Generative Adversarial Networks with Python

Deep Learning Generative Models for Image Synthesis and Image Translation

Jason Brownlee

MACHINE LEARNING MASTERY

Disclaimer

Acknowledgements

Special thanks to my proofreader Sarah Martin and my technical editors Arun Koshy, Andrei Cheremskoy, and Michael Sanderson.

Copyright

Edition: v1.1

Contents

Preface

The study of Generative Adversarial Networks (GANs) is new, just a few years old. Yet, in just a few years GANs have achieved results so remarkable that they have become the state-of-the-art in generative modeling. The field is so new that there are no good theories on how to implement and configure the models. All advice for applying GAN models is based on hard-earned empirical findings, the same as any nascent field of study. This makes it both exciting and frustrating.

It's exciting because although the results achieved so far are significant. Such as the automatic synthesis of large photo-realistic faces and translation of photographs from day to night. We have only scratched the surface on the capabilities of these methods. It's frustrating because the models are fussy and prone to failure modes, even after all care is taken in the choice of model architecture, model configuration hyperparameters, and data preparation.

I carefully designed this book to give you the foundation required to begin developing and applying generative adversarial networks quickly. We skip over many false starts and interesting diversions in the field and focus on the precise tips, tricks, and hacks you need to know in order to develop and train stable GAN models. We also visit state-of-the-art models and discover how we can achieve the same impressive results ourselves. There are three key areas that you must understand when working with GANs; they are:

- **How to develop basic GAN models**. This includes training unsupervised GANs for image synthesis and the heuristics required to configure and successfully train the generative models.
- **How to evaluate GAN models**. This includes widely adopted qualitative and quantitative techniques for evaluating the quality of synthesized images.
- **How to develop alternate GAN models**. This includes the use of alternate loss functions and the use of class-conditional information in the models.

These three key areas are a required foundation for understanding and developing the more advanced GAN techniques. We then build upon these key areas and explore state-of-the-art models for image translation: specifically the Pix2Pix and CycleGAN models. These key topics provide the backbone for the book and the tutorials you will work through. I believe that after completing this book, you will have the skills and knowledge required to bring generative adversarial networks to your own projects.

Jason Brownlee
2019

Introduction

Welcome

Welcome to *Generative Adversarial Networks with Python*. Generative Adversarial Networks, or GANs for short, are a deep learning technique for training generative models. GANs are most commonly used for the generation of synthetic images for a specific domain that are different and practically indistinguishable from other real images. The study and application of GANs are only a few years old, yet the results achieved have been nothing short of remarkable. Because the field is so young, it can be challenging to know how to get started, what to focus on, and how to best use the available techniques. This book is designed to teach you step-by-step how to bring the best and most promising aspects of the exciting field of Generative Adversarial Networks to your own projects.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that know some deep learning. Maybe you want or need to start using Generative Adversarial Networks on your research project or on a project at work. This guide was written to help you do that quickly and efficiently by compressing years of knowledge and experience into a laser-focused course of hands-on tutorials. The lessons in this book assume a few things about you, such as:

- You know your way around basic Python for programming.
- You know your way around basic NumPy for array manipulation.
- You know your way around basic Keras for deep learning.

For some bonus points, perhaps some of the below criteria apply to you. Don't panic if they don't.

- You may know how to work through a predictive modeling problem end-to-end.
- You may know a little bit of computer vision, such as convolutional neural networks.

This guide was written in the top-down and results-first machine learning style that you're used to from MachineLearningMastery.com.

About Your Outcomes

This book will teach you how to get results as a machine learning practitioner interested in using Generative Adversarial Networks on your computer vision project. After reading and working through this book, you will know:

- How to use upsampling and inverse convolutional layers in deep convolutional neural network models.
- How to implement the training procedure for fitting GAN models with the Keras deep learning library.
- How to implement best practice heuristics for the successful configuration and training of GAN models.
- How to develop and train simple GAN models for image synthesis for black and white and color images.
- How to explore the latent space for image generation with point interpolation and vector arithmetic.
- How to evaluate GAN models using qualitative and quantitative measures, such as the inception score.
- How to train GAN models with alternate loss functions, such as least squares and Wasserstein loss.
- How to structure the latent space and influence the generation of synthetic images with conditional GANs.
- How to develop image translation models with Pix2Pix for paired images and CycleGAN for unpaired images.
- How sophisticated GAN models, such as Progressive Growing GAN, are used to achieve remarkable results.

This book will NOT teach you how to be a research scientist, nor all the theory behind why specific methods work. For that, I would recommend good research papers and textbooks. See the *Further Reading* section at the end of each tutorial for a solid starting point.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific method or type of problem, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then

applying your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed around major techniques that are directly relevant to Generative Adversarial Networks. There are a lot of things you could learn about GANs, from theory to abstract concepts to APIs. My goal is to take you straight to developing an intuition for the elements you must understand with laser-focused tutorials. The tutorials were designed to focus on how to get results. As such, the tutorials give you the tools to both rapidly understand and apply each technique or operation. There is a mixture of both tutorial lessons and projects to both introduce the methods and give plenty of examples and opportunities to practice using them.

Each of the tutorials is designed to take you about one hour to read through and complete, excluding the execution time of some of the larger models, as well as extensions and further reading sections. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I would recommend picking a schedule and sticking to it. The tutorials are divided into seven parts; they are:

- **Part 1: Foundations**. Discover the convolutional layers for upsampling, the GAN architecture, the algorithms for training the model, and the best practices for configuring and training GANs.
- **Part 2: GAN Basics**. Discover how to develop GANs starting with a 1D GAN, progressing through black and white and color images and ending with performing vector arithmetic in latent space.
- **Part 3: GAN Evaluation**. Discover qualitative and quantitative methods for evaluating GAN models based on their generated images.
- **Part 4: GAN Loss**. Discover alternate loss functions for GAN models and how to implement some of the more widely used approaches.
- **Part 5: Conditional GANs**. Discover how to incorporate class conditional information into GANs and add controls over the types of images generated by the model.
- **Part 6: Image Translation**. Discover advanced GAN models used for image translation both with and without paired examples in the training data.
- **Part 7. Advanced GANs**. Discover the advanced GAN models that push the limits of the architecture and are the basis for some of the impressive state-of-the-art results.

Each part targets specific learning outcomes, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of digressions.

The tutorials were not designed to teach you everything there is to know about each of the methods. They were designed to give you an understanding of how they work, how to use them, and how to interpret the results the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. For this reason, the examples are highly targeted.

- Models were demonstrated on real-world datasets to give you the context and confidence to bring the techniques to your own projects.
- Model configurations used were based on best practices but were not optimized. This leaves the door open for you to explore new and possibly better configurations.
- Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties needed beyond the installation of the required packages.

A complete working example is presented with each tutorial for you to inspect and copy-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead.

Neural network algorithms are stochastic. This means that they will make different predictions when the same model configuration is trained on the same training data. On top of that, each experimental problem in this book is based around generating stochastic predictions. As a result, this means you will not get exactly the same sample output presented in this book. This is by design. I want you to get used to the stochastic nature of the neural network algorithms. If this bothers you, please note:

- You can re-run a given example a few times and your results should be close to the values reported.
- You can make the output consistent by fixing the NumPy and TensorFlow random number seed.
- You can develop a robust estimate of the skill of a model by fitting and evaluating it multiple times and taking the average of the final skill score (highly recommended).

All code examples were tested on a POSIX-compatible machine with Python 3 and Keras 2 with the TensorFlow backend. All code examples will run on modest and modern computer hardware and were executed on a CPU or GPU. A GPUs is not required but is highly recommended for most of the presented examples. Advice on how to access cheap GPUs via cloud computing is provided in the appendix. I am only human and there may be a bug in the sample code. If you discover a bug, please let me know so I can fix it, correct the book, and send out a free update.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.
- Books and book chapters.
- Web Pages and Articles.
- API documentation.
- Open Source Projects.

Wherever possible, I try to list and link to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I try to list papers that are first to use a specific technique or first in a specific problem domain. These are not required reading but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on `arxiv.org`. You can search for and download any of the papers listed on Google Scholar Search `scholar.google.com`. Wherever possible, I have tried to link to books on Amazon. I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry; you are not alone.

- **Help with a Technique?** If you need help with the technical aspects of a specific operation or technique, see the *Further Reading* section at the end of each tutorial.
- **Help with APIs?** If you need help with using the Keras library, see the list of resources in the *Further Reading* section at the end of each lesson, and also see *Appendix A*.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.
- **Help running large models?** I recommend renting time on Amazon Web Service (AWS) EC2 instances to run large models. If you need help getting started on AWS, see the tutorial in *Appendix C*.
- **Help in general?** You can shoot me an email. My details are in *Appendix A*.

Summary

Are you ready? Let's dive in! Next up, you will discover a concrete overview of Generative Adversarial Networks.

Chapter 1

How to Develop a DCGAN for Grayscale Handwritten Digits

Generative Adversarial Networks, or GANs, are an architecture for training generative models, such as deep convolutional neural networks for generating images. Developing a GAN for generating images requires both a discriminator convolutional neural network model for classifying whether a given image is real or generated and a generator model that uses inverse convolutional layers to transform an input to a full two-dimensional image of pixel values.

It can be challenging to understand both how GANs work and how deep convolutional neural network models can be trained in a GAN architecture for image generation. A good starting point for beginners is to practice developing and using GANs on standard image datasets used in the field of computer vision, such as the MNIST handwritten digit dataset. Using small and well-understood datasets means that smaller models can be developed and trained quickly, allowing the focus to be put on the model architecture and image generation process itself. In this tutorial, you will discover how to develop a generative adversarial network with deep convolutional networks for generating handwritten digits. After completing this tutorial, you will know:

- How to define and train the standalone discriminator model for learning the difference between real and fake images.
- How to define the standalone generator model and train the composite generator and discriminator model.
- How to evaluate the performance of the GAN and use the final standalone generator model to generate new images.

Let's get started.

1.1 Tutorial Overview

This tutorial is divided into seven parts; they are:

1. MNIST Handwritten Digit Dataset
2. How to Define and Train the Discriminator Model

3. How to Define and Use the Generator Model
4. How to Train the Generator Model
5. How to Evaluate GAN Model Performance
6. Complete Example of GAN for MNIST
7. How to Use the Final Generator Model

1.2 MNIST Handwritten Digit Dataset

The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 70,000 small square 28 × 28 pixel grayscale images of handwritten single digits between 0 and 9. The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively. Keras provides access to the MNIST dataset via the `mnist.load_dataset()` function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The example below loads the dataset and summarizes the shape of the loaded dataset.

Note: the first time you load the dataset, Keras will automatically download a compressed version of the images and save them under your home directory in `~/.keras/datasets/`. The download is fast as the dataset is only about eleven megabytes in its compressed form.

```
# example of loading the mnist dataset
from keras.datasets.mnist import load_data
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()
# summarize the shape of the dataset
print('Train', trainX.shape, trainy.shape)
print('Test', testX.shape, testy.shape)
```

Listing 1.1: Example of loading and summarizing the MNIST dataset.

Running the example loads the dataset and prints the shape of the input and output components of the train and test splits of images. We can see that there are 60K examples in the training set and 10K in the test set and that each image is a square of 28 by 28 pixels.

```
Train (60000, 28, 28) (60000,)
Test (10000, 28, 28) (10000,)
```

Listing 1.2: Example output from loading and summarizing the MNIST dataset.

The images are grayscale with a black background (0 pixel value) and the handwritten digits in white (pixel values near 255). This means if the images were plotted, they would be mostly black with a white digit in the middle. We can plot some of the images from the training dataset using the Matplotlib library using the `imshow()` function and specify the color map via the `cmap` argument as `'gray'` to show the pixel values correctly.

```
...
# plot raw pixel data
pyplot.imshow(trainX[i], cmap='gray')
```

Listing 1.3: Example of plotting a single image using the gray color map.

Alternately, the images are easier to review when we reverse the colors and plot the background as white and the handwritten digits in black. They are easier to view as most of the image is now white with the area of interest in black. This can be achieved using a reverse grayscale color map, as follows:

```
...
# plot raw pixel data
pyplot.imshow(trainX[i], cmap='gray_r')
```

Listing 1.4: Example of plotting a single image using the reverse gray color map.

The example below plots the first 25 images from the training dataset in a 5 by 5 square.

```
# example of loading the mnist dataset
from keras.datasets.mnist import load_data
from matplotlib import pyplot
# load the images into memory
(trainX, trainy), (testX, testy) = load_data()
# plot images from the training dataset
for i in range(25):
  # define subplot
  pyplot.subplot(5, 5, 1 + i)
  # turn off axis
  pyplot.axis('off')
  # plot raw pixel data
  pyplot.imshow(trainX[i], cmap='gray_r')
pyplot.show()
```

Listing 1.5: Example of plotting images from the MNIST dataset.

Running the example creates a plot of 25 images from the MNIST training dataset, arranged in a 5×5 square.

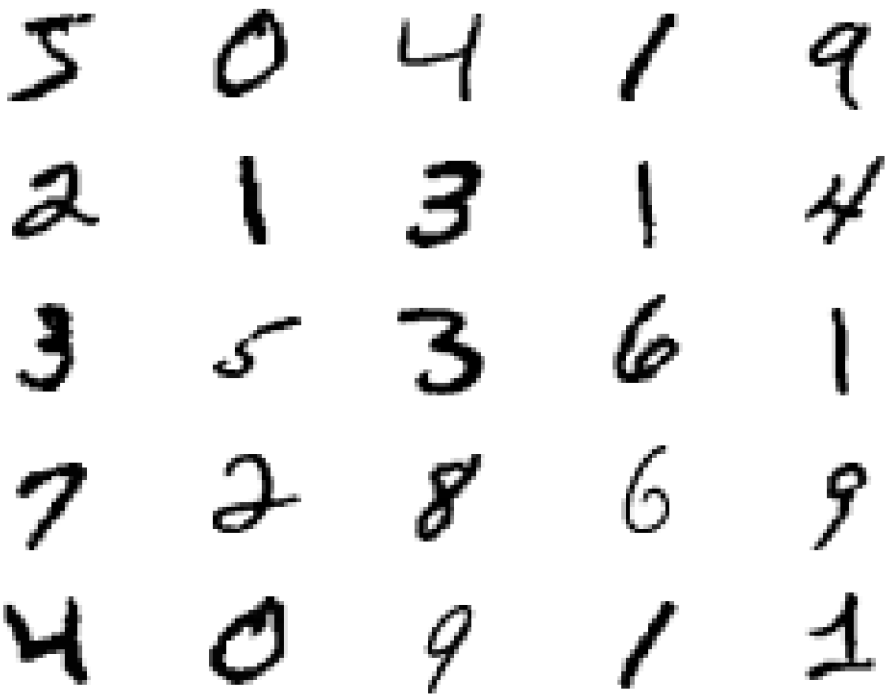

Figure 1.1: Plot of the First 25 Handwritten Digits From the MNIST Dataset.

We will use the images in the training dataset as the basis for training a Generative Adversarial Network. Specifically, the generator model will learn how to generate new plausible handwritten digits between 0 and 9, using a discriminator that will try to distinguish between real images from the MNIST training dataset and new images output by the generator model. This is a relatively simple problem that does not require a sophisticated generator or discriminator model, although it does require the generation of a grayscale output image.

1.3 How to Define and Train the Discriminator Model

The first step is to define the discriminator model. The model must take a sample image from our dataset as input and output a classification prediction as to whether the sample is real or fake. This is a binary classification problem:

- **Inputs**: Image with one channel and 28×28 pixels in size.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

The discriminator model has two convolutional layers with 64 filters each, a small kernel size of 3, and larger than normal stride of 2. The model has no pooling layers and a single node in the output layer with the sigmoid activation function to predict whether the input sample is real

or fake. The model is trained to minimize the binary cross-entropy loss function, appropriate for binary classification. We will use some best practices in defining the discriminator model, such as the use of `LeakyReLU` instead of ReLU, using `Dropout`, and using the Adam version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5. The function `define_discriminator()` below defines the discriminator model and parametrizes the size of the input image.

```
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

Listing 1.6: Example of a function for defining the discriminator model.

We can use this function to define the discriminator model and summarize it. The complete example is listed below.

```
# example of defining the discriminator model
from keras.models import Sequential
from keras.optimizers import Adam
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define model
model = define_discriminator()
```

```
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='discriminator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 1.7: Example of defining and summarizing the discriminator model.

Running the example first summarizes the model architecture, showing the input and output from each layer. We can see that the aggressive 2×2 stride acts to downsample the input image, first from 28×28 to 14×14, then to 7×7, before the model makes an output prediction. This pattern is by design as we do not use pooling layers and use the large stride as to achieve a similar downsampling effect. We will see a similar pattern, but in reverse, in the generator model in the next section.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 14, 14, 64)        640
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 3137
=================================================================
Total params: 40,705
Trainable params: 40,705
Non-trainable params: 0
_________________________________________________________________
```

Listing 1.8: Example output from defining and summarizing the discriminator model.

A plot of the model is also created and we can see that the model expects two inputs and will predict a single output.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

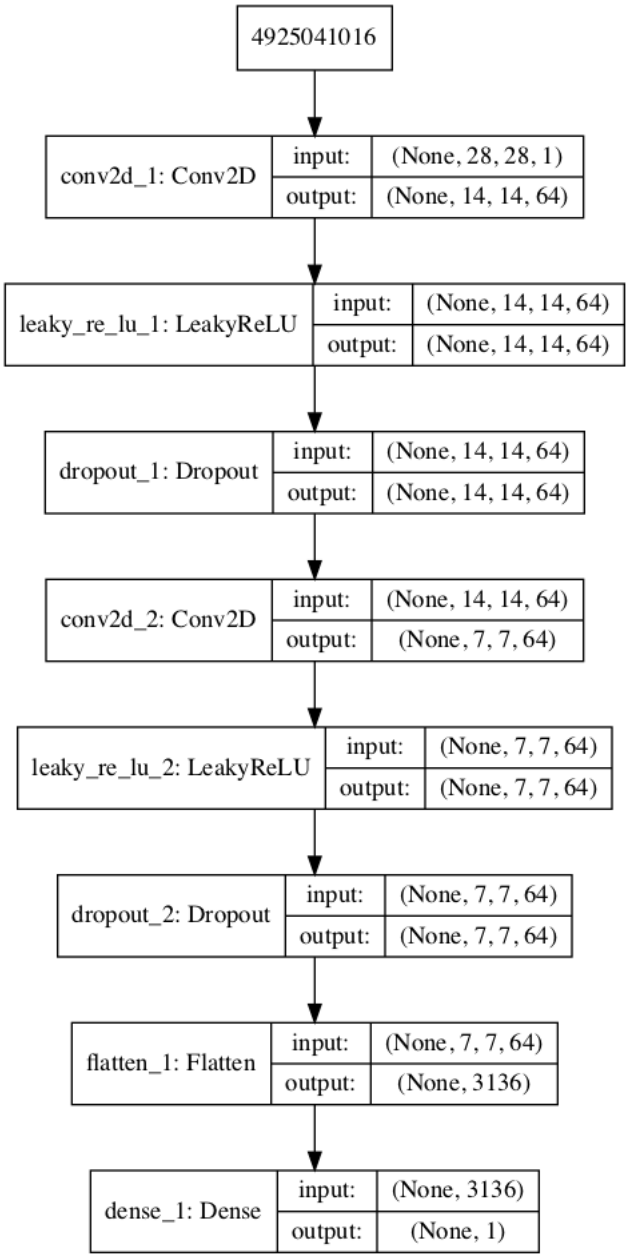

Figure 1.2: Plot of the Discriminator Model in the MNIST GAN.

We could start training this model now with real examples with a class label of one, and randomly generated samples with a class label of zero. The development of these elements will be useful later, and it helps to see that the discriminator is just a normal neural network model for binary classification. First, we need a function to load and prepare the dataset of real images. We will use the `mnist.load_data()` function to load the MNIST dataset and just use the input part of the training dataset as the real images.

```
...
# load mnist dataset
(trainX, _), (_, _) = load_data()
```

Listing 1.9: Example of loading the MNIST training dataset.

The images are 2D arrays of pixels and convolutional neural networks expect 3D arrays of images as input, where each image has one or more channels We must update the images to have an additional dimension for the grayscale channel. We can do this using the `expand_dims()` NumPy function and specify the final dimension for the channels-last image format.

```
...
# expand to 3d, e.g. add channels dimension
X = expand_dims(trainX, axis=-1)
```

Listing 1.10: Example of adding a channels dimension to the dataset.

Finally, we must scale the pixel values from the range of unsigned integers in [0,255] to the normalized range of [0,1]. It is best practice to use the range [-1,1], but in this case the range [0,1] works just fine.

```
# convert from unsigned ints to floats
X = X.astype('float32')
# scale from [0,255] to [0,1]
X = X / 255.0
```

Listing 1.11: Example of normalizing pixel values.

The `load_real_samples()` function below implements this.

```
# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [0,1]
  X = X / 255.0
  return X
```

Listing 1.12: Example of a function for loading and preparing the MNIST training dataset.

The model will be updated in batches, specifically with a collection of real samples and a collection of generated samples. On training, an epoch is defined as one pass through the entire training dataset. We could systematically enumerate all samples in the training dataset, and that is a good approach, but good training via stochastic gradient descent requires that the training dataset be shuffled prior to each epoch. A simpler approach is to select random samples of images from the training dataset. The `generate_real_samples()` function below will take the training dataset as an argument and will select a random subsample of images; it will also return class labels for the sample, specifically a class label of 1, to indicate real images.

```
# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y
```

Listing 1.13: Example of a function for selecting a sample of real images.

Now, we need a source of fake images. We don't have a generator model yet, so instead, we can generate images comprised of random pixel values, specifically random pixel values in the range [0,1] like our scaled real images. The `generate_fake_samples()` function below implements this behavior and generates images of random pixel values and their associated class label of 0, for fake.

```
# generate n fake samples with class labels
def generate_fake_samples(n_samples):
  # generate uniform random numbers in [0,1]
  X = rand(28 * 28 * n_samples)
  # reshape into a batch of grayscale images
  X = X.reshape((n_samples, 28, 28, 1))
  # generate 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

Listing 1.14: Example of a function for generating random fake images.

Finally, we need to train the discriminator model. This involves repeatedly retrieving samples of real images and samples of generated images and updating the model for a fixed number of iterations. We will ignore the idea of epochs for now (e.g. complete passes through the training dataset) and fit the discriminator model for a fixed number of batches. The model will learn to discriminate between real and fake (randomly generated) images rapidly, therefore, not many batches will be required before it learns to discriminate perfectly.

The `train_discriminator()` function implements this, using a batch size of 256 images where 128 are real and 128 are fake each iteration. We update the discriminator separately for real and fake examples so that we can calculate the accuracy of the model on each sample prior to the update. This gives insight into how the discriminator model is performing over time.

```
# train the discriminator model
def train_discriminator(model, dataset, n_iter=100, n_batch=256):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))
```

Listing 1.15: Example of a function for training the discriminator model.

Tying all of this together, the complete example of training an instance of the discriminator model on real and generated (fake) images is listed below.

```
# example of training the discriminator model on real and random mnist images
from numpy import expand_dims
```

```
from numpy import ones
from numpy import zeros
from numpy.random import rand
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [0,1]
  X = X / 255.0
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate n fake samples with class labels
def generate_fake_samples(n_samples):
  # generate uniform random numbers in [0,1]
  X = rand(28 * 28 * n_samples)
  # reshape into a batch of grayscale images
```

```
  X = X.reshape((n_samples, 28, 28, 1))
  # generate 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# train the discriminator model
def train_discriminator(model, dataset, n_iter=100, n_batch=256):
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_iter):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset, half_batch)
    # update discriminator on real samples
    _, real_acc = model.train_on_batch(X_real, y_real)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(half_batch)
    # update discriminator on fake samples
    _, fake_acc = model.train_on_batch(X_fake, y_fake)
    # summarize performance
    print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))

# define the discriminator model
model = define_discriminator()
# load image data
dataset = load_real_samples()
# fit the model
train_discriminator(model, dataset)
```

Listing 1.16: Example of defining and training the discriminator model.

Running the example first defines the model, loads the MNIST dataset, then trains the discriminator model.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the discriminator model learns to tell the difference between real and generated MNIST images very quickly, in about 50 batches.

```
...
>96 real=100% fake=100%
>97 real=100% fake=100%
>98 real=100% fake=100%
>99 real=100% fake=100%
>100 real=100% fake=100%
```

Listing 1.17: Example output from defining and training the discriminator model.

Now that we know how to define and train the discriminator model, we need to look at developing the generator model.

1.4 How to Define and Use the Generator Model

The generator model is responsible for creating new, fake but plausible images of handwritten digits. It does this by taking a point from the latent space as input and outputting a square

grayscale image. The latent space is an arbitrarily defined vector space of Gaussian-distributed values, e.g. 100 dimensions. It has no meaning, but by drawing points from this space randomly and providing them to the generator model during training, the generator model will assign meaning to the latent points. At the end of training, the latent vector space represents a compressed representation of the output space, MNIST images, that only the generator knows how to turn into plausible MNIST images.

- **Inputs**: Point in latent space, e.g. a 100 element vector of Gaussian random numbers.
- **Outputs**: Two-dimensional square grayscale image of 28×28 pixels with pixel values in [0,1].

We don't have to use a 100 element vector as input; it is a round number and widely used, but I would expect that 10, 50, or 500 would work just as well. Developing a generator model requires that we transform a vector from the latent space with, 100 dimensions to a 2D array with 28×28 or 784 values. There are a number of ways to achieve this but there is one approach that has proven effective at deep convolutional generative adversarial networks. It involves two main elements. The first is a `Dense` layer as the first hidden layer that has enough nodes to represent a low-resolution version of the output image. Specifically, an image half the size (one quarter the area) of the output image would be 14×14 or 196 nodes, and an image one quarter the size (one eighth the area) would be 7×7 or 49 nodes.

We don't just want one low-resolution version of the image; we want many parallel versions or interpretations of the input. This is a pattern in convolutional neural networks where we have many parallel filters resulting in multiple parallel activation maps, called feature maps, with different interpretations of the input. We want the same thing in reverse: many parallel versions of our output with different learned features that can be collapsed in the output layer into a final image. The model needs space to invent, create, or generate. Therefore, the first hidden layer, the `Dense` layer needs enough nodes for multiple low-resolution versions of our output image, such as 128.

```
...
# foundation for 7x7 image
model.add(Dense(128 * 7 * 7, input_dim=100))
```

Listing 1.18: Example of defining the base activations in the generator model.

The activations from these nodes can then be reshaped into something image-like to pass into a convolutional layer, such as 128 different 7×7 feature maps.

```
...
model.add(Reshape((7, 7, 128)))
```

Listing 1.19: Example of reshaping the activations into a suitable shape for a `Conv2D` layer.

The next major architectural innovation involves upsampling the low-resolution image to a higher resolution version of the image. There are two common ways to do this upsampling process, sometimes called deconvolution. One way is to use an `UpSampling2D` layer (like a reverse pooling layer) followed by a normal `Conv2D` layer. The other and perhaps more modern way is to combine these two operations into a single layer, called a `Conv2DTranspose`. We will use this latter approach for our generator.

The `Conv2DTranspose` layer can be configured with a stride of (2×2) that will quadruple the area of the input feature maps (double their width and height dimensions). It is also good practice to use a kernel size that is a factor of the stride (e.g. double) to avoid a checkerboard pattern that can be observed when upsampling (for more on upsampling layers, see Chapter **??**).

```
...
# upsample to 14x14
model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
```

Listing 1.20: Example of defining an upsampling layer.

This can be repeated to arrive at our 28×28 output image. Again, we will use the `LeakyReLU` activation with a default slope of 0.2, reported as a best practice when training GAN models. The output layer of the model is a `Conv2D` with one filter and a kernel size of 7×7 and `'same'` padding, designed to create a single feature map and preserve its dimensions at 28×28 pixels. A sigmoid activation is used to ensure output values are in the desired range of [0,1]. The `define_generator()` function below implements this and defines the generator model. The generator model is not compiled and does not specify a loss function or optimization algorithm. This is because the generator is not trained directly. We will learn more about this in the next section.

```
# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model
```

Listing 1.21: Example of a function for defining the generator model.

We can summarize the model to help better understand the input and output shapes. The complete example is listed below.

```
# example of defining the generator model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
```

```
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model

# define the size of the latent space
latent_dim = 100
# define the generator model
model = define_generator(latent_dim)
# summarize the model
model.summary()
# plot the model
plot_model(model, to_file='generator_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 1.22: Example of defining and summarizing the generator model.

Running the example summarizes the layers of the model and their output shape. We can see that, as designed, the first hidden layer has 6,272 parameters or $128 \times 7 \times 7$, the activations of which are reshaped into 128 7×7 feature maps. The feature maps are then upscaled via the two `Conv2DTranspose` layers to the desired output shape of 28×28, until the output layer, where a single activation map is output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 6272)              633472
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 6272)              0
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 14, 14, 128)       262272
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 28, 28, 128)       262272
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 28, 28, 128)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 1)         6273
=================================================================
Total params: 1,164,289
Trainable params: 1,164,289
Non-trainable params: 0
_________________________________________________________________
```

Listing 1.23: Example output from defining and summarizing the generator model.

A plot of the model is also created and we can see that the model expects a 100-element vector from the latent space as input and will generate an image as output.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

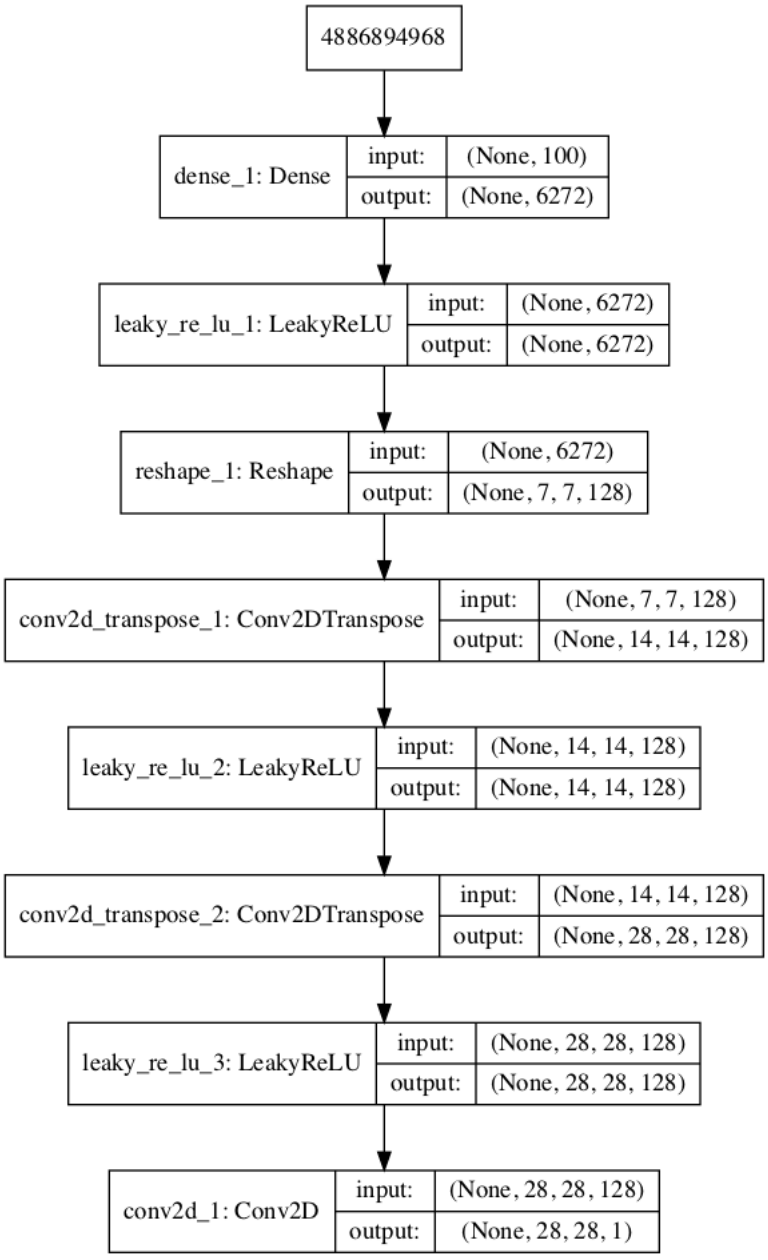

Figure 1.3: Plot of the Generator Model in the MNIST GAN.

This model cannot do much at the moment. Nevertheless, we can demonstrate how to use it to generate samples. This is a helpful demonstration to understand the generator as just another model, and some of these elements will be useful later. The first step is to draw new

points from the latent space. We can achieve this by calling the `randn()` NumPy function for generating arrays of random numbers drawn from a standard Gaussian. The array of random numbers can then be reshaped into samples, that is n rows with 100 elements per row. The `generate_latent_points()` function below implements this and generates the desired number of points in the latent space that can be used as input to the generator model.

```
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

Listing 1.24: Example of a function for generating random points in the latent space.

Next, we can use the generated points as input to the generator model to generate new samples, then plot the samples. We can update the `generate_fake_samples()` function from the previous section to take the generator model as an argument and use it to generate the desired number of samples by first calling the `generate_latent_points()` function to generate the required number of points in latent space as input to the model. The updated `generate_fake_samples()` function is listed below and returns both the generated samples and the associated class labels.

```
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y
```

Listing 1.25: Example of a function for generating synthetic images using the generator model.

We can then plot the generated samples as we did the real MNIST examples in the first section by calling the `imshow()` function with the reversed grayscale color map. The complete example of generating new MNIST images with the untrained generator model is listed below.

```
# example of defining and using the generator model
from numpy import zeros
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from matplotlib import pyplot

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
```

```
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# size of the latent space
latent_dim = 100
# define the discriminator model
model = define_generator(latent_dim)
# generate samples
n_samples = 25
X, _ = generate_fake_samples(model, latent_dim, n_samples)
# plot the generated samples
for i in range(n_samples):
  # define subplot
  pyplot.subplot(5, 5, 1 + i)
  # turn off axis labels
  pyplot.axis('off')
  # plot single image
  pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
# show the figure
pyplot.show()
```

Listing 1.26: Example of using the untrained generator to output random images.

Running the example generates 25 examples of fake MNIST images and visualizes them on a single plot of 5 by 5 images. As the model is not trained, the generated images are completely random pixel values in [0, 1].

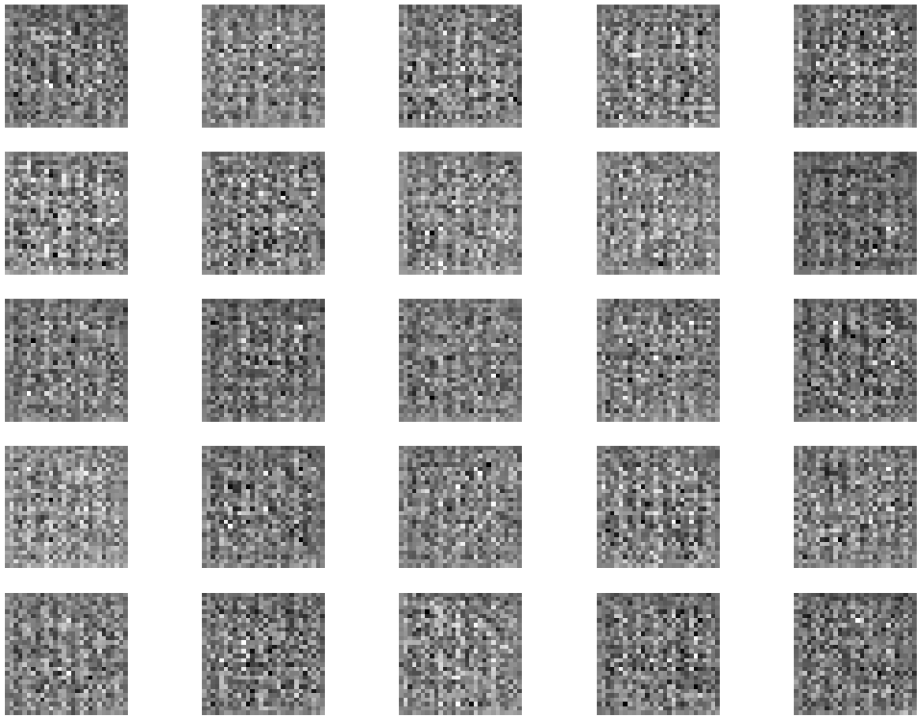

Figure 1.4: Example of 25 MNIST Images Output by the Untrained Generator Model.

Now that we know how to define and use the generator model, the next step is to train the model.

1.5 How to Train the Generator Model

The weights in the generator model are updated based on the performance of the discriminator model. When the discriminator is good at detecting fake samples, the generator is updated more, and when the discriminator model is relatively poor or confused when detecting fake samples, the generator model is updated less. This defines the zero-sum or adversarial relationship between these two models. There may be many ways to implement this using the Keras API, but perhaps the simplest approach is to create a new model that combines the generator and discriminator models. Specifically, a new GAN model can be defined that stacks the generator and discriminator such that the generator receives as input random points in the latent space and generates samples that are fed into the discriminator model directly, classified, and the output of this larger model can be used to update the model weights of the generator.

To be clear, we are not talking about a new third model, just a new logical model that uses the already-defined layers and weights from the standalone generator and discriminator models. Only the discriminator is concerned with distinguishing between real and fake examples, therefore the discriminator model can be trained in a standalone manner on examples of each, as

we did in the section on the discriminator model above. The generator model is only concerned with the discriminator's performance on fake examples. Therefore, we will mark all of the layers in the discriminator as not trainable when it is part of the GAN model so that they cannot be updated and overtrained on fake examples. When training the generator via this logical GAN model, there is one more important change. We want the discriminator to think that the samples output by the generator are real, not fake. Therefore, when the generator is trained as part of the GAN model, we will mark the generated samples as real ($class = 1$).

Why would we want to do this? We can imagine that the discriminator will then classify the generated samples as not real ($class = 0$) or a low probability of being real (0.3 or 0.5). The backpropagation process used to update the model weights will see this as a large error and will update the model weights (i.e. only the weights in the generator) to correct for this error, in turn making the generator better at generating good fake samples. Let's make this concrete.

- **Inputs**: Point in latent space, e.g. a 100 element vector of Gaussian random numbers.
- **Outputs**: Binary classification, likelihood the sample is real (or fake).

The `define_gan()` function below takes as arguments the already-defined generator and discriminator models and creates the new logical third model subsuming these two models. The weights in the discriminator are marked as not trainable, which only affects the weights as seen by the GAN model and not the standalone discriminator model. The GAN model then uses the same binary cross-entropy loss function as the discriminator and the efficient Adam version of stochastic gradient descent with the learning rate of 0.0002 and momentum 0.5, recommended when training deep convolutional GANs.

```
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

Listing 1.27: Example of a function for defining the composite model to update the generator model via the discriminator model.

Making the discriminator not trainable is a clever trick in the Keras API. The trainable property impacts the model after it is compiled. The discriminator model was compiled with trainable layers, therefore the model weights in those layers will be updated when the standalone model is updated via calls to the `train_on_batch()` function. The discriminator model was then marked as not trainable, added to the GAN model, and compiled. In this model, the model weights of the discriminator model are not trainable and cannot be changed when the GAN model is updated via calls to the `train_on_batch()` function. This change in the trainable property does not impact the training of standalone discriminator model. The complete example of creating the discriminator, generator, and composite model is listed below.

```
# demonstrate creating the three models in the gan
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
  model.add(d_model)
```

```
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# summarize gan model
gan_model.summary()
# plot gan model
plot_model(gan_model, to_file='gan_plot.png', show_shapes=True, show_layer_names=True)
```

Listing 1.28: Example of defining and summarizing the composite model.

Running the example first creates a summary of the composite model. We can see that the model expects MNIST images as input and predicts a single value as output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_2 (Sequential) (None, 28, 28, 1)       1164289
_________________________________________________________________
sequential_1 (Sequential) (None, 1)               40705
=================================================================
Total params: 1,204,994
Trainable params: 1,164,289
Non-trainable params: 40,705
_________________________________________________________________
```

Listing 1.29: Example output from defining and summarizing the composite model.

A plot of the model is also created and we can see that the model expects a 100-element point in latent space as input and will predict a single output classification label.

Note: Creating a plot of the model assumes that the `pydot` and `graphviz` libraries are installed. If this is a problem, you can comment out the import statement and the function call for `plot_model()`.

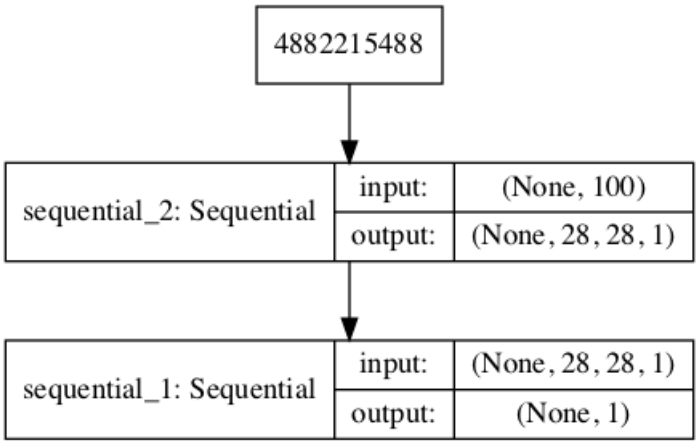

Figure 1.5: Plot of the Composite Generator and Discriminator Model in the MNIST GAN.

Training the composite model involves generating a batch worth of points in the latent space via the `generate_latent_points()` function in the previous section, and $class = 1$ labels and calling the `train_on_batch()` function. The `train_gan()` function below demonstrates this, although it is pretty simple as only the generator will be updated each epoch, leaving the discriminator with default model weights.

```
# train the composite model
def train_gan(gan_model, latent_dim, n_epochs=100, n_batch=256):
  # manually enumerate epochs
  for i in range(n_epochs):
    # prepare points in latent space as input for the generator
    x_gan = generate_latent_points(latent_dim, n_batch)
    # create inverted labels for the fake samples
    y_gan = ones((n_batch, 1))
    # update the generator via the discriminator's error
    gan_model.train_on_batch(x_gan, y_gan)
```

Listing 1.30: Example of a function for training the composite model.

Instead, what is required is that we first update the discriminator model with real and fake samples, then update the generator via the composite model. This requires combining elements from the `train_discriminator()` function defined in the discriminator section above and the `train_gan()` function defined above. It also requires that we enumerate over both epochs and batches within in an epoch. The complete train function for updating the discriminator model and the generator (via the composite model) is listed below. There are a few things to note in this model training function. First, the number of batches within an epoch is defined by how many times the batch size divides into the training dataset. We have a dataset size of 60K samples and a batch size of 256, so with rounding down, there are $\frac{60000}{256}$ or 234 batches per epoch.

The discriminator model is updated once per batch by combining one half a batch (128) of fake and real (128) examples into a single batch via the `vstack()` NumPy function. You could update the discriminator with each half batch separately (recommended for more complex datasets) but combining the samples into a single batch will be faster over a long run, especially when training on GPU hardware. Finally, we report the loss for each batch. It is critical to keep an eye on the loss over batches. The reason for this is that a crash in the discriminator

loss indicates that the generator model has started generating rubbish examples that the discriminator can easily discriminate. Monitor the discriminator loss and expect it to hover around 0.5 to 0.8 per batch on this dataset. The generator loss is less critical and may hover between 0.5 and 2 or higher on this dataset. A clever programmer might even attempt to detect the crashing loss of the discriminator, halt, and then restart the training process.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # create training set for the discriminator
      X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
      # update discriminator model weights
      d_loss, _ = d_model.train_on_batch(X, y)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, bat_per_epo, d_loss, g_loss))
```

Listing 1.31: Example of a function for training the GAN models.

We almost have everything we need to develop a GAN for the MNIST handwritten digits dataset. One remaining aspect is the evaluation of the model.

1.6 How to Evaluate GAN Model Performance

Generally, there are no objective ways to evaluate the performance of a GAN model. We cannot calculate this objective error score for generated images. It might be possible in the case of MNIST images because the images are so well constrained, but in general, it is not possible (yet). Instead, images must be subjectively evaluated for quality by a human operator. This means that we cannot know when to stop training without looking at examples of generated images. In turn, the adversarial nature of the training process means that the generator is changing after every batch, meaning that once *good enough* images can be generated, the subjective quality of the images may then begin to vary, improve, or even degrade with subsequent updates. There are three ways to handle this complex training situation.

1. Periodically evaluate the classification accuracy of the discriminator on real and fake images.

2. Periodically generate many images and save them to file for subjective review.

3. Periodically save the generator model.

All three of these actions can be performed at the same time for a given training epoch, such as every five or 10 training epochs. The result will be a saved generator model for which we have a way of subjectively assessing the quality of its output and objectively knowing how well the discriminator was fooled at the time the model was saved. Training the GAN over many epochs, such as hundreds or thousands of epochs, will result in many snapshots of the model that can be inspected and from which specific outputs and models can be cherry-picked for later use.

First, we can define a function called `summarize_performance()` that will summarize the performance of the discriminator model. It does this by retrieving a sample of real MNIST images, as well as generating the same number of fake MNIST images with the generator model, then evaluating the classification accuracy of the discriminator model on each sample and reporting these scores.

```
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
```

Listing 1.32: Example of a function for summarizing the performance of the models.

This function can be called from the `train()` function based on the current epoch number, such as every 10 epochs.

```
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
  ...
  # evaluate the model performance, sometimes
  if (i+1) % 10 == 0:
    summarize_performance(i, g_model, d_model, dataset, latent_dim)
```

Listing 1.33: Example of how model performance can be summarized from the train function.

Next, we can update the `summarize_performance()` function to both save the model and to create and save a plot generated examples. The generator model can be saved by calling the `save()` function on the generator model and providing a unique filename based on the training epoch number.

```
...
# save the generator model tile file
filename = 'generator_model_%03d.h5' % (epoch + 1)
g_model.save(filename)
```

Listing 1.34: Example of saving the generator model.

We can develop a function to create a plot of the generated samples. As we are evaluating the discriminator on 100 generated MNIST images, we can plot all 100 images as a 10 by 10 grid. The `save_plot()` function below implements this, again saving the resulting plot with a unique filename based on the epoch number.

```
# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=10):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()
```

Listing 1.35: Example of a function for saving a plot of generated images and the generator model.

The updated `summarize_performance()` function with these additions is listed below.

```
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = 'generator_model_%03d.h5' % (epoch + 1)
  g_model.save(filename)
```

Listing 1.36: Example of an updated function for summarizing the performance of the GAN.

1.7 Complete Example of GAN for MNIST

We now have everything we need to train and evaluate a GAN on the MNIST handwritten digit dataset. The complete example is listed below.

```
# example of training a gan on mnist
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import vstack
```

```
from numpy.random import randn
from numpy.random import randint
from keras.datasets.mnist import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
  model = Sequential()
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Dropout(0.4))
  model.add(Flatten())
  model.add(Dense(1, activation='sigmoid'))
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model

# define the standalone generator model
def define_generator(latent_dim):
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
  return model

# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
  # make weights in the discriminator not trainable
  d_model.trainable = False
  # connect them
  model = Sequential()
  # add generator
  model.add(g_model)
  # add the discriminator
```

```
  model.add(d_model)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model

# load and prepare mnist training images
def load_real_samples():
  # load mnist dataset
  (trainX, _), (_, _) = load_data()
  # expand to 3d, e.g. add channels dimension
  X = expand_dims(trainX, axis=-1)
  # convert from unsigned ints to floats
  X = X.astype('float32')
  # scale from [0,255] to [0,1]
  X = X / 255.0
  return X

# select real samples
def generate_real_samples(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  # generate 'real' class labels (1)
  y = ones((n_samples, 1))
  return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
  # generate points in latent space
  x_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  X = g_model.predict(x_input)
  # create 'fake' class labels (0)
  y = zeros((n_samples, 1))
  return X, y

# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=10):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
```

```
  # save plot to file
  filename = 'generated_plot_e%03d.png' % (epoch+1)
  pyplot.savefig(filename)
  pyplot.close()

# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
  # prepare real samples
  X_real, y_real = generate_real_samples(dataset, n_samples)
  # evaluate discriminator on real examples
  _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
  # prepare fake examples
  x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
  # evaluate discriminator on fake examples
  _, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
  # summarize discriminator performance
  print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
  # save plot
  save_plot(x_fake, epoch)
  # save the generator model tile file
  filename = 'generator_model_%03d.h5' % (epoch + 1)
  g_model.save(filename)

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
  bat_per_epo = int(dataset.shape[0] / n_batch)
  half_batch = int(n_batch / 2)
  # manually enumerate epochs
  for i in range(n_epochs):
    # enumerate batches over the training set
    for j in range(bat_per_epo):
      # get randomly selected 'real' samples
      X_real, y_real = generate_real_samples(dataset, half_batch)
      # generate 'fake' examples
      X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
      # create training set for the discriminator
      X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
      # update discriminator model weights
      d_loss, _ = d_model.train_on_batch(X, y)
      # prepare points in latent space as input for the generator
      X_gan = generate_latent_points(latent_dim, n_batch)
      # create inverted labels for the fake samples
      y_gan = ones((n_batch, 1))
      # update the generator via the discriminator's error
      g_loss = gan_model.train_on_batch(X_gan, y_gan)
      # summarize loss on this batch
      print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, bat_per_epo, d_loss, g_loss))
    # evaluate the model performance, sometimes
    if (i+1) % 10 == 0:
      summarize_performance(i, g_model, d_model, dataset, latent_dim)

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
```

```
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

Listing 1.37: Complete example of training a GAN to generate grayscale handwritten digits.

Note: Running the example may take many hours to run on CPU hardware. I recommend running the example on GPU hardware if possible. If you need help, you can get started quickly by using an AWS EC2 instance to train the model. See the instructions in Appendix **??**.

The chosen configuration results in the stable training of both the generative and discriminative model. The model performance is reported every batch, including the loss of both the discriminative (d) and generative (g) models.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the loss remains stable over the course of training.

```
>1, 1/234, d=0.711, g=0.678
>1, 2/234, d=0.703, g=0.698
>1, 3/234, d=0.694, g=0.717
>1, 4/234, d=0.684, g=0.740
>1, 5/234, d=0.679, g=0.757
>1, 6/234, d=0.668, g=0.777
...
>100, 230/234, d=0.690, g=0.710
>100, 231/234, d=0.692, g=0.705
>100, 232/234, d=0.698, g=0.701
>100, 233/234, d=0.697, g=0.688
>100, 234/234, d=0.693, g=0.698
```

Listing 1.38: Example output of loss from training a GAN to generate grayscale handwritten digits.

The generator is evaluated every 10 epochs, resulting in 10 evaluations, 10 plots of generated images, and 10 saved models. In this case, we can see that the accuracy fluctuates over training. When viewing the discriminator model's accuracy score in concert with generated images, we can see that the accuracy on fake examples does not correlate well with the subjective quality of images, but the accuracy for real examples may. It is crude and possibly unreliable metric of GAN performance, along with loss.

```
>Accuracy real: 51%, fake: 78%
>Accuracy real: 30%, fake: 95%
>Accuracy real: 75%, fake: 59%
>Accuracy real: 98%, fake: 11%
>Accuracy real: 27%, fake: 92%
>Accuracy real: 21%, fake: 92%
>Accuracy real: 29%, fake: 96%
>Accuracy real: 4%, fake: 99%
```

```
>Accuracy real: 18%, fake: 97%
>Accuracy real: 28%, fake: 89%
```

Listing 1.39: Example output of accuracy from training a GAN to generate grayscale handwritten digits.

More training, beyond some point, does not mean better quality generated images. In this case, the results after 10 epochs are low quality, although we can see that the generator has learned to generate centered figures in white on a black background (recall we have inverted the grayscale in the plot).

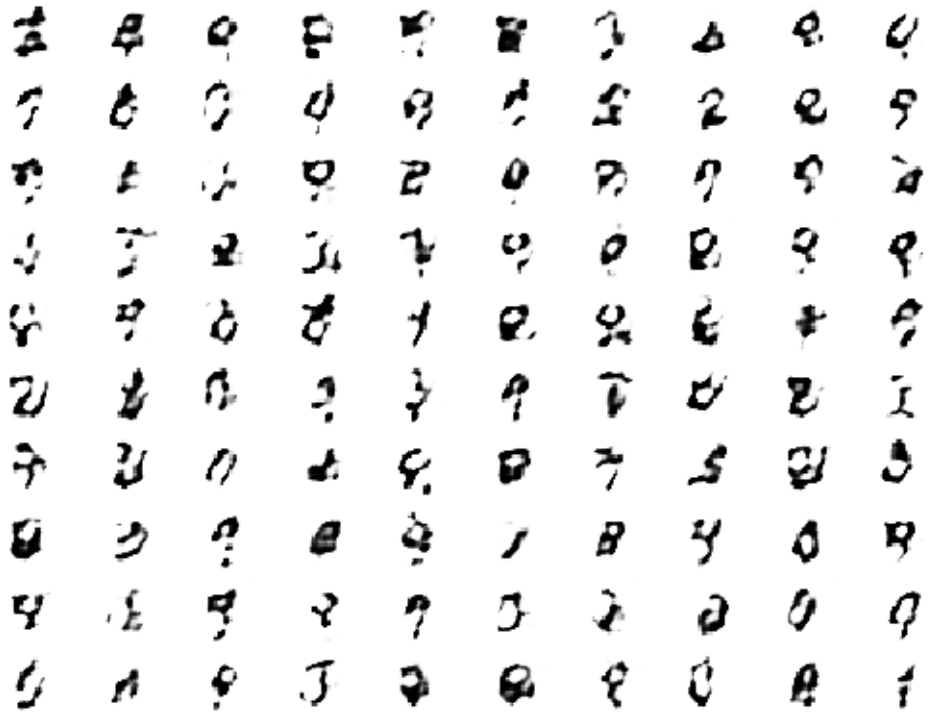

Figure 1.6: Plot of 100 GAN Generated MNIST Figures After 10 Epochs.

After 20 or 30 more epochs, the model begins to generate very plausible MNIST figures, suggesting that 100 epochs are probably not required for the chosen model configurations.

Figure 1.7: Plot of 100 GAN Generated MNIST Figures After 40 Epochs.

The generated images after 100 epochs are not greatly different, but I believe I can detect less blocky-ness in the curves.

Figure 1.8: Plot of 100 GAN Generated MNIST Figures After 100 Epochs.

1.8 How to Use the Final Generator Model

Once a final generator model is selected, it can be used in a standalone manner for your application. This involves first loading the model from file, then using it to generate images. The generation of each image requires a point in the latent space as input. The complete example of loading the saved model and generating images is listed below. In this case, we will use the model saved after 100 training epochs, but the model saved after 40 or 50 epochs would work just as well.

```
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  x_input = x_input.reshape(n_samples, latent_dim)
  return x_input
```

```
# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, n):
  # plot images
  for i in range(n * n):
    # define subplot
    pyplot.subplot(n, n, 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
  pyplot.show()

# load model
model = load_model('generator_model_100.h5')
# generate images
latent_points = generate_latent_points(100, 25)
# generate images
X = model.predict(latent_points)
# plot the result
save_plot(X, 5)
```

Listing 1.40: Complete example of loading and using the saved generator model.

Running the example first loads the model, samples 25 random points in the latent space, generates 25 images, then plots the results as a single image.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that most of the images are plausible, or plausible pieces of handwritten digits.

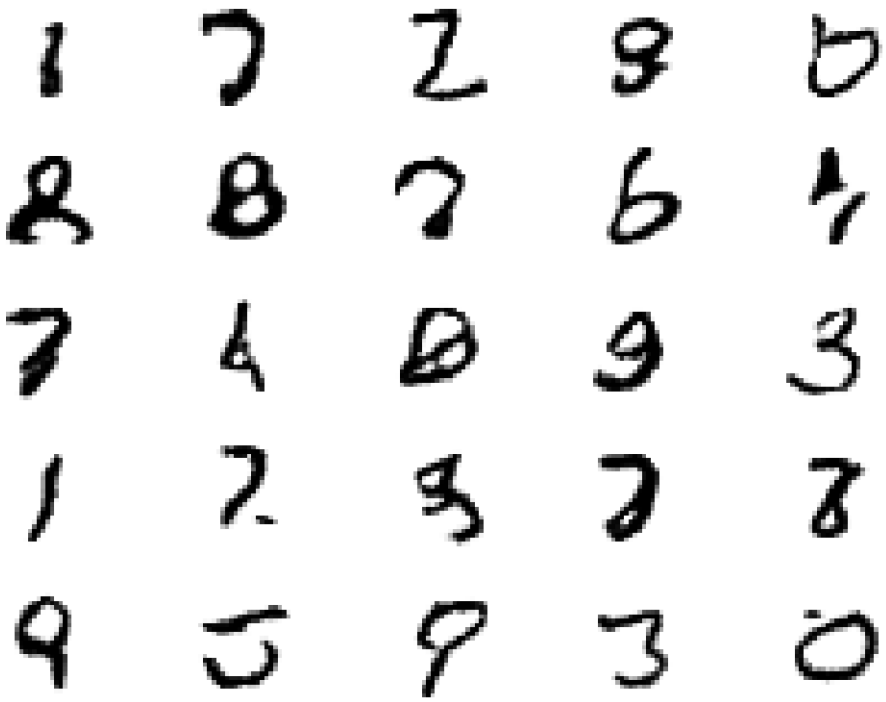

Figure 1.9: Example of 25 GAN Generated MNIST Handwritten Images.

The latent space now defines a compressed representation of MNIST handwritten digits. You can experiment with generating different points in this space and see what types of numbers they generate. The example below generates a single handwritten digit using a vector of all 0.0 values.

```
# example of generating an image for a specific point in the latent space
from keras.models import load_model
from numpy import asarray
from matplotlib import pyplot
# load model
model = load_model('generator_model_100.h5')
# all 0s
vector = asarray([[0.0 for _ in range(100)]])
# generate image
X = model.predict(vector)
# plot the result
pyplot.imshow(X[0, :, :, 0], cmap='gray_r')
pyplot.show()
```

Listing 1.41: Complete example of using the saved model to generate a single image.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, a vector of all zeros results in a handwritten 9 or maybe an 8. You can then try navigating the space and see if you can generate a range of similar, but different handwritten digits.

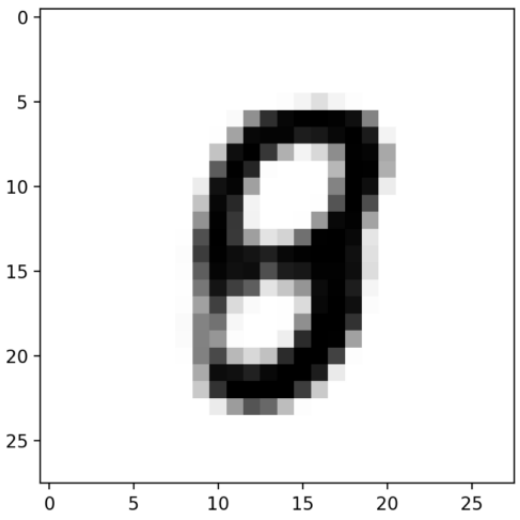

Figure 1.10: Example of a GAN Generated MNIST Handwritten Digit for a Vector of Zeros.

1.9 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **TanH Activation and Scaling**. Update the example to use the Tanh activation function in the generator and scale all pixel values to the range [-1, 1].
- **Change Latent Space**. Update the example to use a larger or smaller latent space and compare the quality of the results and speed of training.
- **Batch Normalization**. Update the discriminator and/or the generator to make use of batch normalization, recommended for DCGAN models.
- **Label Smoothing**. Update the example to use one-sided label smoothing when training the discriminator, specifically change the target label of real examples from 1.0 to 0.9, and review the effects on image quality and speed of training.
- **Model Configuration**. Update the model configuration to use deeper or more shallow discriminator and/or generator models, perhaps experiment with the `UpSampling2D` layers in the generator.

If you explore any of these extensions, I'd love to know.

1.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

1.10.1 APIs

- Keras API.
  https://keras.io/
- How can I "freeze" Keras layers?.
  https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers
- MatplotLib API.
  https://matplotlib.org/api/
- `numpy.random.rand` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html
- `numpy.random.randn` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html
- `numpy.zeros` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html
- `numpy.ones` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html
- `numpy.hstack` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html

1.10.2 Articles

- MNIST Dataset, Wikipedia.
  https://en.wikipedia.org/wiki/MNIST_database

1.11 Summary

In this tutorial, you discovered how to develop a generative adversarial network with deep convolutional networks for generating handwritten digits. Specifically, you learned:

- How to define and train the standalone discriminator model for learning the difference between real and fake images.
- How to define the standalone generator model and train the composite generator and discriminator model.
- How to evaluate the performance of the GAN and use the final standalone generator model to generate new images.

1.11.1 Next

In the next tutorial, you will develop a deep convolutional GAN for small color photos in the CIFAR-10 dataset.

This is Just a Sample

Thank-you for your interest in **Generative Adversarial Networks with Python**. This is just a sample of the full text. You can purchase the complete book online from: https://machinelearningmastery.com/generative_adversarial_networks/