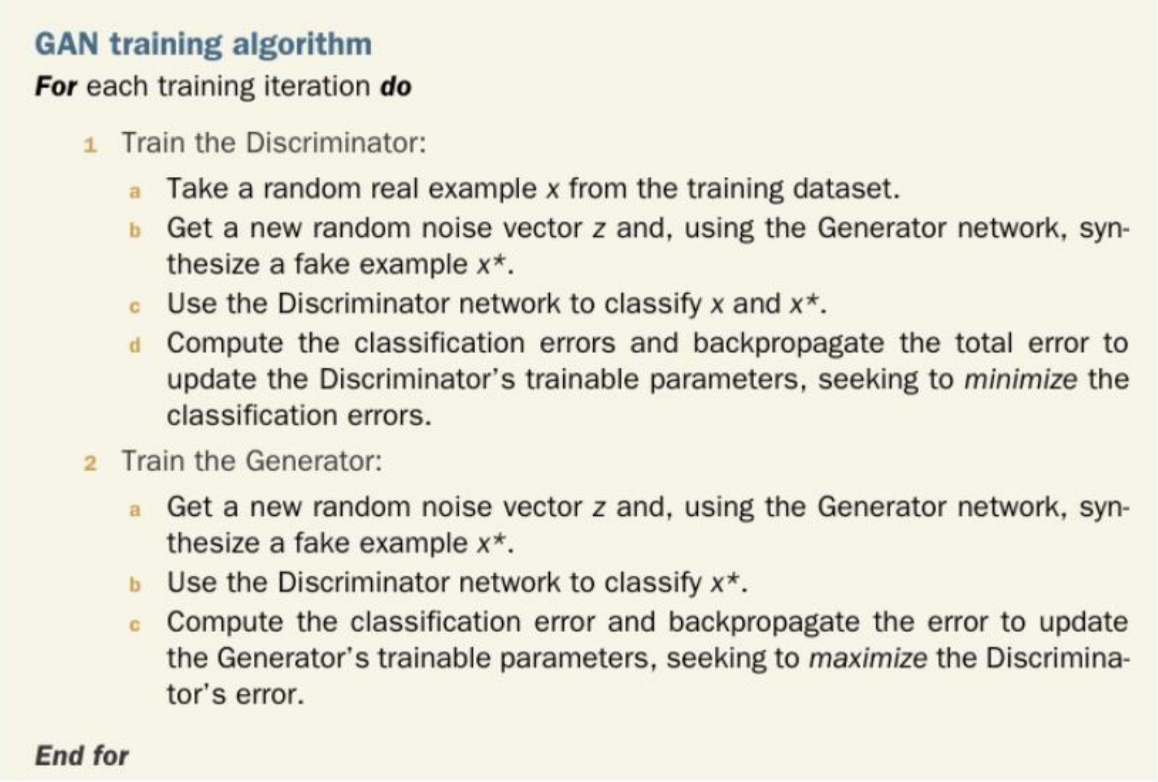
GANS



**GAN training algorithm**

```
For each training iteration do
    1 Train the Discriminator:
        a Take a random real example $x$ from the training dataset.
        b Get a new random noise vector $z$ and, using the Generator network, synthesize a fake example $x^*$.
        c Use the Discriminator network to classify $x$ and $x^*$.
        d Compute the classification errors and backpropagate the total error to update the Discriminator's trainable parameters, seeking to minimize the classification errors.
    2 Train the Generator:
        a Get a new random noise vector $z$ and, using the Generator network, synthesize a fake example $x^*$.
        b Use the Discriminator network to classify $x^*$.
        c Compute the classification error and backpropagate the error to update the Generator's trainable parameters, seeking to maximize the Discriminator's error.
End for
```

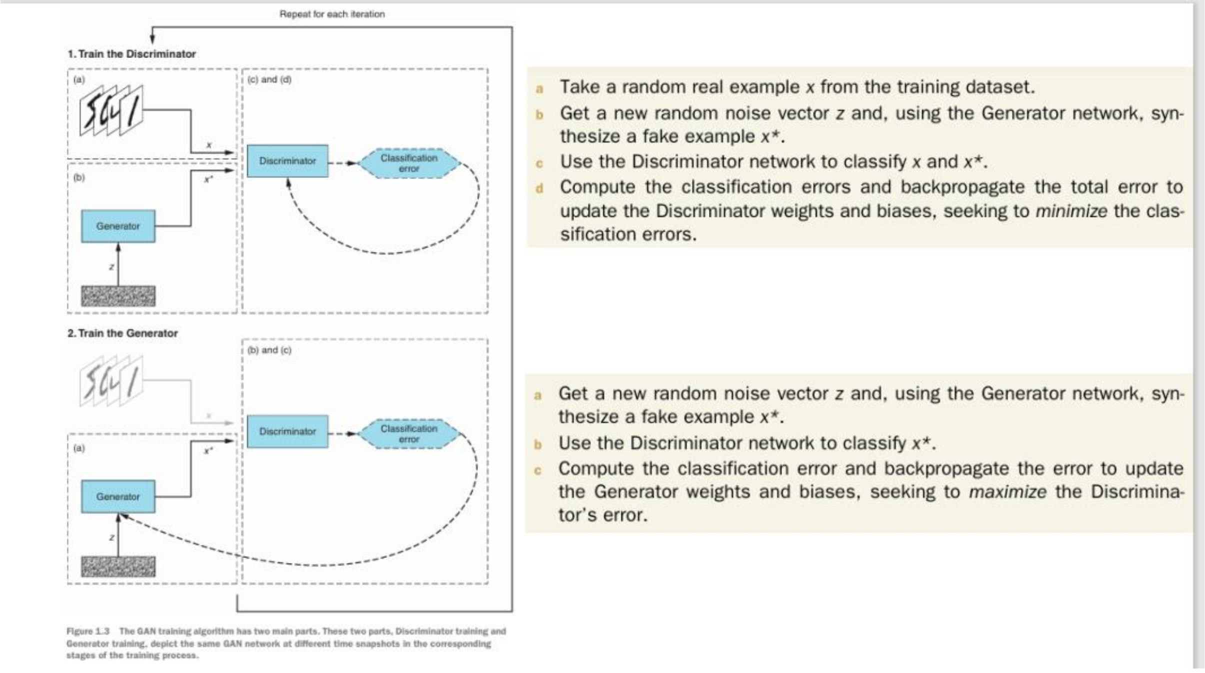


Figure 1.3 The GAN training algorithm has two main parts. These two parts, Discriminator training and Generator training, depict the same GAN network at different time snapshots in the corresponding stages of the training process.

- a Take a random real example *x* from the training dataset.
- b Get a new random noise vector *z* and, using the Generator network, synthesize a fake example $x^*$.
- c Use the Discriminator network to classify *x* and $x^*$.
- d Compute the classification errors and backpropagate the total error to update the Discriminator weights and biases, seeking to *minimize* the classification errors.

- a Get a new random noise vector *z* and, using the Generator network, synthesize a fake example $x^*$.
- b Use the Discriminator network to classify $x^*$.
- c Compute the classification error and backpropagate the error to update the Generator weights and biases, seeking to *maximize* the Discriminator's error.

gans-in-action/chapter-3/Chapter_3_GAN.ipynb at master · GANs-in-Action/gans-in-action

DCGANS

# Generating handwritten digits using DCGAN

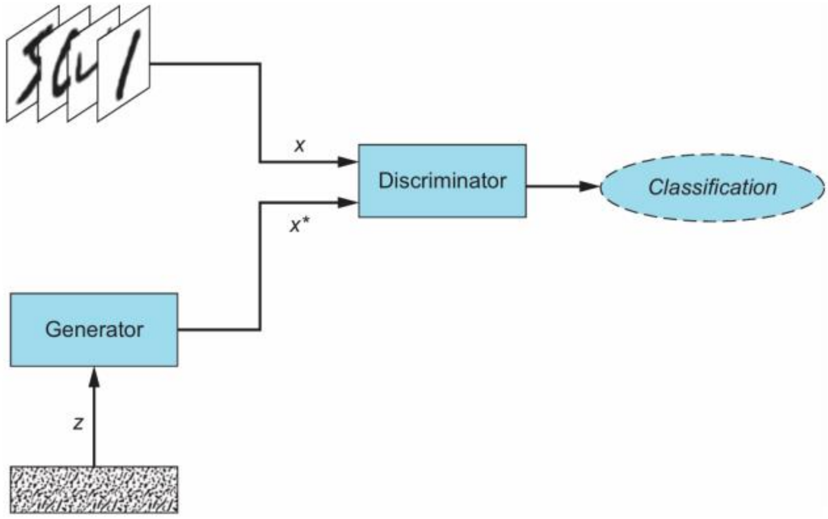


Figure 4.3 The overall model architecture for this chapter's tutorial is the same as the GAN we implemented in chapter 3. The only differences (not visible on this high-level diagram) are the internal representations of the Generator and Discriminator networks (the insides of the Generator and Discriminator boxes). These networks are covered in detail later in this tutorial.

**Listing 4.1 Import statements**

```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

from keras.datasets import mnist
from keras.layers import (
    Activation, BatchNormalization, Dense, Dropout, Flatten, Reshape)
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import Conv2D, Conv2DTranspose
from keras.models import Sequential
from keras.optimizers import Adam
```

We also specify the model input dimensions: the image shape and the length of the noise vector $z$.

**Listing 4.2 Model input dimensions**

```
img_rows = 28
img_cols = 28
channels = 1
                                                              Input image
img_shape = (img_rows, img_cols, channels)                 <- dimensions

z_dim = 100        <- Size of the noise vector, used
                      as input to the Generator
```

## Implementing the Generator

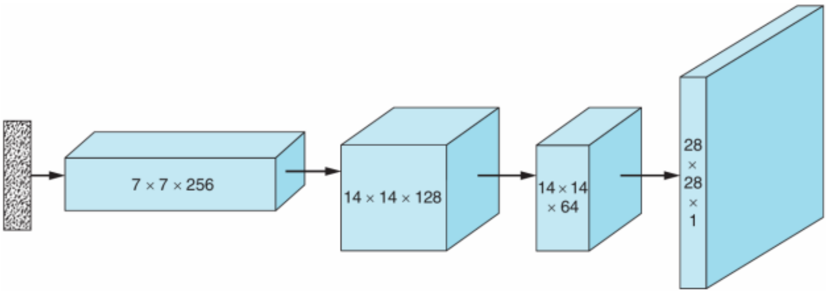
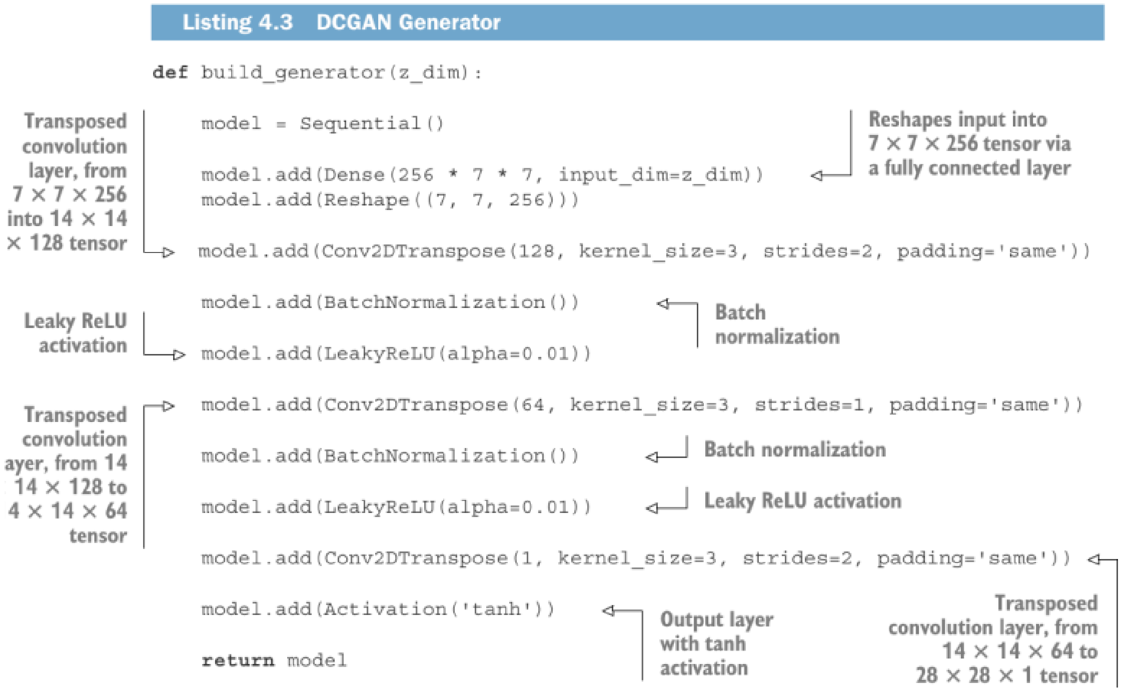


Figure 4.4 The Generator takes in a random noise vector as input and produces a 28 × 28 × 1 image. It does so by multiple layers of transposed convolutions. Between the convolutional layers, we apply batch normalization to stabilize the training process. (Image is not to scale.)

1. Take a random noise vector and reshape it into a 7 × 7 × 256 tensor through a fully connected layer.
2. Use transposed convolution, transforming the 7 × 7 × 256 tensor into a 14 × 14 × 128 tensor.
3. Apply batch normalization and the *Leaky ReLU* activation function.
4. Use transposed convolution, transforming the 14 × 14 × 128 tensor into a 14 × 14 × 64 tensor. Notice that the width and height dimensions remain unchanged; this is accomplished by setting the stride parameter in `Conv2DTranspose` to 1.
5. Apply batch normalization and the *Leaky ReLU* activation function.
6. Use transposed convolution, transforming the 14 × 14 × 64 tensor into the output image size, 28 × 28 × 1.
7. Apply the *tanh* activation function.



**Listing 4.3 DCGAN Generator**

```
def build_generator(z_dim):

    model = Sequential()

    model.add(Dense(256 * 7 * 7, input_dim=z_dim))
    model.add(Reshape((7, 7, 256)))

    model.add(Conv2DTranspose(128, kernel_size=3, strides=2, padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(alpha=0.01))

    model.add(Conv2DTranspose(64, kernel_size=3, strides=1, padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(alpha=0.01))

    model.add(Conv2DTranspose(1, kernel_size=3, strides=2, padding='same'))

    model.add(Activation('tanh'))

    return model
```

Reshapes input into 7 × 7 × 256 tensor via a fully connected layer

Transposed convolution layer, from 7 × 7 × 256 into 14 × 14 × 128 tensor

Batch normalization

Leaky ReLU activation

Transposed convolution ayer, from 14 14 × 128 to 4 × 14 × 64 tensor

Batch normalization

Leaky ReLU activation

Transposed convolution layer, from 14 × 14 × 64 to 28 × 28 × 1 tensor

Output layer with tanh activation

# Implementing the Discriminator

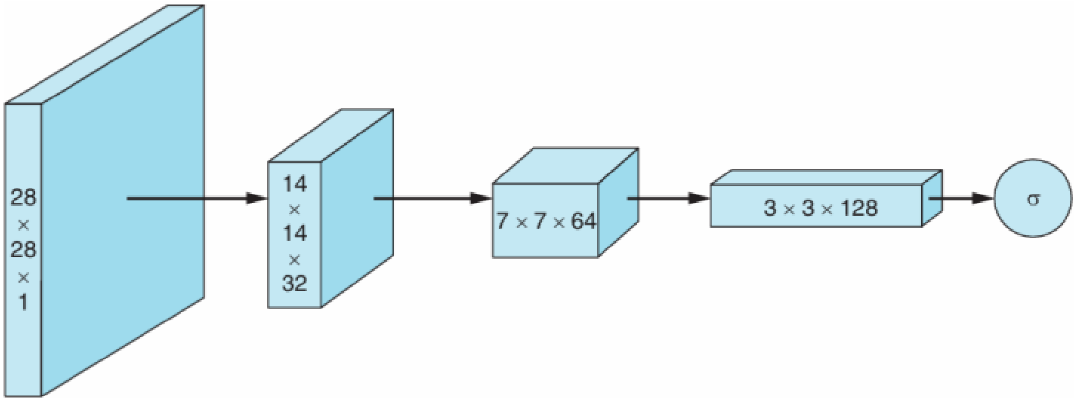


**Figure 4.5** The Discriminator takes in a 28 × 28 × 1 image as input, applies several convolutional layers, and—using the *sigmoid* activation function σ—outputs a probability that the input image is real rather than fake. Between the convolutional layers, we apply batch normalization to stabilize the training process. (Image is not to scale.)

1. Use a convolutional layer to transform a 28 × 28 × 1 input image into a 14 × 14 × 32 tensor.
2. Apply the *Leaky ReLU* activation function.
3. Use a convolutional layer, transforming the 14 × 14 × 32 tensor into a 7 × 7 × 64 tensor.
4. Apply batch normalization and the *Leaky ReLU* activation function.
5. Use a convolutional layer, transforming the 7 × 7 × 64 tensor into a 3 × 3 × 128 tensor.
6. Apply batch normalization and the *Leaky ReLU* activation function.
7. Flatten the 3 × 3 × 128 tensor into a vector of size 3 × 3 × 128 = 1152.

**Listing 4.4 DCGAN Discriminator**

```
def build_discriminator(img_shape):

    model = Sequential()

    model.add(
        Conv2D(32,
               kernel_size=3,
               strides=2,
               input_shape=img_shape,
               padding='same'))

    model.add(LeakyReLU(alpha=0.01))

    model.add(
        Conv2D(64,
               kernel_size=3,
               strides=2,
               input_shape=img_shape,
               padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(alpha=0.01))

    model.add(
        Conv2D(128,
               kernel_size=3,
               strides=2,
               input_shape=img_shape,
               padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(alpha=0.01))

    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
```

Convolutional layer, from 28 × 28 × 1 into 14 × 14 × 32 tensor

Leaky ReLU activation

Convolutional layer, from 14 × 14 × 32 into 7 × 7 × 64 tensor

Batch normalization

Leaky ReLU activation

Convolutional layer, from 7 × 7 × 64 tensor into 3 × 3 × 128 tensor

Batch normalization

Leaky ReLU activation

Output layer with sigmoid activation

## Building and running the DCGAN

**Listing 4.5 Building and compiling the DCGAN**

```
def build_gan(generator, discriminator):

    model = Sequential()

    model.add(generator)
    model.add(discriminator)

    return model


discriminator = build_discriminator(img_shape)
discriminator.compile(loss='binary_crossentropy',
                      optimizer=Adam(),
                      metrics=['accuracy'])

generator = build_generator(z_dim)

discriminator.trainable = False

gan = build_gan(generator, discriminator)
gan.compile(loss='binary_crossentropy', optimizer=Adam())
```

Combined Generator + Discriminator model

Builds and compiles the Discriminator

Builds the Generator

Keeps Discriminator's parameters constant for Generator training

Builds and compiles GAN model with fixed Discriminator to train the Generator

**Listing 4.6 DCGAN training loop**

```
losses = []
accuracies = []
iteration_checkpoints = []


def train(iterations, batch_size, sample_interval):

    (X_train, _), (_, _) = mnist.load_data()

    X_train = X_train / 127.5 - 1.0
    X_train = np.expand_dims(X_train, axis=3)

    real = np.ones((batch_size, 1))

    fake = np.zeros((batch_size, 1))

    for iteration in range(iterations):

        idx = np.random.randint(0, X_train.shape[0], batch_size)
        imgs = X_train[idx]

        z = np.random.normal(0, 1, (batch_size, 100))
        gen_imgs = generator.predict(z)

        d_loss_real = discriminator.train_on_batch(imgs, real)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
```

Loads the MNIST dataset

Rescales [0, 255] grayscale pixel values to [–1, 1]

Labels for real images: all 1s

Labels for fake images: all 0s

Gets a random batch of real images

Generates a batch of fake images

Trains the Discriminator

```
d_loss, accuracy = 0.5 * np.add(d_loss_real, d_loss_fake)

z = np.random.normal(0, 1, (batch_size, 100))
gen_imgs = generator.predict(z)
```

Generates a batch of fake images

```
g_loss = gan.train_on_batch(z, real)
```

Trains the Generator

```
if (iteration + 1) % sample_interval == 0:

    losses.append((d_loss, g_loss))
    accuracies.append(100.0 * accuracy)
    iteration_checkpoints.append(iteration + 1)
```

Saves losses and accuracies so they can be plotted after training

Outputs training progress

```
    print("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" %
          (iteration + 1, d_loss, 100.0 * accuracy, g_loss))

    sample_images(generator)
```

Outputs a sample generated image

**Listing 4.7 Displaying generated images**

```
def sample_images(generator, image_grid_rows=4, image_grid_columns=4):

    z = np.random.normal(0, 1, (image_grid_rows * image_grid_columns, z_dim))
```

Sample random noise

```
    gen_imgs = generator.predict(z)
```

Generates images from random noise

```
    gen_imgs = 0.5 * gen_imgs + 0.5
```

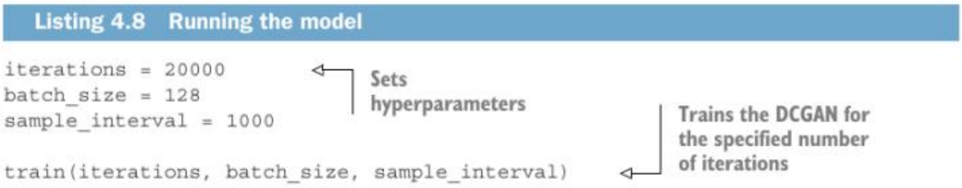
Rescales image pixel values to [0, 1]

```
    fig, axs = plt.subplots(image_grid_rows,
                            image_grid_columns,
                            figsize=(4, 4),
                            sharey=True,
                            sharex=True)
```

Sets image grid

```
    cnt = 0
    for i in range(image_grid_rows):
        for j in range(image_grid_columns):
            axs[i, j].imshow(gen_imgs[cnt, :, :, 0], cmap='gray')
            axs[i, j].axis('off')
            cnt += 1
```

Outputs a grid of images



**Listing 4.8 Running the model**

```
iterations = 20000
batch_size = 128
sample_interval = 1000

train(iterations, batch_size, sample_interval)
```

Sets hyperparameters

Trains the DCGAN for the specified number of iterations

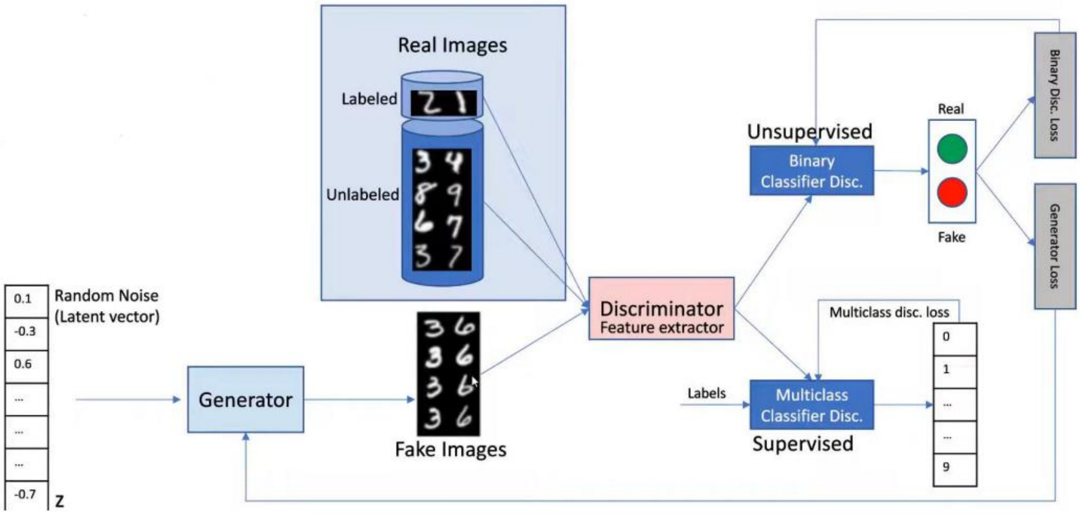
# Model Output

Figure 4.6 A sample of handwritten digits generated by a fully trained DCGAN

Figure 4.7 A sample of handwritten digits generated by the GAN implemented in chapter 3

[gans-in-action/chapter-4/Chapter_4_DCGAN.ipynb at master · GANs-in-Action/gans-in-action]()

SGANS



Implementing a SGAN
Data set:
MNIST handwritten digits
100 training examples

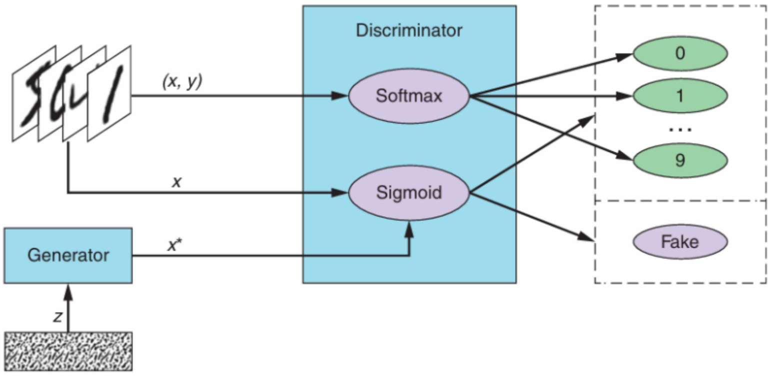
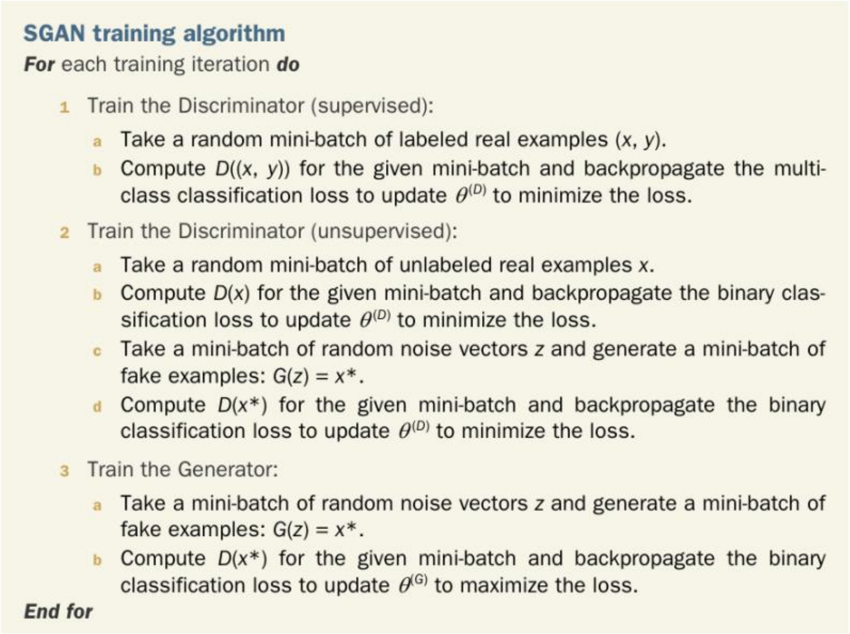


Figure 7.3 This SGAN diagram is a high-level illustration of the SGAN we implement in this chapter's tutorial. The Generator turns random noise into fake examples. The Discriminator receives real images with labels ($(x, y)$), real images without labels ($x$), and fake images produced by the Generator ($x^*$). To distinguish real examples from fake ones, the Discriminator uses the *sigmoid* function. To distinguish between the real classes, the Discriminator uses the *softmax* function.

---

Training



**SGAN training algorithm**

```
For each training iteration do
  1 Train the Discriminator (supervised):
      a Take a random mini-batch of labeled real examples $(x, y)$.
      b Compute $D((x, y))$ for the given mini-batch and backpropagate the multi-class classification loss to update $\theta^{(D)}$ to minimize the loss.
  2 Train the Discriminator (unsupervised):
      a Take a random mini-batch of unlabeled real examples $x$.
      b Compute $D(x)$ for the given mini-batch and backpropagate the binary classification loss to update $\theta^{(D)}$ to minimize the loss.
      c Take a mini-batch of random noise vectors $z$ and generate a mini-batch of fake examples: $G(z) = x^*$.
      d Compute $D(x^*)$ for the given mini-batch and backpropagate the binary classification loss to update $\theta^{(D)}$ to minimize the loss.
  3 Train the Generator:
      a Take a mini-batch of random noise vectors $z$ and generate a mini-batch of fake examples: $G(z) = x^*$.
      b Compute $D(x^*)$ for the given mini-batch and backpropagate the binary classification loss to update $\theta^{(G)}$ to maximize the loss.
End for
```

gans-in-action/chapter-7/Chapter_7_SGAN.ipynb at master · GANs-in-Action/gans-in-action

CGANS

## Architecture diagram:

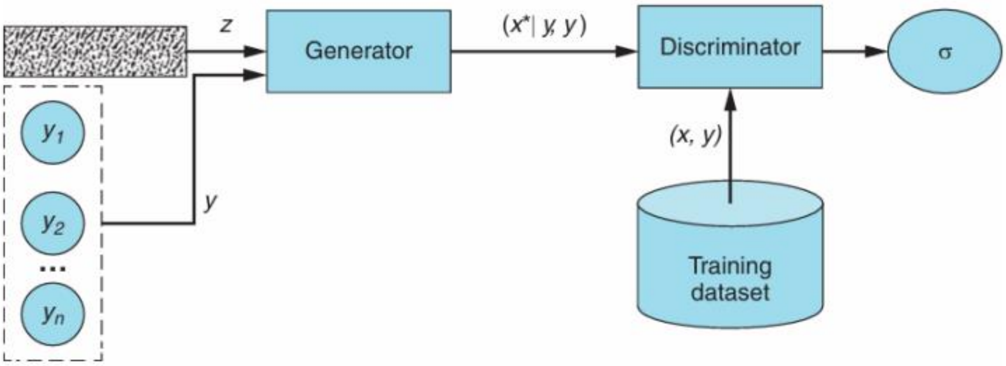


Figure 8.3 The CGAN Generator uses a random noise vector *z* and a label *y* (one of the *n* possible labels) as inputs and produces a fake example *x**|*y* that strives to be both realistic looking and a convincing match for the label *y*.

## CGAN Generator

1. Take label y (an integer from 0 to 9) and turn it into a dense vector of size z_dim (the length of the random noise vector) by using the Keras Embedding layer.
2. Combine the label embedding with the noise vector z into a joint representation by using the Keras Multiply layer. As its name suggests, this layer multiplies the corresponding entries of the two equal-length vectors and outputs a single vector of the resulting products.
3. Feed the resulting vector as input into the rest of the CGAN Generator network to synthesize an image.

The process for label “7”:

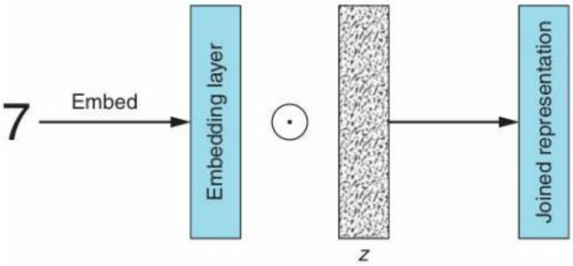


Figure 8.4 The steps used to combine the conditioning label (7 in this example) and the random noise vector *z* into a single joint representation

⊙ denotes element-wise multiplication

CGAN Discriminator

1. Take a label (an integer from 0 to 9) and—using the Keras Embedding layer— turn the label into a dense vector of size 28×28×1=784 (the length of a flattened image).
2. Reshape the label embeddings into the image dimensions (28×28×1).
3. Concatenate the reshaped label embedding onto the corresponding image, creating a joint representation with the shape (28×28×2). You can think of it as an image with its embedded label “stamped” on top of it.
4. Feed the image-label joint representation as input into the CGAN Discriminator network. Note that in order for things to work, we have to adjust the model input dimensions to (28×28×2) to reflect the new input shape.

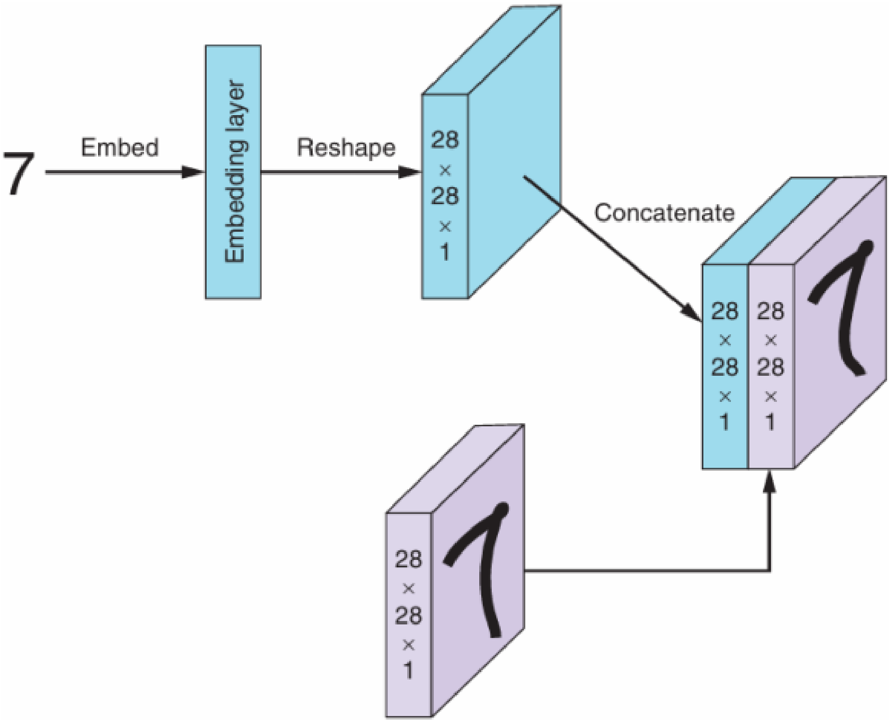
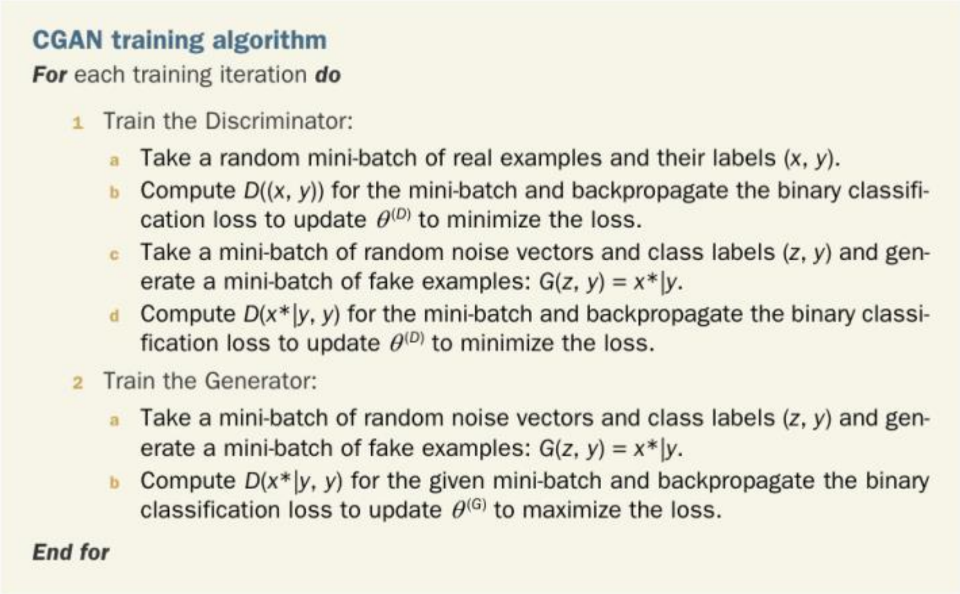


Figure 8.5 The steps used to combine the label (7 in this case) and the input image into a single joint representation

## Training



**CGAN training algorithm**

```
For each training iteration do
    1 Train the Discriminator:
        a Take a random mini-batch of real examples and their labels $(x, y)$.
        b Compute $D((x, y))$ for the mini-batch and backpropagate the binary classification loss to update $\theta^{(D)}$ to minimize the loss.
        c Take a mini-batch of random noise vectors and class labels $(z, y)$ and generate a mini-batch of fake examples: $G(z, y) = x^*|y$.
        d Compute $D(x^*|y, y)$ for the mini-batch and backpropagate the binary classification loss to update $\theta^{(D)}$ to minimize the loss.
    2 Train the Generator:
        a Take a mini-batch of random noise vectors and class labels $(z, y)$ and generate a mini-batch of fake examples: $G(z, y) = x^*|y$.
        b Compute $D(x^*|y, y)$ for the given mini-batch and backpropagate the binary classification loss to update $\theta^{(G)}$ to maximize the loss.
End for
```

gans-in-action/chapter-8/Chapter_8_CGAN.ipynb at master · GANs-in-Action/gans-in-action

CYCLEGANS

### Building the network

1. Creating the two Discriminators DA and DB and compiling them
2. Creating the two Generators:
    a. Instantiating GAB and GBA
    b. Creating placeholders for the image input for both directions
    c. Linking them both to an image in the other domain
    d. Creating placeholders for the reconstructed images back in the original domain
    e. Creating the identity loss constraint for both directions
    f. Not making the parameters of the Discriminators trainable for now
    g. Compiling the two Generators

Building the generator

1. Define the conv2d() function as follows:
    a. Standard 2D convolutional layer
    b. Leaky ReLU activation
    c. Instance normalization8
2. Define the deconv2d() function as a transposed9 convolution (aka deconvolution) layer that does the following:
    a. Upsamples the input_layer
    b. Possibly applies dropout if we set the dropout rate
    c. Always applies InstanceNormalization
    d. This creates a skip connection between its output layer and the layer of corresponding dimensionality from the downsampling part from figure 9.4

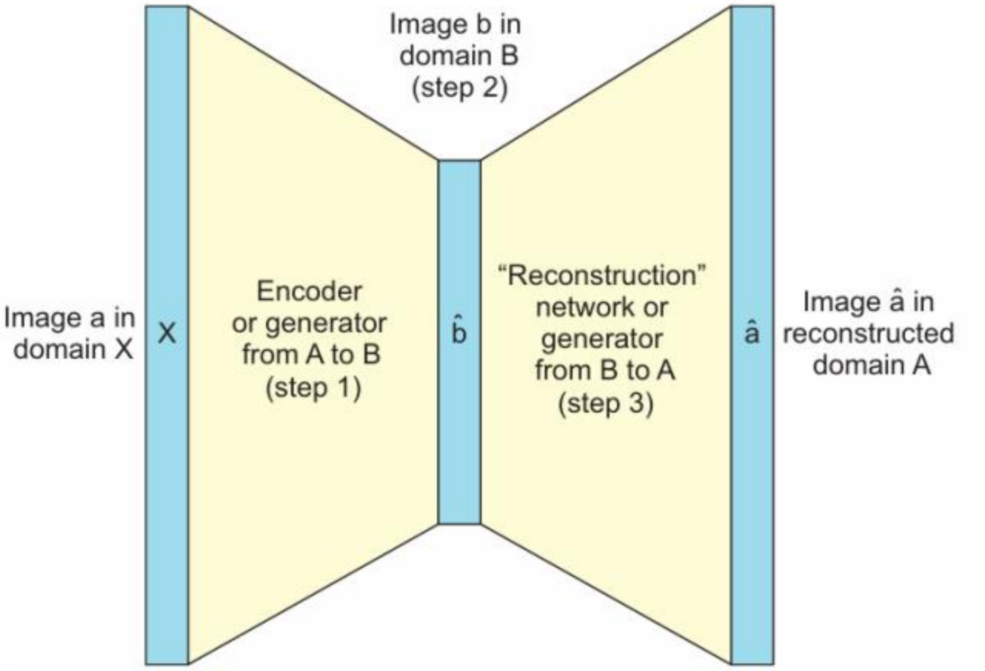


**Figure 9.4** In this image of an autoencoder from chapter 2, we used the analogy of compressing (step 1) a human concept into a more compact written form in a letter (step 2) and then expanding this concept out to the (imperfect) idea of the same notion in someone else's head (step 3).

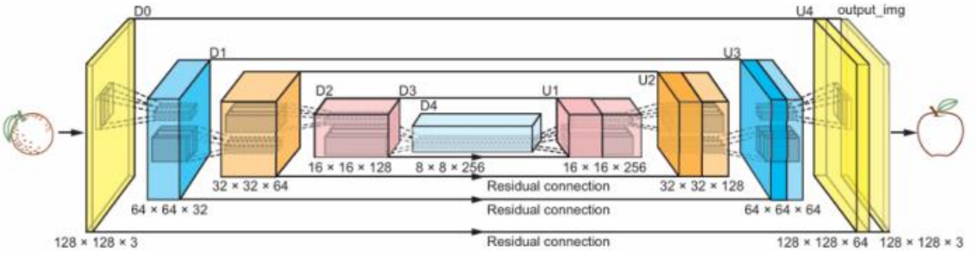


**Figure 9.6** Architecture of the Generator. The generator itself has a *contraction path* (d0 to d3) and *expanding path* (u1 to u4). The contraction and expanding paths are sometimes referred to as *encoder* and *decoder*, respectively.

Creating the actual generator:

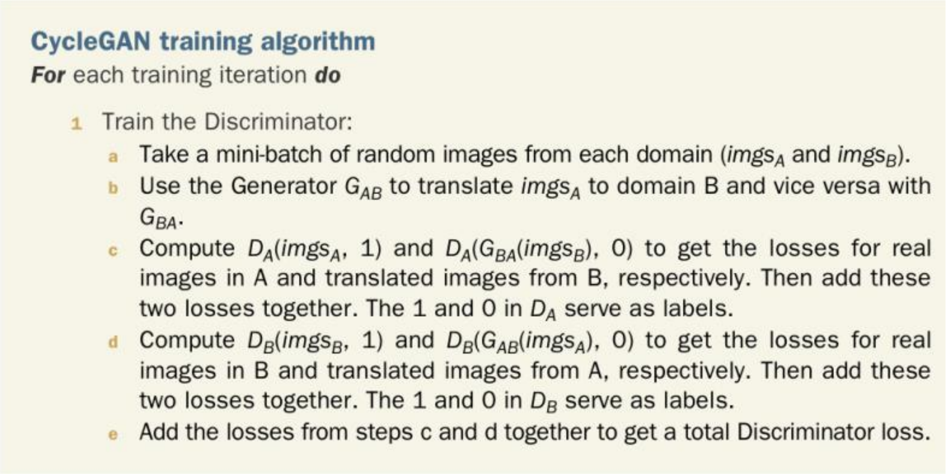
3. Take the input (128 × 128 × 3) and assign that to d0.
4. Run that through a convolutional layer d1, arriving at a 64 × 64 × 32 layer.
5. Take d1 (64 × 64 × 32) and apply conv2d to get 32 ×32 × 64 (d2).
6. Take d2 (32 × 32 × 64) and apply conv2d to get 16 × 16 × 128 (d3).
7. Take d3 (16 × 16 × 128) and apply conv2d to get 8 × 8 × 256 (d4).
8. u1: Upsample d4 and create a skip connection between d3 and u1.
9. u2: Upsample u1 and create a skip connection between d2 and u2.
10. u3: Upsample u2 and create a skip connection between d1 and u3.
11. u4: Use regular upsampling to arrive at a 128 × 128 × 64 image.
12. Use a regular 2D convolution to get rid of the extra feature maps and get only 128 × 128 × 3 (height × width × color_channels)

## Building the Discriminator

Uses a helper function that creates layers formed of 2D convolutions, *LeakyReLU*, and optionally, *InstanceNormalization*.

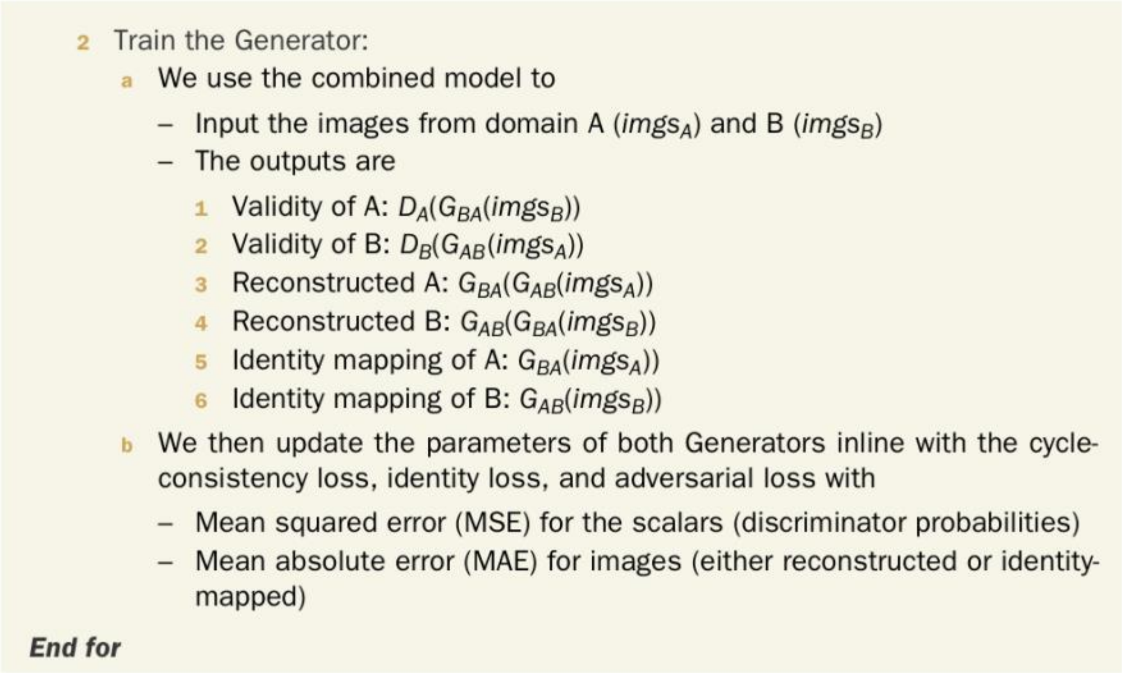
1. Take the input image (128 × 128 × 3) and assign that to d1 (64 × 64 × 64).
2. Take d1 (64 × 64 × 64) and assign that to d2 (32 × 32 × 128).
3. Take d2 (32 × 32 × 128) and assign that to d3 (16 × 16 × 256).
4. Take d3 (16 × 16 × 256) and assign that to d4 (8 × 8 × 512).
5. Take d4 (8 × 8 × 512) and flatten by conv2d to 8 × 8 × 1.

# Training the CycleGAN:



**CycleGAN training algorithm**

```
For each training iteration do
    1 Train the Discriminator:
        a Take a mini-batch of random images from each domain ($imgs_A$ and $imgs_B$).
        b Use the Generator $G_{AB}$ to translate $imgs_A$ to domain B and vice versa with $G_{BA}$.
        c Compute $D_A(imgs_A, 1)$ and $D_A(G_{BA}(imgs_B), 0)$ to get the losses for real images in A and translated images from B, respectively. Then add these two losses together. The 1 and 0 in $D_A$ serve as labels.
        d Compute $D_B(imgs_B, 1)$ and $D_B(G_{AB}(imgs_A), 0)$ to get the losses for real images in B and translated images from A, respectively. Then add these two losses together. The 1 and 0 in $D_B$ serve as labels.
        e Add the losses from steps c and d together to get a total Discriminator loss.
```



```
    2 Train the Generator:
        a We use the combined model to
            – Input the images from domain A ($imgs_A$) and B ($imgs_B$)
            – The outputs are
                1 Validity of A: $D_A(G_{BA}(imgs_B))$
                2 Validity of B: $D_B(G_{AB}(imgs_A))$
                3 Reconstructed A: $G_{BA}(G_{AB}(imgs_A))$
                4 Reconstructed B: $G_{AB}(G_{BA}(imgs_B))$
                5 Identity mapping of A: $G_{BA}(imgs_A))$
                6 Identity mapping of B: $G_{AB}(imgs_B))$
        b We then update the parameters of both Generators inline with the cycle-consistency loss, identity loss, and adversarial loss with
            – Mean squared error (MSE) for the scalars (discriminator probabilities)
            – Mean absolute error (MAE) for images (either reconstructed or identity-mapped)
End for
```

gans-in-action/chapter-9 at master · GANs-in-Action/gans-in-action