plot sine function with
155         label
156     plt.plot(num_epoch_vector, discr_loss_vector, label='
157         Discriminator', color = 'r') # Plot cosine function
158         with label
159     plt.legend() # Show legend with function labels
160     plt.xlabel('epoch')
161     plt.ylabel('loss')
162     plt.title('Generator and discriminator loss over epochs'
163         )
164     plt.show()

```

Listing 1: pb 1

6.2 Part 2

```

1 # Import the necessary libs for the homework
2 import torch
3 from torch import nn
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import math
7 import pandas as pd
8
9
10 # Generate the initial dataset :
11 train_len = 1024
12 train_data = torch.zeros(train_len, 2)
13
14 # Generate random angles
15 theta = 2 * math.pi * torch.rand(train_len)
16
17 # Convert polar coordinates to Cartesian coordinates
18 radius = 1
19 x = radius * torch.cos(theta)
20 y = radius * torch.sin(theta)
21
22 train_data[:,0] = x
23 train_data[:,1] = y
24
25 # Plots the content of the tensor to make sure it's what i
    want
26 plt.scatter(train_data[:,0], train_data[:,1])
27 plt.xlabel('X-axis')
28 plt.ylabel('Y-axis')
29 plt.title('2D Tensor Scatter Plot')
30 plt.show()
31
32
33 # Format the initial data into a loader to make a more
    efficient code :
34 batch_size = 32                                     #
    -----
        Batch size
35
36 # Wrap the tensor in a TensorDataset to assure compatibility
    in the code :
37 train_dataset = torch.utils.data.TensorDataset(train_data)
38
39 # Create a DataLoader using the dataset :
40 train_loader = torch.utils.data.DataLoader(train_dataset,
        batch_size=batch_size, shuffle=True)
41
42 # Let's start with the generator as a simple neural network
43 class Generator(nn.Module):

```

```

44
45     def __init__(self):
46         super().__init__()
47         self.model = nn.Sequential(
48             nn.Linear(3, 16), # Input is 3 dimensional
49             nn.ReLU(),
50             nn.Linear(16, 32),
51             nn.ReLU(),
52             nn.Linear(32, 64),
53             nn.ReLU(),
54             nn.Linear(64, 32),
55             nn.ReLU(),
56             nn.Linear(32, 2), # Output is two dimensional
57         )
58
59     def forward(self, x):
60         output = self.model(x)
61         return output
62
63 # Build the discriminator as a NN
64 class Discriminator(nn.Module):
65     def __init__(self):
66         super().__init__()
67         self.model = nn.Sequential(
68             nn.Linear(2, 256), #the input is two-dimensional
69             nn.ReLU(),
70             nn.Dropout(0.3), #dropout layers reduce
71                             #overfitting
72             nn.Linear(256, 128),
73             nn.ReLU(),
74             nn.Dropout(0.3),
75             nn.Linear(128, 64),
76             nn.ReLU(),
77             nn.Dropout(0.3),
78             nn.Linear(64, 1),
79             nn.Sigmoid(), #sigmoid activation to represent
80                             #probability
81         )
82
83     def forward(self, x):
84         output = self.model(x)
85         return output
86
87 # Sets the parameters
88 lr = 0.001
89 loss_function = nn.BCELoss()
90 num_epochs = 500
91
92 # Training loop :

```

```

92 discriminator = Discriminator()
93 generator = Generator()
94 gen_loss_vector = []
95 discr_loss_vector = []
96 num_epoch_vector = []
97
98
99 optimizer_discriminator = torch.optim.Adam(discriminator.
    parameters(), lr=2*lr)
100 optimizer_generator = torch.optim.Adam(generator.parameters
    (), lr=lr)
101
102 for epoch in range(num_epochs):
103     for idx, real_data_set in enumerate(train_loader):
104         real_data_set = real_data_set[0]
105         # Preparing the real data to train the discriminator
106         :
107         real_data_label = torch.ones(batch_size,1)
108
109         # Preparing the fake data to train the discriminator
110         :
111         noise_data_set = torch.randn((batch_size, 3))
112         fake_data_set = generator(noise_data_set)
113         fake_data_label = torch.zeros(batch_size, 1)
114
115         # Creating the training samples set:
116         training_data_set = torch.cat((real_data_set,
117             fake_data_set))
118
119         # Creating the training labels set:
120         training_labels_set = torch.cat((real_data_label,
121             fake_data_label))
122
123         # Train the discriminator:
124         discriminator.zero_grad()
125         output_discriminator = discriminator(
126             training_data_set)
127         loss_discriminator = loss_function(
128             output_discriminator, training_labels_set)
129         loss_discriminator.backward()
130         optimizer_discriminator.step()
131
132         # Initialising the data for the generator:
133         noise_data_set = torch.randn((batch_size, 3))
134
135         # Train the generatot:
136         generator.zero_grad()
137         output_generator = generator(noise_data_set)
138
139         # We use the discriminator output to back propagate:

```

```

135     output_discriminator_generated = discriminator(
136         output_generator)
137     loss_generator = loss_function(
138         output_discriminator_generated, real_data_label)
139     # We put label = 1 so that the error we want to
140     # minimize is the distance between our generated
141     # data and the label 1
142
143     loss_generator.backward()
144     optimizer_generator.step()
145
146     # prepares data for loss plot afterwise:
147     if idx == batch_size-1:
148         gen_loss_vector.append(float(loss_generator))
149         discr_loss_vector.append(float(
150             loss_discriminator))
151         num_epoch_vector.append(epoch)
152
153     # Show loss
154     if epoch % 100 == 0 and idx == batch_size - 1:
155         print(f"Epoch: {epoch} Loss D.: {
156             loss_discriminator}")
157         print(f"Epoch: {epoch} Loss G.: {loss_generator}
158             ")
159
160     generated_samples_for_plotting =
161         output_generator.detach()
162
163     # Plot real samples in blue
164     plt.plot(real_data_set[:, 0], real_data_set[:,
165         1], ".", color='blue')
166
167     # Plot generated samples in red
168     plt.plot(generated_samples_for_plotting[:, 0],
169         generated_samples_for_plotting[:, 1], ".",
170         color='red')
171
172     plt.show()
173
174     # Plot the functions on the same graph
175     plt.plot(num_epoch_vector, gen_loss_vector, label='Generator
176         ', color = 'b') # Plot sine function with label
177     plt.plot(num_epoch_vector, discr_loss_vector, label='
178         Discriminator', color = 'r') # Plot cosine function with
179         label
180     plt.legend() # Show legend with function labels
181     plt.xlabel('epoch')
182     plt.ylabel('loss')
183     plt.title('Generator and discriminator loss over epochs')

```

```
172 plt.show()
```

Listing 2: pb 1

6.3 Part 3

```
1 # Import the necessary libs for the homework
2 import torch
3 from torch import nn
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import math
7 import pandas as pd
8
9 # Generate the initial dataset :
10 train_len = 1024
11 train_data = torch.zeros(train_len, 2)
12
13 # Generate polar coordinates
14 theta = torch.linspace(0, 2 * math.pi, train_len) # 1000
15           points between 0 and 2
16 r = torch.cos(2 * theta)
17
18 # Convert to Cartesian coordinates
19 x = r * torch.cos(theta)
20 y = r * torch.sin(theta)
21
22 train_data[:,0] = r * torch.cos(theta)
23 train_data[:,1] = r * torch.sin(theta)
24
25 # Plot the points
26 plt.plot(x, y, label='( , cos(2 ))')
27 plt.scatter(x, y, color='red') # Scatter plot to highlight
28           individual points
29 plt.title('Two-Dimensional Polar Coordinates')
30 plt.xlabel('x')
31 plt.ylabel('y')
32 plt.legend()
33 plt.axis('equal')
34 plt.show()
35
36 # Format the initial data into a loader to make a more
37           efficient code :
38 batch_size = 32
39
40 -----
41           Batch size
42
43 # Wrap the tensor in a TensorDataset to assure compatibility
44           in the code :
```

```

38 train_dataset = torch.utils.data.TensorDataset(train_data)
39
40 # Create a DataLoader using the dataset :
41 train_loader = torch.utils.data.DataLoader(train_dataset,
42                                             batch_size=batch_size, shuffle=True)
43
44 # Let's start with the generator as a simple neural network
45 class Generator(nn.Module):
46
47     def __init__(self):
48         super().__init__()
49         self.model = nn.Sequential(
50             nn.Linear(2, 16), # Input is two dimensional
51             nn.ReLU(),
52             nn.Linear(16, 32),
53             nn.ReLU(),
54             nn.Linear(32, 64),
55             nn.ReLU(),
56             nn.Linear(64, 32),
57             nn.ReLU(),
58             nn.Linear(32, 2), # Output is two dimensional
59         )
60
61     def forward(self, x):
62         output = self.model(x)
63         return output
64
65 # Build the discriminator as a NN
66 class Discriminator(nn.Module):
67
68     def __init__(self):
69         super().__init__()
70         self.model = nn.Sequential(
71             nn.Linear(2, 256), #the input is two-dimensional
72             nn.ReLU(),
73             nn.Dropout(0.3), #dropout layers reduce
74                             #overfitting
75             nn.Linear(256, 128),
76             nn.ReLU(),
77             nn.Dropout(0.3),
78             nn.Linear(128, 64),
79             nn.ReLU(),
80             nn.Dropout(0.3),
81             nn.Linear(64, 1),
82             nn.Sigmoid(), #sigmoid activation to represent
83                             #probability
84         )
85
86     def forward(self, x):
87         output = self.model(x)
88         return output

```



```

85
86 # Training loop :
87
88 discriminator = Discriminator()
89 generator = Generator()
90 gen_loss_vector = []
91 discr_loss_vector = []
92 num_epoch_vector = []
93
94
95 optimizer_discriminator = torch.optim.Adam(discriminator.
96     parameters(), lr=lr_d)
97 optimizer_generator = torch.optim.Adam(generator.parameters
98     (), lr=lr_g)
99
100 for epoch in range(num_epochs):
101     for idx, real_data_set in enumerate(train_loader):
102         real_data_set = real_data_set[0]
103         # Preparing the real data to train the discriminator
104         :
105         real_data_label = torch.ones(batch_size,1)
106
107         # Preparing the fake data to train the discriminator
108         :
109         noise_data_set = torch.randn((batch_size, 2))
110         fake_data_set = generator(noise_data_set)
111         fake_data_label = torch.zeros(batch_size, 1)
112
113         # Creating the training samples set:
114         training_data_set = torch.cat((real_data_set,
115             fake_data_set))
116
117         # Creating the training labels set:
118         training_labels_set = torch.cat((real_data_label,
119             fake_data_label))
120
121         # Train the discriminator:
122         discriminator.zero_grad()
123         output_discriminator = discriminator(
124             training_data_set)
125         loss_discriminator = loss_function(
126             output_discriminator, training_labels_set)
127         loss_discriminator.backward()
128         optimizer_discriminator.step()
129
130         # Initialising the data for the discriminator:
131         noise_data_set = torch.randn((batch_size, 2))
132
133         # Train the generatot:
134         generator.zero_grad()

```

```

128     output_generator = generator(noise_data_set)
129
130     # We use the discriminator output to back propagate:
131     output_discriminator_generated = discriminator(
132         output_generator)
133     loss_generator = loss_function(
134         output_discriminator_generated, real_data_label)
135     # We put label = 1 so that the error we want to
136     # minimize is the distance between our generated
137     # data and the label 1
138
139     loss_generator.backward()
140     optimizer_generator.step()
141
142     # prepares data for loss plot afterwise:
143     if idx == 1:
144         gen_loss_vector.append(float(loss_generator))
145         discr_loss_vector.append(float(
146             loss_discriminator))
147         num_epoch_vector.append(epoch)
148
149     # Show loss
150     if epoch % 100 == 0 and idx == 1:
151         print(f"Epoch: {epoch} Loss D.: {
152             loss_discriminator}")
153         print(f"Epoch: {epoch} Loss G.: {loss_generator}
154             ")
155
156     generated_samples_for_plotting =
157         output_generator.detach()
158
159     # Plot real samples in blue
160     plt.plot(real_data_set[:, 0], real_data_set[:,
161         1], ".", color='blue')
162
163     # Plot generated samples in red
164     plt.plot(generated_samples_for_plotting[:, 0],
165         generated_samples_for_plotting[:, 1], ".",
166         color='red')
167
168     plt.show()

```

Listing 3: pb 1

6.4 Part 4

```

1 # Import the necessary libs for the homework
2 import torch

```

```

3 from torch import nn
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import math
7 import pandas as pd
8 import yfinance as yf
9 from sklearn.preprocessing import MinMaxScaler
10
11
12
13 # Import Amazon data from yfinance
14 stock_data = yf.download("AMZN", start="2015-01-01", end="
    2020-01-01")
15
16 # Extract closing volumes
17 closing_volumes = stock_data['Close']
18
19 # Choose the slicing window
20 N = 60
21
22 # Create a tensor containing the slicing windows of closing
    volumes
23 input_sequences = []
24 for i in range(len(closing_volumes) - N):
25     input_sequence = closing_volumes.iloc[i:i+N].values
26
27     # Scale the data for better convergence of the algorithm.
28     input_array = np.array(input_sequence).reshape(-1, 1)
29     scaler = MinMaxScaler(feature_range=(-1, 1)) # Scale
        between -1 and 1 because its the area where ReLu is
        the most efficient
30     scaled_data = scaler.fit_transform(input_array)
31     scaled_data_list = []
32
33     # Using a list format for algorithm compatibility after
34     for data in scaled_data:
35         scaled_data_list.append(data[0])
36
37     input_sequences.append(scaled_data_list)
38
39 train_data = torch.tensor(input_sequences, dtype=torch.
    float32)
40
41 # Convert your data to a TensorDataset using torch.
    TensorDataset for loader
42 train_dataset = torch.utils.data.TensorDataset(train_data)
43
44 batch_size = 32
45

```

```

46 # Create a DataLoader with shuffle=True for shuffling at
    each epoch
47 train_loader = torch.utils.data.DataLoader(train_dataset,
    batch_size=batch_size, shuffle=True, drop_last=True)
48
49 # Designing the architecture of the GAN
50 class Generator(nn.Module):
51
52     def __init__(self):
53         super().__init__()
54         self.model = nn.Sequential(
55             nn.Linear(N, 2*N), # Input is a N slicing
                window
56             nn.ReLU(),
57             nn.Linear(2*N, 4*N),
58             nn.ReLU(),
59             nn.Dropout(0.3), #dropout layers reduce
                overfitting
60             nn.Linear(4*N, 4*N),
61             nn.ReLU(),
62             nn.Dropout(0.3), #dropout layers reduce
                overfitting
63             nn.Linear(4*N, 8*N),
64             nn.ReLU(),
65             nn.Dropout(0.3), #dropout layers reduce
                overfitting
66             nn.Linear(8*N, 4*N),
67             nn.ReLU(),
68             nn.Dropout(0.3), #dropout layers reduce
                overfitting
69             nn.Linear(4*N, 4*N),
70             nn.ReLU(),
71             nn.Linear(4*N, 2*N),
72             nn.ReLU(),
73             nn.Linear(2*N, N), # Output is N slicing
                window
74         )
75
76     def forward(self, x):
77         output = self.model(x)
78         return output
79
80 # Build the discriminator as a NN
81 class Discriminator(nn.Module):
82     def __init__(self):
83         super().__init__()
84         self.model = nn.Sequential(
85             nn.Linear(N, 2*N), #the input is lenght N
86             nn.ReLU(),

```

```

87         nn.Dropout(0.3), #droput layers reduce
            overfitting
88         nn.Linear(2*N, 4*N),
89         nn.ReLU(),
90         nn.Dropout(0.3),
91         nn.Linear(4*N, 4*N),
92         nn.ReLU(),
93         nn.Dropout(0.3),
94         nn.Linear(4*N, 2*N),
95         nn.ReLU(),
96         nn.Dropout(0.3),
97         nn.Linear(2*N, N),
98         nn.ReLU(),
99         nn.Dropout(0.3),
100        nn.Linear(N, 1),
101        nn.Sigmoid(), # sigmoid activation to represent
            probability
102    )
103
104    def forward(self, x):
105        output = self.model(x)
106        return output
107
108    from tqdm import tqdm
109    # Training loop :
110    num_epochs = 1000
111    discriminator = Discriminator()
112    generator = Generator()
113    gen_loss_vector = []
114    discr_loss_vector = []
115    num_epoch_vector = []
116
117
118    optimizer_discriminator = torch.optim.Adam(discriminator.
        parameters(), lr=lr_d)
119    optimizer_generator = torch.optim.Adam(generator.parameters
        ()), lr=lr_g)
120
121
122    for epoch in tqdm(range(num_epochs)):
123
124        for index, batch in enumerate(train_loader):
125            real_data_set = batch[0]
126
127            # Preparing the real data to train the discriminator
128            :
129            real_data_label = torch.ones(batch_size,1)
130
131            # Preparing the fake data to train the discriminator
132            :

```

```

131     noise_data_set = torch.randn((batch_size, N))
132     fake_data_set = generator(noise_data_set)
133     fake_data_label = torch.zeros(batch_size, 1)
134
135     # Creating the training samples set:
136     training_data_set = torch.cat((real_data_set,
137                                     fake_data_set))
138
139     # Creating the training labels set:
140     training_labels_set = torch.cat((real_data_label,
141                                     fake_data_label))
142
143     # Train the discriminator:
144     discriminator.zero_grad()
145     output_discriminator = discriminator(
146         training_data_set)
147     loss_discriminator = loss_function(
148         output_discriminator, training_labels_set)
149     loss_discriminator.backward()
150     optimizer_discriminator.step()
151
152     # Initialising the data for the gznzrator:
153     noise_data_set = torch.randn((batch_size, N))
154
155     # Train the generatot:
156     generator.zero_grad()
157     output_generator = generator(noise_data_set)
158     output_discriminator_generated = discriminator(
159         output_generator)
160     loss_generator = loss_function(
161         output_discriminator_generated, real_data_label)
162     # We put label = 1 so that the error we want to
163     # minimize is the distance between our generated
164     # data and the label 1
165     loss_generator.backward()
166     optimizer_generator.step()
167
168     # prepares data for loss plot afterwise:
169     if index == 0:
170         gen_loss_vector.append(float(loss_generator))
171         discr_loss_vector.append(float(
172             loss_discriminator))
173         num_epoch_vector.append(epoch)
174
175     # Show loss
176     if epoch % 50 == 0 and index == 0:
177         print(f"Epoch: {epoch} Loss D.: {
178             loss_discriminator}")

```

```

171     print(f"Epoch: {epoch} Loss G.: {loss_generator}
172           ")
173
174     # Let's plot the first time series of the batch
175     # in blue
176     time_steps = []
177     for time in range(len(real_data_set[0])):
178         time_steps.append(time)
179     plt.plot(time_steps, real_data_set[0], "-", color='
180             blue')
181
182     generated_samples_for_plotting =
183         output_generator.detach()[0]
184
185     # Plot generated samples in red
186     plt.plot(time_steps,
187              generated_samples_for_plotting, "-", color='
188              red')
189     plt.show()
190
191     # let's plot the lag 1 return distribution
192     # Calculate lag-1 returns
193     lag_1_returns = generated_samples_for_plotting
194                     [1:] - generated_samples_for_plotting[:-1]
195
196     # Plot the distribution of lag-1 returns
197     plt.hist(lag_1_returns, bins=50, color='green',
198             alpha=0.7)
199     plt.xlabel('Lag-1 Returns')
200     plt.ylabel('Frequency')
201     plt.title('Distribution of Lag-1 Returns of
202             Generated Samples')
203     plt.show()

```

Listing 4: pb 1