Project 3: Generative Adversarial Networks

1 INTRODUCTION

Generative Adversarial Networks (GANs) are machine learning models capable of generating images (or any other data) similar to real-world samples. As shown in Figure 1.1, GANs are two-part networks, which consist of a discriminator and a generator which behave as a policeman and counterfitter, respectively. The generator, or counterfitter, generates images from a D-dimensional noise vector in an attempt to mimic the real-world data, and the discriminator, or policeman, behaves as a binary classifier and learns to discern the true 'real' images and the generated 'fake' images. The generator's objective is to fool the discriminator into classifying the generated data as 'real' while the discriminator is simultaneously learning how to distinguish the 'real' and 'fake' data. The objective of both the generator and discriminator is to degrade the performance of the alternate model. By training the two models against each other in this fashion, the generator will learn, from the discriminator, which features are the most prevalent in the 'real' data and will eventually produce comparable images.

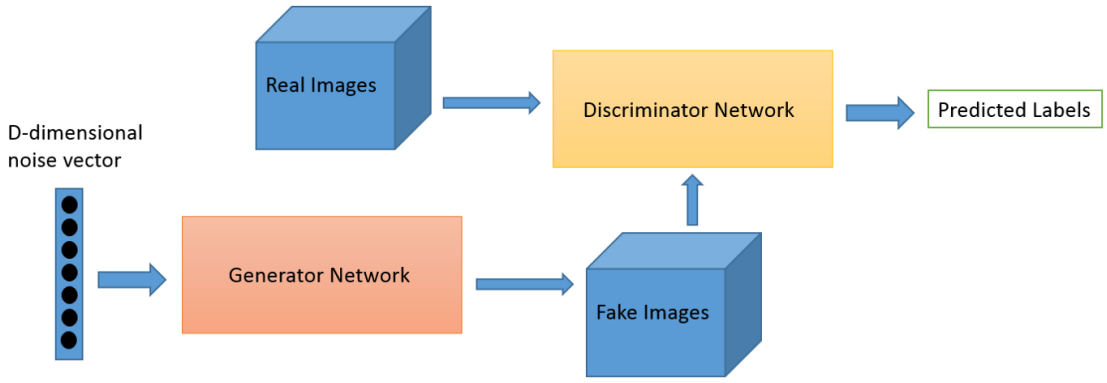

Figure 1.1: An overview of a simple GAN.

In the context of GANs, both the discriminator and generator can be typical artificial neural network models consisting of various layers, activation functions, initializations, etc. However, contrary to training a neural network, GANs are trained with different data in each epoch. Training a GAN begins with generating 'fake' image samples, using the generator, from D-dimensional noise. Then, a random subset of 'real' images are selected from the available training data. The generated 'fake' and 'real' images are then combined into a single batch and labeled as either 0 or 1 based on whether the image is 'fake' or 'real,' respectively. [1] The training labels, as well as the output of the discriminator, can be interpreted as the the network's confidence of the image being 'real.' The discriminator is trained on the aforementioned batch using one epoch while keeping the parameters of the generator constant. Then, a new batch of

[1] In practice, the 'real' images can sometimes be labeled something closer to 0.9 to avoid the vanishing gradient problem during training.

images is generated and labeled as 'real.' The mislabeled batch of data is then used to train the generator portion of the GAN using one epoch while the parameters of the discriminator are held constant. Repeating this process for several iterations allows the generator to eventually mimic the properties of the 'real' data by learning from the discriminator portion of the GAN.

The objective of this project is to implement a GAN to produce images that resemble samples found in the MNIST dataset of hand written digits. We will train a generative model capable of creating images resembling human hand-written digits from a 100-dimensional noise vector. After training a GAN to satisfy this task, we will generate sample images using the generator, label them manually, and use a pre-trained artificial neural network to classify them. The accuracy of your samples will, somewhat arbitrarily, indicate the robustness of your generative model.

For this project, the MNIST data can be loaded using Keras (as in Project 2) and the classifier required in Part 3 is provided to you as a .h5 file. You are expected to fill in the Jupyter Notebook provided to you as instructed in each of the following sections. **Submit your final code as a single .ipynb file. Ensure that the deliverables are clearly shown for each part to earn full credit. Name your submission *lastname_firstname_gan.ipynb*.**

2 EXAMPLE GAN CODE

The following listing shows how to build the discriminator, generator, and GAN model using Keras in Python. You should adopt the following code into your own solution (with the appropriate architecture) as needed.

Listing 1: Implementing a GAN in Keras

```
#ECE 595 Machine Learning II
#Project 3: GAN – Example Code
#Written by: Rajeev Sahay and Aly El Gamal

from keras.layers import Dense, Input
from keras.models import Model, Sequential
from keras.layers.advanced_activations import LeakyReLU
from keras.optimizers import adam

# The D-dimensional noise vector length
latent_dim = 500

# Dimension of generaed output
data_dim = 1000

# Optimizer for discriminator, which will have a higher learning rate
# than adversarial model
def dis_optimizer():
    return adam(lr=0.2, beta_1=0.5)

def gan_optimizer():
    return adam(lr=0.1)

# Genrerator model
def create_generator():
    generator=Sequential()
    generator.add(Dense(25, input_dim=latent_dim))
    generator.add(LeakyReLU(0.5))
    generator.add(Dense(50))
    generator.add(LeakyReLU(0.5))
    generator.add(Dense(data_dim, activation='tanh'))

    return generator

# Discriminator model
def create_discriminator():
    discriminator=Sequential()
    discriminator.add(Dense(100, input_dim=data_dim))
    discriminator.add(LeakyReLU(0.5))
    discriminator.add(Dense(100))
    discriminator.add(LeakyReLU(0.5))
    discriminator.add(Dense(units=1, activation='sigmoid'))
    discriminator.compile(loss='choose_loss_fcn',
                          optimizer=dis_optimizer(),
```

```
45                                  metrics=['accuracy'])
46
47      return discriminator
48
49  # Create adversarial model
50  def create_gan(discriminator, generator):
51
52      # Do not allow discriminator paramaters to be updated
53      discriminator.trainable = False
54      gan_input = Input(shape=(latent_dim,))
55      x = generator(gan_input)
56      gan_output = discriminator(x)
57      gan = Model(inputs=gan_input, outputs=gan_output)
58      gan.compile(loss='choose_loss_fcn',
59                  optimizer=gan_optimizer(),
60                  metrics=['accuracy'])
61
62      return gan
63
64  # Creating graph for GAN
65  generator = create_generator()
66  discriminator = create_discriminator()
67  gan = create_gan(discriminator, generator)
```

The following pseudo-code outlines the algorithm that should be implemented when training a GAN. Ideally, this algorithm will be implemented in your code directly after Line 69 from Listing 1 above.

Algorithm 1 GAN Pseudo-Code

```
epochs ← x.                                          ▷ x: number of training steps
batch_size ← y                                       ▷ y: batch size
for epochs do
    noise ← gaussian_noise      ▷ Dimensions of Gaussian noise: batch_size × latent_dim
    fake_images ← generator(noise)
    real_images ← random subset of batch_size samples from training set
    data_total ← concatenate(real_images, fake_images)
    labels_real ← ones(batch_size)
    labels_fake ← zeros(batch_size)
    labels_discriminator ← concatenate(labels_real, labels_fake)
    discriminator.trainable ← True
    discriminator.train(data_total, labels_discriminator)
    noise ← gaussian_noise                           ▷ New noise vectors
    labels_generator ← ones(batch_size)
    discriminator.trainable ← False
    generator.train(noise, labels_generator)
```

3 PART 1: IMPLEMENTING THE GAN

In this part, you will implement a GAN capable of generating MNIST-like hand-written digits from 100-dimensional Gaussian noise vectors. Implement your generator and discriminator according to the following architectures below. The learning rate of the GAN (stacked generator and discriminator) should be higher than the learning rate of the discriminator.

Begin by importing the MNIST data from Keras and normalizing the training set to lie in $[-1,1]$. Then, create a function that returns the Adam optimizer for the discriminator (set the learning rate to 0.0002, beta_1 to 0.5 and beta_2 to its default 0.999). Create another function that returns the Adam optimizer for the GAN model (set the learning rate to 0.001, beta_1 to its default 0.9, and beta_2 to its default 0.999). Use the following architectures for the generator and discriminator. Note that the 'real' input data is normalized to lie in $[-1,1]$ so a hyperbolic tangent activation is used on the output layer of the discriminator to maintain the same range for the generated images. Before plotting, the images should be re-scaled to $[0,1]$.

Generator architecture:

- Input layer: a 100-dimensional noise vector
- First hidden layer: a dense (fully-connected) layer consisting of 200 - 300 units with a LeakyReLU activation
- Second hidden layer: a dense (fully-connected) layer consisting of 500 - 600 units with a LeakyReLU activation
- Third hidden layer: a dense (fully-connected) layer consisting of 1000 - 1200 units with a LeakyReLU activation
- Output layer: the 784-dimensional generated image with a hyperbolic tangent activation
- Compile the model with the appropriate loss and optimizer

Discriminator architecture:

- Input layer: a 784-dimensional vector of the image data
- First hidden layer: a dense (fully-connected) layer consisting of 1000 - 1200 units with a LeakyReLU activation (add a 'Dropout' layer to avoid overfitting)
- Second hidden layer: a dense (fully-connected) layer consisting of 500 - 600 units with a LeakyReLU activation (add a 'Dropout' layer to avoid overfitting)
- Third hidden layer: a dense (fully-connected) layer consisting of 200 - 300 units with a LeakyReLU activation
- Output layer: a single Dense unit with a sigmoid activation indicating whether the image is 'real' or 'fake'
- Compile the model with the appropriate loss and optimizer

After creating and initializing the GAN graphs, train the GAN by adopting Algorithm 1 using **100,000 epochs**. Set the **sample_interval to 10,000** so that the progress can be visually examined (while debugging, you may want to use smaller epochs, such as 5, and small samples_intervals such as 1, to see if your code can run). Finally, during training, save the accuracy and loss of both the discriminator and GAN so that you can plot them after training.

Deliverables for Part 1:

1. 10 sets (one every 10,000 epochs) each containing 25 sample images that were generated.
2. Show a plot of the Discriminator and GAN model loss vs. epoch
3. Show a plot of Discriminator and GAN model accuracy vs. epoch
4. **Assessing importance of Dropout in the discriminator**: For 1,2 and 3, if your discriminator has dropout layers, remove them, without changing any other parameters in the network and train the network. Observe the generated images and compare & comment on the results with 1. If you did not have dropout layers, add a dropout layer, without changing any other parameters and train the network.Compare & comment on the results with 1.
5. **Importance of hyper-parameter tuning in GANs**:
    a) Increase or decrease the dropout rate in each dropout layer by 0.1 and train the network by keeping all the other hyper-parameters the same. Observe and comment on the results.
    b) **Bonus:** With the new dropout rate in every dropout layer, find other hyper-parameter settings which produce similar or better results than 1.
6. Answer the following questions:
    a) Why does the accuracy of the discriminator remain around 50%? Is this a good trait of the GAN?
    b) How could this model be modified to produce cleaner (less noisy) images?

4 PART 2: GENERATING SAMPLES

Generate and show **ten images** using the trained generator from Part 1. Additionally, show the noise, as images, used to generate the ten samples. Finally, visually examine the ten generated samples and fill in their labels as indicated in Part 3 of the code. Simply use the digit as the label. For example, if the digits look like a 2, 3, 1, 5, 6, and 7, then use the following line:

```
labels = [2, 3, 1, 5, 6, 7]
```
[*1*]

The line of code directly following these labels will turn these integer labels into one-hot labels.

Deliverables for Part 2:

1. Show ten sample images of the noise used for creating the images.
2. Show ten sample images that were created from the trained generator

5 PART 3: ASSESSING THE GENERATOR

In Part 3, you are going to use a a classifier pre-trained on real MNIST data to classify the generated images from Part 2. This classifier is available to you as 'mnist_classifier.h5.' After filling in the labels to the generated images from Part 2, load the MNIST classifier into your Notebook by selecting Files in the left sidebar and then clicking Upload and uploading 'mnist_classifier.h5,' which was made available with this project. Then, print the classifications and evaluate the accuracy of the generated images using the given classifier.

Deliverables for Part 3:

1. Print the classifications (from the given pre-trained classifier) for each of the ten generated images
2. Print the accuracy of the pre-trained model on the ten generated samples
3. Answer the following questions:

    a) State the accuracy of the classifier on your ten generated images. Based on this accuracy, would you say your generator does well in producing images comparable to those in the MNIST dataset of hand-written digits? Why or why not?

    b) In this project, we only tested the performance of the pre-trained classifier on ten samples and used its result to determine the robustness of the generator. How could we better assess the quality of the generated images using this pre-trained classifier and the saved generator?