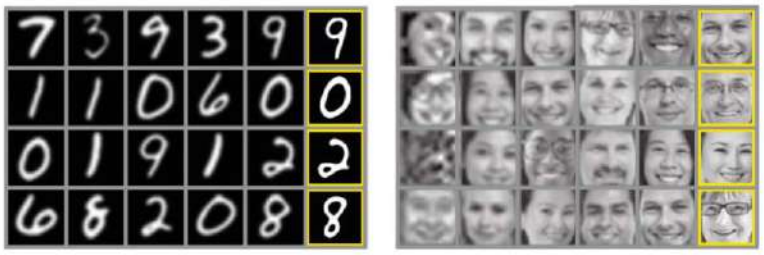
Generative adversarial networks (GANs) are a new type of neural architecture introduced by Ian Goodfellow and other researchers at the University of Montreal, including Yoshua Bengio, in 2014.

GANs have been called “the most interesting idea in the last 10 years in ML” by Yann LeCun, Facebook’s AI research director.

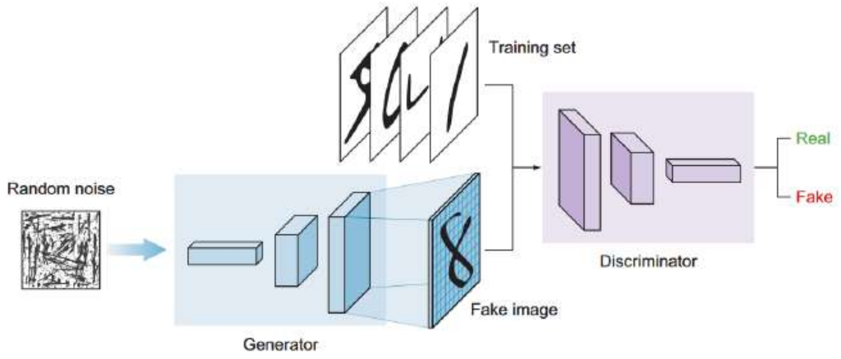
The most notable feature of GANs is their capacity to create hyper realistic images, videos, music, and text.

This Figure shows a GAN’s ability to learn features from the training images and imagine its own new images using the patterns it has learned.

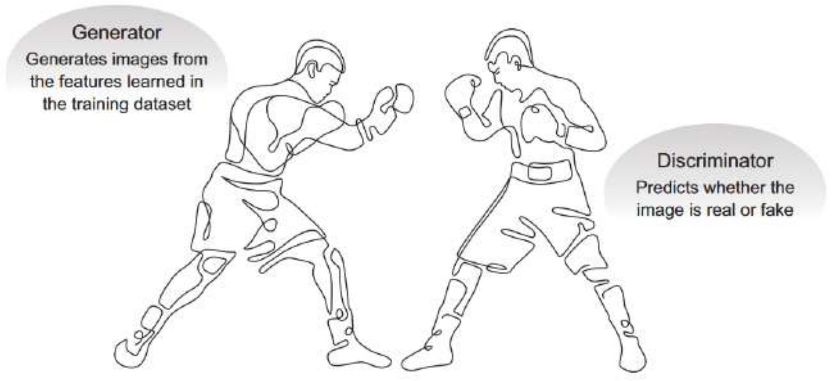


1. GAN architecture

GANs are based on the idea of *adversarial training*. The GAN architecture basically consists of two neural networks that compete against each other



We can think of the GAN architecture as two boxers fighting in their quest to win the bout, both are learning each others’ moves and techniques.
They start with less knowledge about their opponent, and as the match goes on, they learn and become better.


Generator
Generates images from the features learned in the training dataset
Discriminator
Predicts whether the image is real or fake

Deep convolutional GANs (DCGANs)

In original GAN introduced in 2014 , multi-layer perceptron (MLP) networks were used to build the generator and discriminator networks. However, since then, it has been proven that convolutional layers give greater predictive power to the discriminator, which in turn enhances the accuracy of the generator and the overall model. This type of GAN is called a deep convolutional GAN (DCGAN) and was developed by Alec Radford et al. in 2016. Now, all GAN architectures contain convolutional layers, so the "DC" is implied rest of this chapter, we refer to DCGANs as both GANs and DCGANs.

The GAN takes the following steps:

1 The generator takes in random numbers and returns an image.

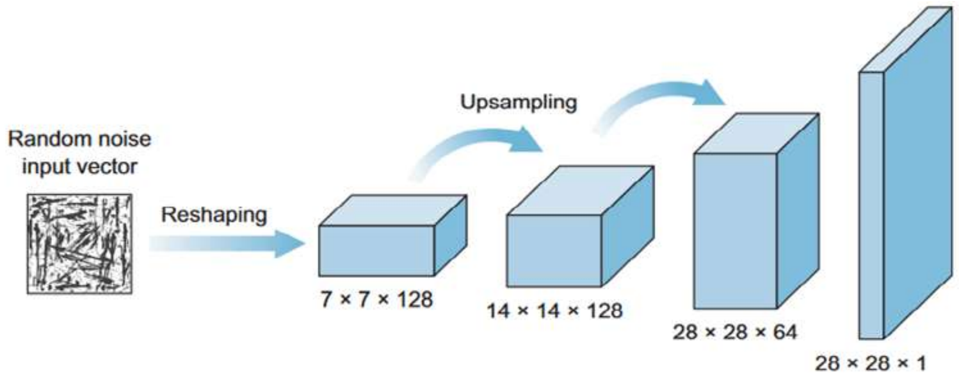
2 This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.

3 The discriminator takes in both real and fake images and returns probabilities: numbers between 0 and 1, with 1 representing a prediction of authenticity and 0 representing a prediction of fake.

The GAN architecture is composed of generator and discriminator networks.

The generator model :

- The *generator* tries to convert random noise into observations that look as if they have been sampled from the original dataset.
- The generator network, on the other hand, is an inverted CNN that starts with the flattened vector: the convolutional layers increase in size until they form the dimension of the input images.
- The generator network is also an inverted ConvNet that starts with the flattened vector.
- The images are upscaled until they are similar in size to the images in the training dataset.

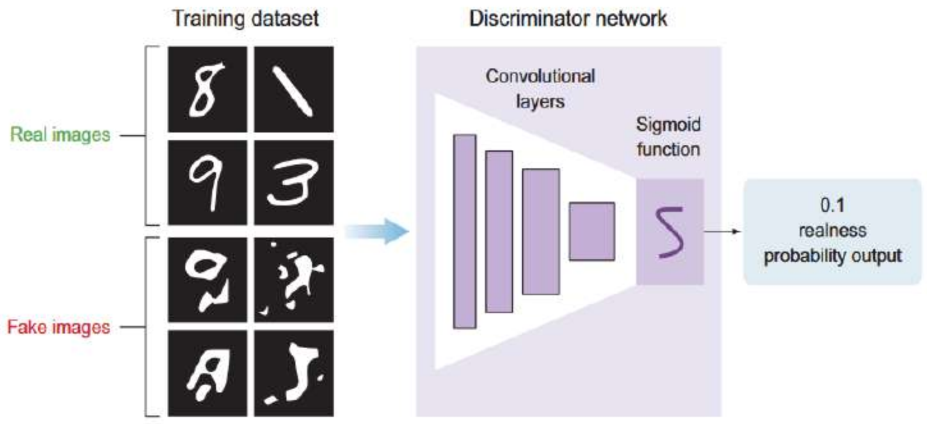


The generator takes in some random data and tries to mimic the training dataset to generate fake images. Its goal is to trick the discriminator by trying to generate images that are perfect replicas of the training dataset. As it is trained, it gets better and better after each iteration. But the discriminator is being trained at the same time, so the generator has to keep improving as the discriminator learns its tricks.

As you can see in figure above , the generator model looks like an inverted ConvNet. The generator takes a vector input with some random noise data and reshapes it into a cube volume that has a width, height, and depth. This volume is meant to be treated as a feature map that will be fed to several convolutional layers that will create the final image.

The discriminator model :

- The *discriminator* tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries.
- The discriminator network is a typical CNN where the convolutional layers reduce in size until they get to the flattened layer.
- This competitiveness helps them to mimic any distribution of data.



The goal of the discriminator is to predict whether an image is real or fake. This is a typical supervised classification problem, so we can use the traditional classifier network. We can also use the network consists of stacked convolutional layers, followed by a dense output layer with a sigmoid activation function. We use a sigmoid activation function because this is a binary classification problem: the goal of the network is to output prediction probabilities values that range between 0 and 1, where 0 means the image generated by the generator is fake and 1 means it is 100% real. The discriminator is a normal, well understood classification model. As shown in figure below, training the discriminator is pretty straightforward. We feed the discriminator labelled images: fake (or generated) and real images. The real images come from the training dataset, and the fake images are the output of the generator model.

.Example : The GAN's generator and discriminator models are like a counterfeiter and a police officer.

UPSAMPLING TO SCALE FEATURE MAPS

Traditional convolutional neural networks use pooling layers to down sample input images. In order to scale the feature maps, we use upsampling layers that scale the image dimensions by repeating each row and column of the input pixels. Keras has an upsampling layer (Upsampling2D) that scales the image dimensions by taking a scaling factor (size) as an argument: keras.layers.UpSampling2D(size=(2, 2))

This line of code repeats every row and column of the image matrix two times, because the size of the scaling factor is set to (2, 2); see figure 8.8. If the scaling factor is (3, 3), the upsampling layer repeats each row and column of the input matrix three times, as shown in figure below.

```
Input  =  1, 2
          3, 4

Output =  1, 1, 2, 2
          1, 1, 2, 2
          3, 3, 4, 4
          3, 3, 4, 4
```

Figure 8.8 Upsampling example when the scaling size is (2, 2)

```
[[1. 1. 1. 2. 2. 2.]
 [1. 1. 1. 2. 2. 2.]
 [1. 1. 1. 2. 2. 2.]
 [3. 3. 3. 4. 4. 4.]
 [3. 3. 3. 4. 4. 4.]
 [3. 3. 3. 4. 4. 4.]]
```

Figure 8.9 Upsampling example when scaling size is (3, 3)

Training the GAN

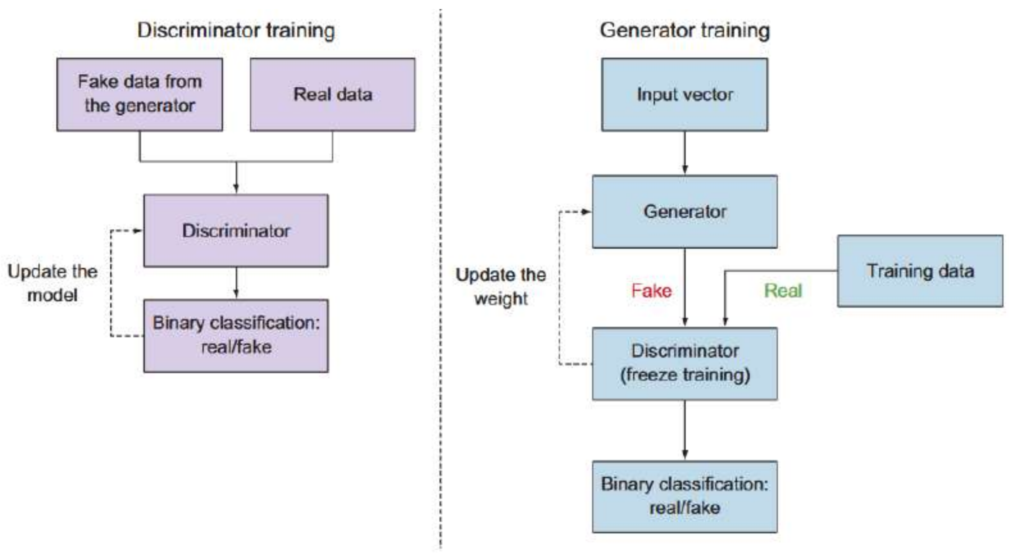
let’s put the discriminator and generator models together to train an end-to-end GAN. The discriminator is being trained to become a better classifier to maximize the probability of assigning the correct label to both training examples (real) and images generated by the generator (fake)

The process of training GAN models involves two processes:

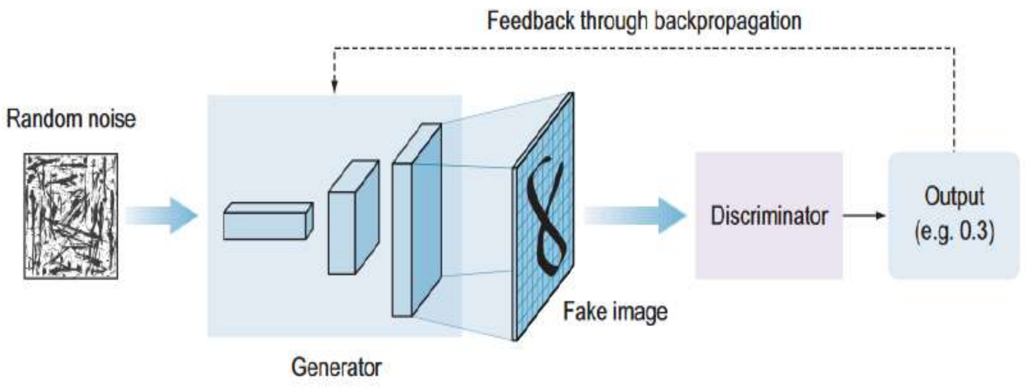
1 Train the discriminator. This is a straightforward supervised training process. The network is given labeled images coming from the generator (fake) and the training data (real), and it learns to classify between real and fake images with a sigmoid prediction output.

2 Train the generator. This process is a little tricky. The generator model cannot be trained alone like the discriminator. It needs the discriminator model to tell it whether it did a good job of faking images. So, we create a combined network to train the generator, composed of both discriminator and generator models.

Think of the training processes as two parallel lanes. One lane trains the discriminator alone, and the other lane is the combined model that trains the generator. The GAN training process is illustrated in figure below.

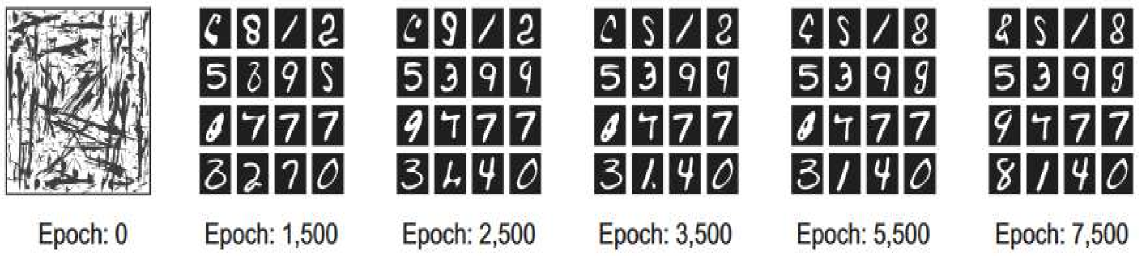


As you can see in above figure , when training the combined model, we freeze the weights of the discriminator because this model focuses only on training the generator.



TRAINING EPOCHS

For each epoch, the two compiled models (discriminator and combined) are trained simultaneously. During the training process, both the generator and discriminator improve. the performance of your GAN by printing out the results after each epoch (or a set of epochs) to see how the generator is doing at generating synthetic images. Figure below shows an example of the evolution of the generator's performance throughout its training process on the MNIST dataset.



GAN minimax function

GAN training is more of a zero-sum game than an optimization problem. In zero-sum games, the total utility score is divided among the players. An increase in one player's score results in a decrease in another player's score. In AI, this is called mini-max game theory. Minimax is a

decision-making algorithm, typically used in turn-based, two-player games. The goal of the algorithm is to find the optimal next move.
One player, called the maximizer, works to get the maximum possible score; the other player, called the minimizer, tries to get the lowest score by counter-moving against the maximizer.
GANs play a minimax game where the entire network attempts to optimize the function V(D,G) in the following equation:

$$\underset{G}{\text{Min}}\ \underset{D}{\text{Max}}\ V(D,G) = E_{x\sim p_{data}}\underbrace{[\log D(x)]}_{\substack{\textbf{Discriminator output}\\ \textbf{for real data } \boldsymbol{x}}} + E_{z\sim P_{z(z)}}[\log(1 - \underbrace{D(G(z)))]}_{\substack{\textbf{Discriminator output}\\ \textbf{for generated fake data } \boldsymbol{G(z)}}}$$

The goal of the discriminator (D) is to maximize the probability of getting the correct label of the image. The generator's (G), is to minimize the chances of getting caught. So, we train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize log(1 –D(G(z))). In other words, D and G play a two-player minimax game with the value function V(D,G).

2. Evaluating GAN models

DLNN models that are used for classification and detection problems are trained with a loss function until convergence.
A GAN generator model, is trained using a discriminator that learns to classify images as real or generated. Both the generator and discriminator models are trained together to maintain an equilibrium. No objective loss function is used to train the GAN generator models. These models are evaluated using the quality of the generated synthetic images or by manually inspecting the generated images.
Other non-manual approaches were used by Salimans et al. and by other researchers are discussed below .
In general, there is no consensus about a correct way to evaluate a given GAN generator model. This makes it challenging for researchers and practitioners to do the following:

- Select the best GAN generator model during a training run—in other words, decide when to stop training.
- Choose generated images to demonstrate the capability of a GAN generator model.
- Compare and benchmark GAN model architectures.
- Tune the model hyperparameters and configuration and compare results.

Two commonly used evaluation metrics for image quality and diversity are the inception score and the Fréchet inception distance (FID). for evaluating GAN models based on generated synthetic images.

Inception score

The inception score is based on a heuristic that realistic samples should be able to be classified when passed through a pretrained network such as Inception on ImageNet (hence the name inception score). The idea is really simple.

The heuristic relies on two values:

- *High predictability of the generated image*—We apply a pretrained inception classifier model to every generated image and get its softmax prediction. If the generated image is good enough, then it should give us a high predictability score.
- *Diverse generated samples*—No classes should dominate the distribution of the generated images.

A large number of generated images are classified using the model. Specifically, the probability of the image belonging to each class is predicted. The probabilities are then summarized in the score to capture both how much each image looks like a known class and how diverse the set of images is across the known classes. If both these traits are satisfied, there should be a large inception score. A higher inception score indicates better-quality generated images.

Fréchet inception distance (FID)

The FID score was proposed and used by Martin Heusel et al. in 2017. The score was proposed as an improvement over the existing inception score. The FID score uses the Inception model to capture specific features of an input image. The activations for each real and generated image are summarized as a multivariate Gaussian, and the distance between these two distributions is then calculated using the Fréchet distance, also called the Wasserstein-2 distance.

An important note is that the FID needs a decent sample size to give good results (the suggested size is 50,000 samples). If you use too few samples, you will end up overestimating your actual FID, and the estimates will have a large variance. A lower FID score indicates more realistic images that match the statistical properties of real images.

Which evaluation scheme to use: Both measures (inception score and FID) are easy to implement and calculate on batches of generated images. As such, the practice of systematically generating images and saving models during training can and should continue to be used to allow post hoc model selection.

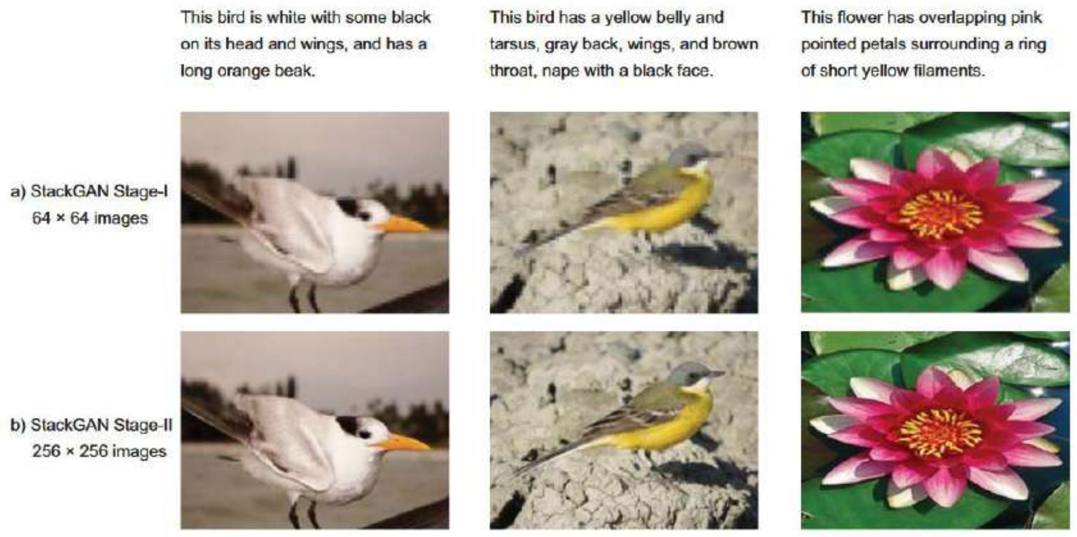
3. Popular GAN applications

Generative modeling has come a long way in the last five years. The field has developed to the point where it is expected that the next generation of generative models will be more comfortable creating art than humans. GANs now have the power to solve the problems of industries like healthcare, automotive, fine arts, and many others. In this section, we will learn about some of the use cases of adversarial networks and which GAN architecture is used for that application. The goal of this section is not to implement the variations of the GAN network, but to provide some exposure to potential applications of GAN models and resources for further reading.

Text-to-photo synthesis

Synthesis of high-quality images from text descriptions is a challenging problem in CV. Samples generated by existing text-to-image approaches can roughly reflect the meaning of the given descriptions, but they fail to contain necessary details and vivid object parts.

The GAN network that was built for this application is the stacked generative adversarial network (StackGAN).6 Zhang et al. were able to generate 256 × 256 photo realistic images conditioned on text descriptions.


This bird is white with some black on its head and wings, and has a long orange beak.
This bird has a yellow belly and tarsus, gray back, wings, and brown throat, nape with a black face.
This flower has overlapping pink pointed petals surrounding a ring of short yellow filaments.
a) StackGAN Stage-I 64 × 64 images
b) StackGAN Stage-II 256 × 256 images

a) Stage-I: Given text descriptions, StackGAN sketches rough shapes and basic colors of objects, yielding low-resolution images.

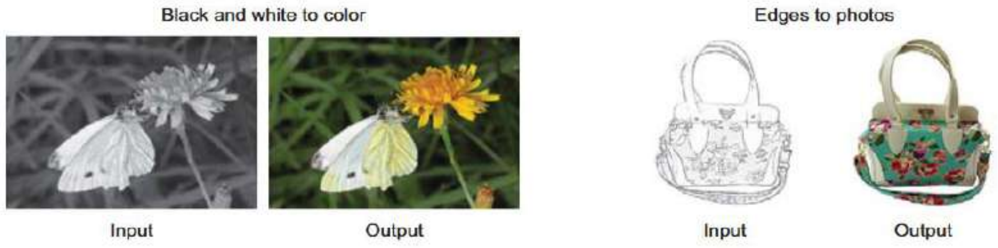
(b) Stage-II takes Stage-I results and text descriptions as inputs, and generates high-resolution images with photorealistic details. (Source: Zhang et al., 2016.)

Image-to-image translation (Pix2Pix GAN)

Image-to-image translation is defined as translating one representation of a scene into another, given sufficient training data. It is inspired by the language translation analogy: just as an idea can be expressed by many different languages, a scene may be rendered by a grayscale image, RGB image, semantic label maps, edge sketches, and so on.

In figure 8.15, image-to-image translation tasks are demonstrated on a range of applications such as converting street scene segmentation labels to real images, grayscale to color images, sketches of products to product photographs, and day photographs to night ones.

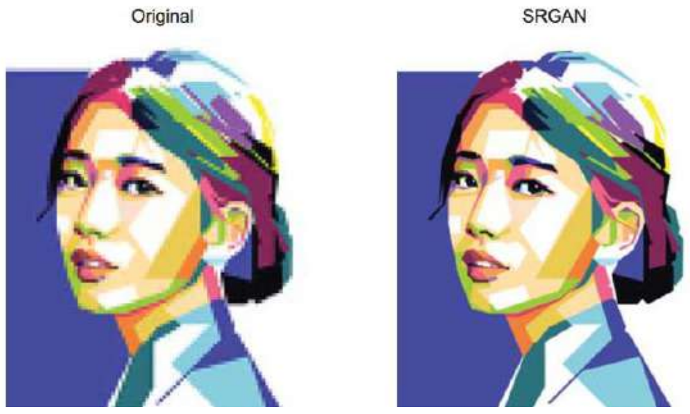
Pix2Pix is a member of the GAN family designed by Phillip Isola et al. in 2016 for general-purpose image-to-image translation. 7 The Pix2Pix network architecture is similar to the GAN concept: it consists of a generator model for outputting new synthetic images that look realistic, and a discriminator model that classifies images as real (from the dataset) or fake (generated). The training process is also similar to that used for GANs: the discriminator model is updated directly, whereas the generator model is updated via the discriminator model. As such, the two models are trained simultaneously in an adversarial process where the generator seeks to better fool the discriminator and the discriminator seeks to better identify the counterfeit images.



The novel idea of Pix2Pix networks is that they learn a loss function adapted to the task and data at hand, which makes them applicable in a wide variety of settings. They are a type of conditional GAN (cGAN) where the generation of the output image is conditional on an input source image. The discriminator is provided with both a source image and the target image and must determine whether the target is a plausible transformation of the source image.
The results of the Pix2Pix network are really promising for many image-to-image translation tasks. Visit https://affinelayer.com/pixsrv to play more with the Pix2Pix network; this site has an interactive demo created by Isola and team in which you can convert sketch edges of cats or products to photos and façades to real images.

Image super-resolution GAN (SRGAN)

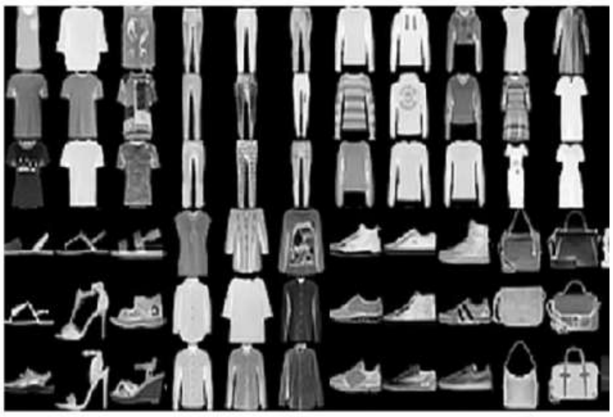
A certain type of GAN models can be used to convert low-resolution images into high-resolution images. This type is called a super-resolution generative adversarial network (SRGAN) and was introduced by Christian Ledig et al. in 2016.8 Figure 8.16 shows how SRGAN was able to create a very high-resolution image.

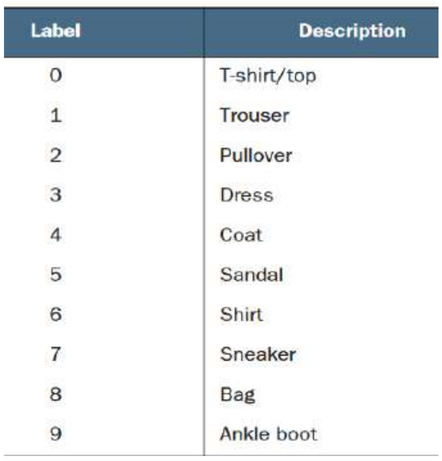


- GANs learn patterns from the training dataset and create new images that have a similar distribution of the training set.
- The GAN architecture consists of two deep neural networks that compete with each other.
- The generator tries to convert random noise into observations that look as if they have been sampled from the original dataset.
- The discriminator tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries.
- The discriminator's model is a typical classification neural network that aims to classify images generated by the generator as real or fake.
- The generator's architecture looks like an inverted CNN that starts with a narrow input and is upsampled a few times until it reaches the desired size.
- The upsampling layer scales the image dimensions by repeating each row and column of its input pixels.
- To train the GAN, we train the network in batches through two parallel networks: the discriminator and a combined network where we freeze the weights of the discriminator and update only the generator's weights.

4. Project: Building your own GAN

Building a GAN using convolutional layers in the generator and discriminator. This is called a deep convolutional GAN (DCGAN) for short. The DCGAN architecture was first explored by Alec Radford et al. (2016).In this project, the training DCGAN on the Fashion-MNIST dataset (https://github.com/zalandoresearch/fashion-mnist). Fashion-MNIST consists of 60,000 grayscale images for training and a test set of 10,000 images (figure 8.17). Each 28 × 28 grayscale image is associated with a label from 10 classes. Fashion-MNIST is intended to serve as a direct replacement for the original MNIST dataset for benchmarking machine learning algorithms. I chose grayscale images for this project because it requires less computational power to train convolutional networks on one channel grayscale images compared to three-channel colored images, which makes it easier for you to train on a personal computer without a GPU.





Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Step 1: Import libraries

As always, the first thing to do is to import all the libraries we use in this project:

```
from __future__ import print_function, division

from keras.datasets import fashion_mnist

from keras.layers import Input, Dense, Reshape, Flatten, Dropout
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D, Conv2D
from keras.models import Sequential, Model
from keras.optimizers import Adam

import numpy as np
import matplotlib.pyplot as plt
```

Imports the fashion_mnist dataset from Keras

Imports Keras layers and models

Imports numpy and matplotlib

Step 2: Download and visualize the dataset

Keras makes the Fashion-MNIST dataset available for us to download with just one command: `fashion_mnist.load_data()`. Here, we download the dataset and rescale the training set to the range –1 to 1 to allow the model to converge faster (see the "Data normalization" section in chapter 4 for more details on image scaling):

```
(training_data, _), (_, _) = fashion_mnist.load_data()

X_train = training_data / 127.5 - 1.
X_train = np.expand_dims(X_train, axis=3)
```

Loads the dataset

Rescales the training data to scale –1 to 1

Just for the fun of it, let's visualize the image matrix (figure 8.18):

```
def visualize_input(img, ax):
    ax.imshow(img, cmap='gray')
    width, height = img.shape
    thresh = img.max()/2.5
    for x in range(width):
        for y in range(height):
            ax.annotate(str(round(img[x][y],2)), xy=(y,x),
                        horizontalalignment='center',
                        verticalalignment='center',
                        color='white' if img[x][y]<thresh else 'black')

fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
visualize_input(training_data[3343], ax)
```

3. Build the Generator:

build the generator model. The input will be our noise vector (z) as explained in section 8.1.5. The generator architecture is shown in figure 8.19. The first layer is a fully connected layer that is then reshaped into a deep, narrow layer, something like 7 × 7 × 128 (in the original DCGAN paper, the team reshaped the input to 4 × 4 × 1024). Then we use the upsampling layer to double the feature map dimensions from 7 × 7 to 14 × 14 and then again to 28 × 28. In this network use three convolutional layers. We also use batch normalization and a ReLU activation. For each of these layers, the general scheme is convolution ⇒ batch normalization ⇒ ReLU. We keep stacking up layers like this until we get the final transposed convolution layer with shape 28 × 28 × 1.

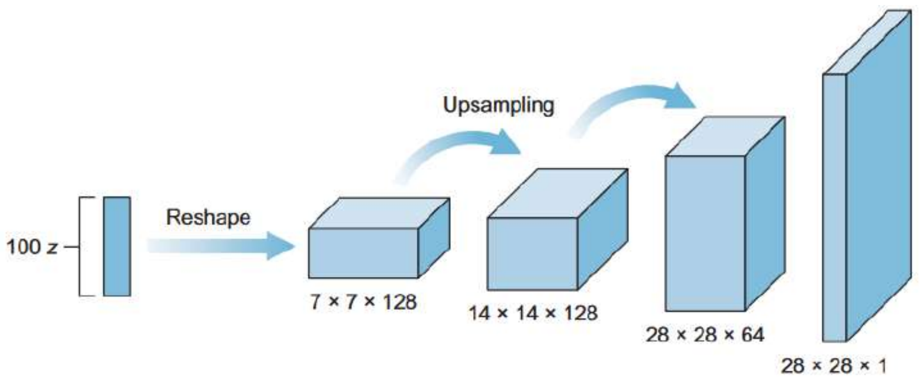


Figure 8.19 Architecture of the generator model

```
def build_generator():
    generator = Sequential()

    generator.add(Dense(128 * 7 * 7, activation="relu", input_dim=100))

    generator.add(Reshape((7, 7, 128)))

    generator.add(UpSampling2D())

    generator.add(Conv2D(128, kernel_size=3, padding="same",
                  activation="relu"))
    generator.add(BatchNormalization(momentum=0.8))
    generator.add(UpSampling2D())

# convolutional + batch normalization layers
    generator.add(Conv2D(64, kernel_size=3, padding="same",
                  activation="relu"))
    generator.add(BatchNormalization(momentum=0.8))

    # convolutional layer with filters = 1
    generator.add(Conv2D(1, kernel_size=3, padding="same",
                  activation="relu"))

    generator.summary()
```

Instantiates a sequential model and names it generator

Adds the dense layer that has a number of neurons = 128 × 7 × 7

Reshapes the image dimensions to 7 × 7 × 128

Upsampling layer to double the size of the image dimensions to 14 × 14

Upsamples the image dimensions to 28 × 28

We don't add upsampling here because the image size of 28 × 28 is equal to the image size in the MNIST dataset. You can adjust this for your own problem.

Adds a convolutional layer to run the convolutional process and batch normalization

Prints the model summary

```
noise = Input(shape=(100,))

fake_image = generator(noise)

return Model(inputs=noise, outputs=fake_image)
```

Runs the generator model to create the fake image

Returns a model that takes the noise vector as an input and outputs the fake image

Generates the input noise vector of length = 100. We chose 100 here to create a simple network.

STEP 4: BUILD THE DISCRIMINATOR

The discriminator is just a convolutional classifier like what we have built before (figure 8.20). The inputs to the discriminator are 28 × 28 × 1 images. We want a few convolutional layers and then a fully connected layer for the output. As before, we want a sigmoid output, and we need to return the logits as well. For the depths of the convolutional layers, I suggest starting with 32 or 64 filters in the first layer, and then double the depth as you add layers. In this implementation, we start with 64 layers, then 128, and then 256. For downsampling, we do not use pooling layers. Instead, we use only strided convolutional layers for downsampling, similar to Radford et al.'s implementation.

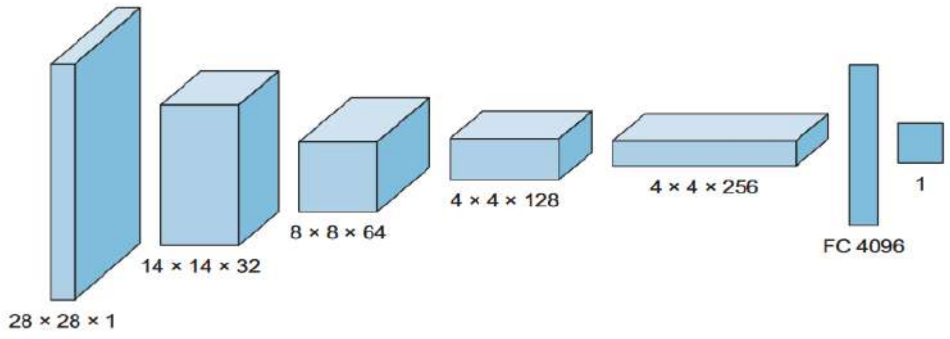


Figure 8.20 Architecture of the discriminator model

```
def build_discriminator():
    discriminator = Sequential()

    discriminator.add(Conv2D(32, kernel_size=3, strides=2,
                      input_shape=(28,28,1), padding="same"))

    discriminator.add(LeakyReLU(alpha=0.2))

    discriminator.add(Dropout(0.25))


    discriminator.add(Conv2D(64, kernel_size=3, strides=2,
                      padding="same"))

    discriminator.add(ZeroPadding2D(padding=((0,1),(0,1))))

    discriminator.add(BatchNormalization(momentum=0.8))

    discriminator.add(LeakyReLU(alpha=0.2))
    discriminator.add(Dropout(0.25))

    discriminator.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
    discriminator.add(BatchNormalization(momentum=0.8))
    discriminator.add(LeakyReLU(alpha=0.2))
    discriminator.add(Dropout(0.25))

    discriminator.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
    discriminator.add(BatchNormalization(momentum=0.8))
    discriminator.add(LeakyReLU(alpha=0.2))
    discriminator.add(Dropout(0.25))

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

    img = Input(shape=(28,28,1))
    probability = discriminator(img)

    return Model(inputs=img, outputs=probability)
```

Instantiates a sequential model and names it discriminator

Adds a convolutional layer to the discriminator model

Adds a leaky ReLU activation function

Adds a dropout layer with a 25% dropout probability

Adds a second convolutional layer with zero padding

Adds a zero-padding layer to change the dimension from 7 × 7 to 8 × 8

Adds a batch normalization layer for faster learning and higher accuracy

Adds a third convolutional layer with batch normalization, leaky ReLU, and a dropout

Flattens the network and adds the output dense layer with sigmoid activation function

Adds the fourth convolutional layer with batch normalization, leaky ReLU, and a dropout

Sets the input image shape

Returns a model that takes the image as input and produces the probability output

Runs the discriminator model to get the output probability

STEP 5: BUILD THE COMBINED MODEL

As explained in section 8.1.3, to train the generator, we need to build a combined network that contains both the generator and the discriminator (figure 8.21). The combined model takes the noise signal as input (*z*) and outputs the discriminator's prediction output as fake or real.

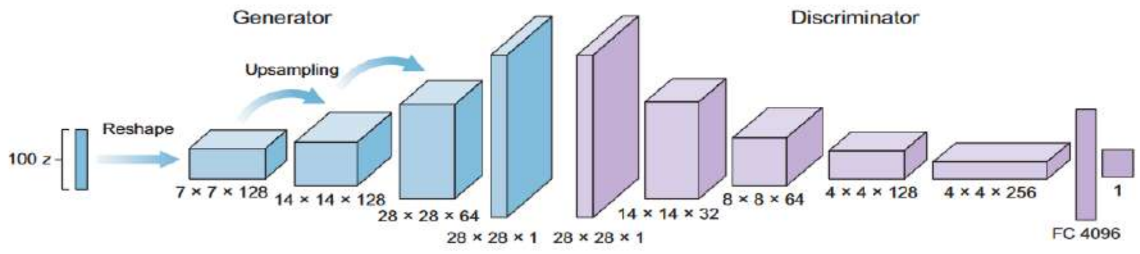


Figure 8.21 Architecture of the combined model

we created a combined network that includes both models but freeze the weights of the discriminator model in the combined network:

Defines the optimizer

```
optimizer = Adam(learning_rate=0.0002, beta_1=0.5)

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

discriminator.trainable = False
```

Builds and compiles the discriminator

Freezes the discriminator weights because we don't want to train it during generator training

Builds the generator

```
# Build the generator
generator = build_generator()

z = Input(shape=(100,))
img = generator(z)

valid = discriminator(img)

combined = Model(inputs=z, outputs=valid)
combined.compile(loss='binary_crossentropy', optimizer=optimizer)
```

The generator takes noise as input with latent_dim = 100 and generates images.

The discriminator takes generated images as input and determines their validity.

The combined model (stacked generator and discriminator) trains the generator to fool the discriminator.

S TEP 6: B UILD THE TRAINING FUNCTION

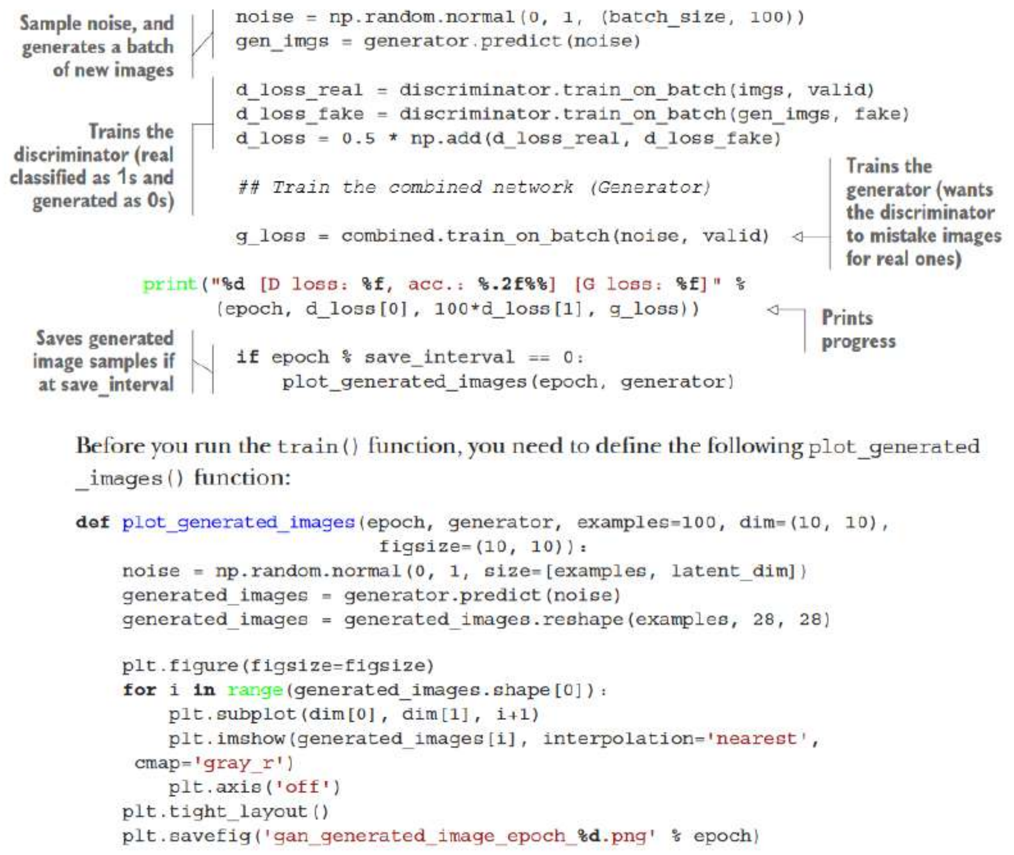
When training the GAN model, we train two networks: the discriminator and the combined network. Let's build the train function, which takes the following arguments:

- The number of epochs
- The batch size
- save_interval to state how often we want to save the results

```
def train(epochs, batch_size=128, save_interval=50):

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

    for epoch in range(epochs):

        ## Train Discriminator network

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

Adversarial ground truths

Selects a random half of images


Sample noise, and generates a batch of new images

Trains the discriminator (real classified as 1s and generated as 0s)

Trains the generator (wants the discriminator to mistake images for real ones)

Prints progress

Saves generated image samples if at save_interval

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

        d_loss_real = discriminator.train_on_batch(imgs, valid)
        d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        ## Train the combined network (Generator)

        g_loss = combined.train_on_batch(noise, valid)

    print("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" %
          (epoch, d_loss[0], 100*d_loss[1], g_loss))

        if epoch % save_interval == 0:
            plot_generated_images(epoch, generator)
```

Before you run the train() function, you need to define the following plot_generated_images() function:

```
def plot_generated_images(epoch, generator, examples=100, dim=(10, 10),
                          figsize=(10, 10)):
    noise = np.random.normal(0, 1, size=[examples, latent_dim])
    generated_images = generator.predict(noise)
    generated_images = generated_images.reshape(examples, 28, 28)

    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest',
     cmap='gray_r')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig('gan_generated_image_epoch_%d.png' % epoch)
```

STEP 7: TRAIN AND OBSERVE RESULTS

The code implementation is complete, we are ready to start the DCGAN training.

To train the model, run the following code snippet:

train(epochs=1000, batch_size=32, save_interval=50)

This will run the training for 1,000 epochs and saves images every 50 epochs. When you run the train() function, the training progress prints as shown in figure below. I ran this training myself for 10,000 epochs. Figure 8.23 shows my results after 0,50, 1,000, and 10,000 epochs. As you can see in figure 8.23, at epoch 0, the images are just random noise—no patterns or meaningful data. At epoch 50, patterns have started to form. One very apparent pattern is the bright pixels beginning to form at the center of the image, and the surroundings' darker pixels. This happens because in the training data, all of the shapes are located at the center of the image.

```
0 [D loss: 0.963556, acc.: 42.19%] [G loss: 0.726341]
1 [D loss: 0.707453, acc.: 65.62%] [G loss: 1.239887]
2 [D loss: 0.478705, acc.: 76.56%] [G loss: 1.666347]
3 [D loss: 0.721997, acc.: 60.94%] [G loss: 2.243804]
4 [D loss: 0.937356, acc.: 45.31%] [G loss: 1.459240]
5 [D loss: 0.881121, acc.: 50.00%] [G loss: 1.417385]
6 [D loss: 0.558153, acc.: 73.44%] [G loss: 1.393961]
7 [D loss: 0.404117, acc.: 78.12%] [G loss: 1.141378]
8 [D loss: 0.452483, acc.: 82.81%] [G loss: 0.802813]
9 [D loss: 0.591792, acc.: 76.56%] [G loss: 0.690274]
10 [D loss: 0.753802, acc.: 67.19%] [G loss: 0.934047]
11 [D loss: 0.957626, acc.: 50.00%] [G loss: 1.140045]
12 [D loss: 0.919308, acc.: 51.56%] [G loss: 1.311618]
13 [D loss: 0.776363, acc.: 56.25%] [G loss: 1.041264]
14 [D loss: 0.763993, acc.: 56.25%] [G loss: 1.090716]
15 [D loss: 0.754735, acc.: 56.25%] [G loss: 1.530865]
16 [D loss: 0.739731, acc.: 68.75%] [G loss: 1.887644]
```

Figure 8.22 Training progress for the first 16 epochs

epoch 1,000, you can see clear shapes and can probably guess the type of training data fed to the GAN model. Fast-forward to epoch 10,000, and you can see that the generator has become very good at re-creating new images not present in the training dataset.

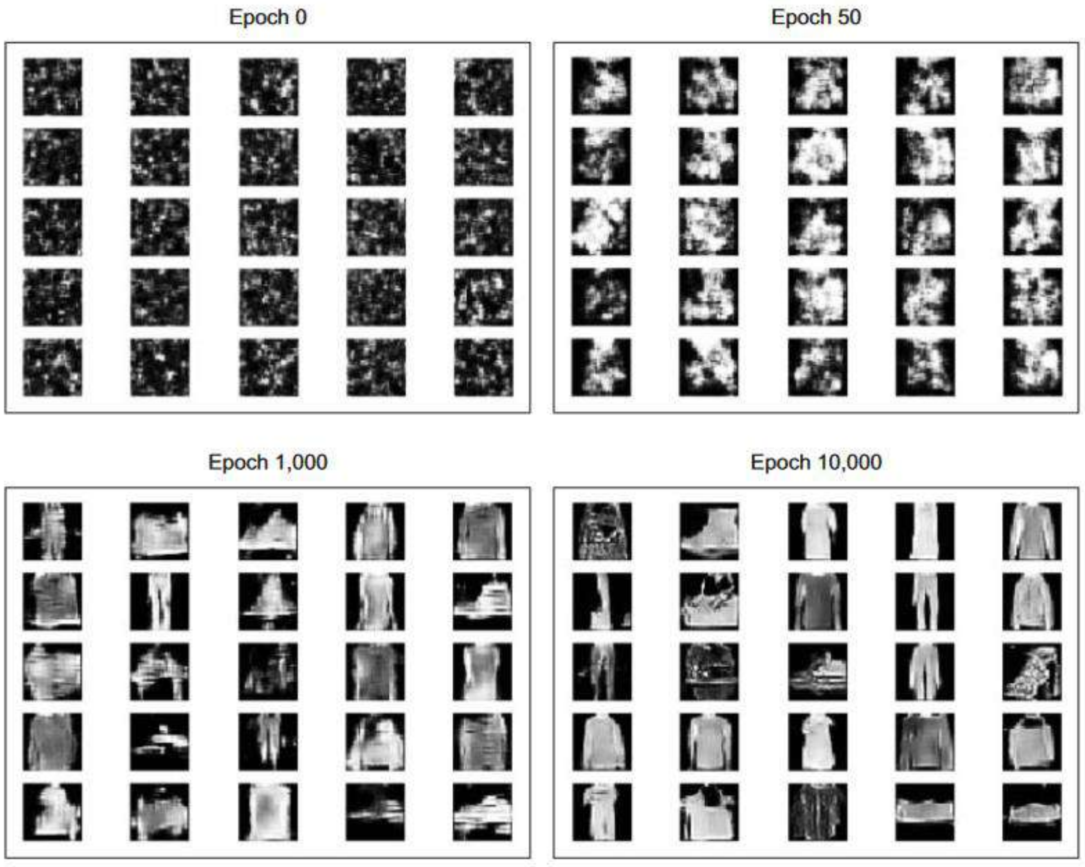


Figure 8.23 Output of the GAN generator after 0, 50, 1,000, and 10,000 epochs

For example, pick any of the objects created at this epoch: let's say the top-left image (dress). This is a totally new dress design that is not present in the training dataset. The GAN model created a completely new dress design after learning the dress patterns from the training set. You can run the training longer or make the generator network even deeper to get more refined results.

IN CLOSING For this project, we used the Fashion-MNIST dataset because the images are very small and are in grayscale (one-channel), which makes it computationally inexpensive for you to train on your local computer with no GPU. Fashion-MNIST is also very clean data: all of the images are centered and have less noise so they don’t require much preprocessing before you kick off your GAN training. you can try CIFAR as your next step (https://www.cs.toronto.edu/~kriz/cifar.html) or Google’s Quick, Draw! dataset (https://quickdraw.withgoogle.com), which is considered the world’s largest doodle dataset at the time of writing. Another, more serious, dataset is Stanford’s Cars Dataset (https://ai.stanford.edu/~jkrause/cars/car_dataset.html), which contains more than 16,000 images of 196 classes of cars. You can try to train your GAN model to design a completely new design for your dream car!

Generative adversarial networks (GANs) have been called as "the most interesting idea in the last 10 years in ML" by Yann LeCun, Facebook's AI research director.

The most notable feature of GANs is their capacity to create hyper realistic images, videos, music, and text.

The GAN architecture is composed of generator and discriminator networks.

- The *generator* tries to convert random noise into observations that look as if they have been sampled from the original dataset.
- The *discriminator* tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries. The goal of the discriminator is to predict whether an image is real or fake.

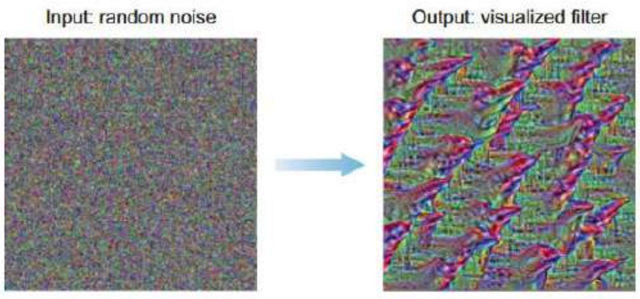
In fine arts, especially painting, humans have mastered the skill of creating unique visual experiences through composing a complex interplay between the content and style of an image.

The two new techniques to create artistic images using neural networks are **DeepDream** and **neural style transfer**.

The CNNs are used to extract features in object classification and detection problems; we learn how to visualize the extracted feature maps. One reason is that we need this visualization technique in order to understand the DeepDream algorithm.

how do CNNs see the world? How do they see the extracted features between all the layers?

In computer vision problems, we can visualize the feature maps inside the convolutional network to understand how they see the world and what features they think are distinctive in an object for differentiating between classes. The idea of visualizing convolutional layers was proposed by Erhan et al. in 2009.



From above figure Start with an image consisting of random noise, and tweak it until we visualize what the network considers important features of a bird.

from the bird example and see how to visualize the network filters. The takeaway from this introduction is that neural networks are smart enough to understand which are the important features to pass along through its layers to be classified by its fully connected layers. Non-important features are discarded along the way. To put it simply, neural networks learn the features of the objects in the training dataset. If we are able to visualize these feature maps at the deeper layers of the network, we can find out where the neural network is paying attention and see the exact features that it uses to make its predictions.

From the VGG16 diagram in figure below, visualizing the output feature maps of the first, middle, and deep layers as follows: block1_conv1, block3_conv2, and block5_conv3. Figures below show how the features evolve throughout the network layers.

From the VGG16 diagram in figure below, visualizing the output feature maps of the first, middle, and deep layers as follows: block1_conv1, block3_conv2, and block5_conv3. Figures below show how the features evolve throughout the network layers.

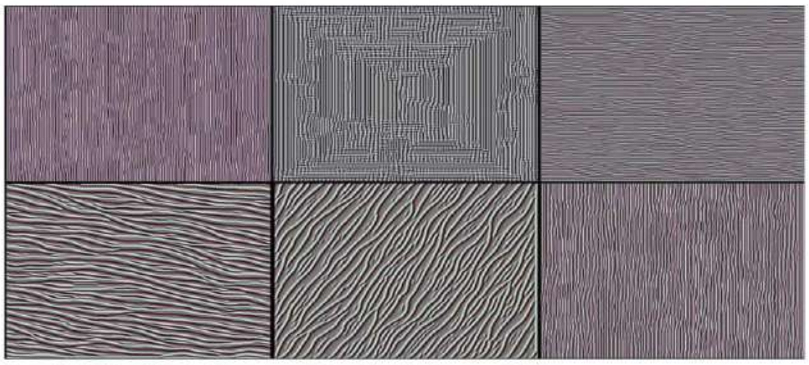


Figure 9.2 Visualizing feature maps produced by block1_conv1 filters

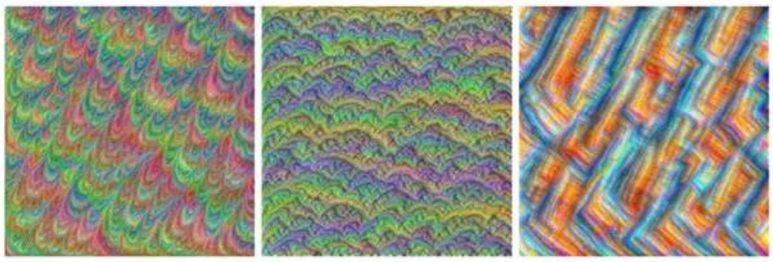


Figure 9.3 Visualizing feature maps produced by block3_conv2 filters

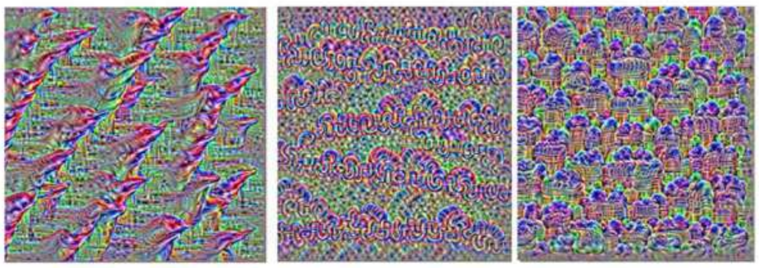


Figure 9.4 Visualizing feature maps produced by block5_conv3 filters

How is this helpful in classification and detection problems? The left feature map in figure above as an example. Looking at the visible features like eyes and beaks, I can interpret that the network relies on these two features to identify a bird. With this knowledge about what the network learned about birds, I will guess that it can detect the bird in figure below, because the bird’s eye and beak are visible.

Now, let’s consider a more adversarial case where we can see the bird’s body but the eye and beak are covered by leaves (figure above). Given that the network adds high weights on the eye and beak features to recognize a bird, there is a good chance that it might miss this bird because the bird’s main features are hidden. On the other hand, an average human can easily detect the bird in the image. The solution to this problem is using one of several data-augmentation techniques and collecting more adversarial cases in your training dataset to force the network to add higher weights on other features of a bird, like shape and color.

DeepDream

DeepDream was developed by Google researchers Alexander Mordvintsev et al. in 2015. It is an artistic image modification technique that ***creates dream-like***, ***hallucinogenic images*** using CNNs, as shown figure below.

Here the original input image is a scenic image from the ocean, containing two dolphins and other creatures. DeepDream merged both dolphins into one object and replaced one of the faces with what looks like a dog face. Other objects were also deformed in an artistic way, and the sea back-ground has an edge-like texture.

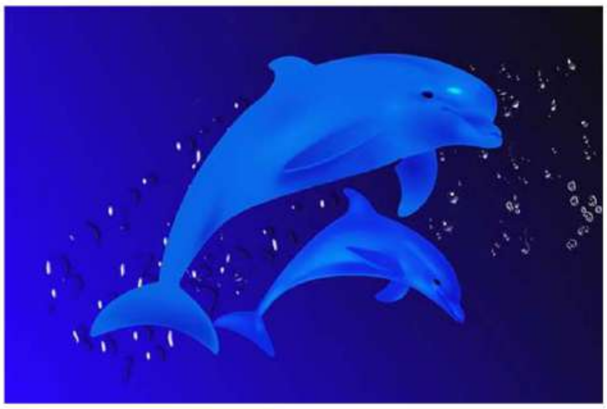


Figure 9.10 DeepDream input image

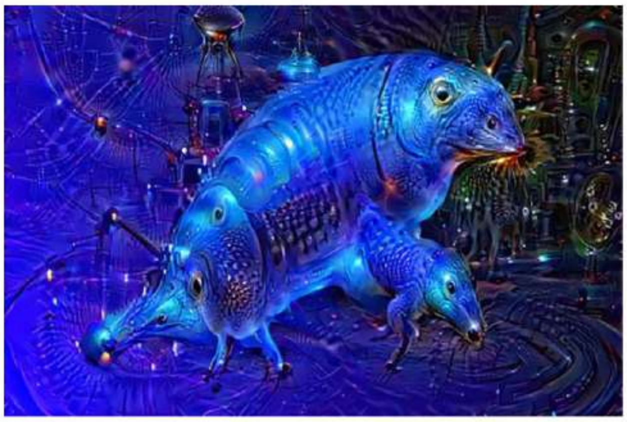


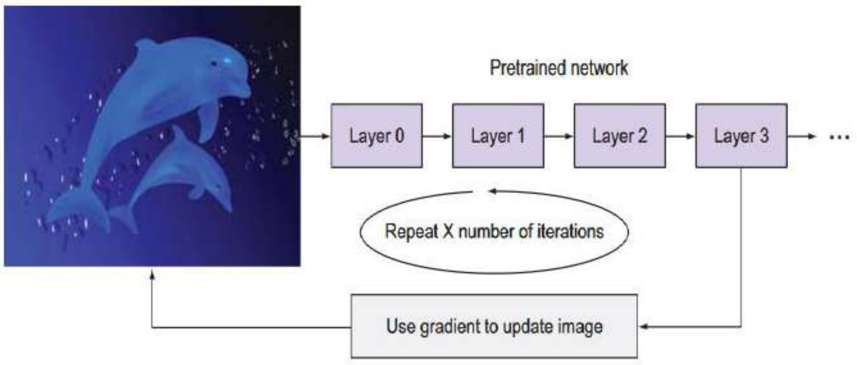
Figure 9.9 DeepDream output image

DeepDream quickly became an internet sensation, thanks to the trippy pictures it generates, full of algorithmic artifacts, bird feathers, dog faces, and eyes. These artifacts are byproducts of the fact that the DeepDream. ConvNet was trained on ImageNet, where dog breeds and bird species are vastly overrepresented. A CNN in reverse and visualize its activation maps using the same convolutional filter-visualization techniques. ConvNet in reverse, doing gradient ascent on the input in order to maximize the activation of a specific filter in an upper layer of the ConvNet.

DeepDream uses this same idea, with a few alterations:

 Input image —In ***filter visualization***, we don't use an input image. We start from a blank image (or a slightly noisy one) and then maximize the filter activations of the convolutional layers to view their features.
In ***DeepDream***, we use an input image to the network because the goal is to print these visualized features on an image.

 Maximizing filters versus layers—***In filter visualization***, as the name implies, we only maximize activations of specific filters within the layer.
But in ***DeepDream***, we aim to maximize the activation of the entire layer to mix together a large number of features at once.
 Octaves—In ***DeepDream***, the input images are processed at different scales called octaves to improve the quality of the visualized features. This process will be explained next.



The steps in DeepDream algorithm.

1 Load the input image.
2 Define the number of scales, from smallest to largest.
3 Resize the input image to the smallest scale.
4 For each scale, start with the smallest and apply the following:
– Gradient ascent function
– Upscaling to the next scale
– Re-injecting details that were lost during the upscale process
5 Stop the process when we are back to the original size.

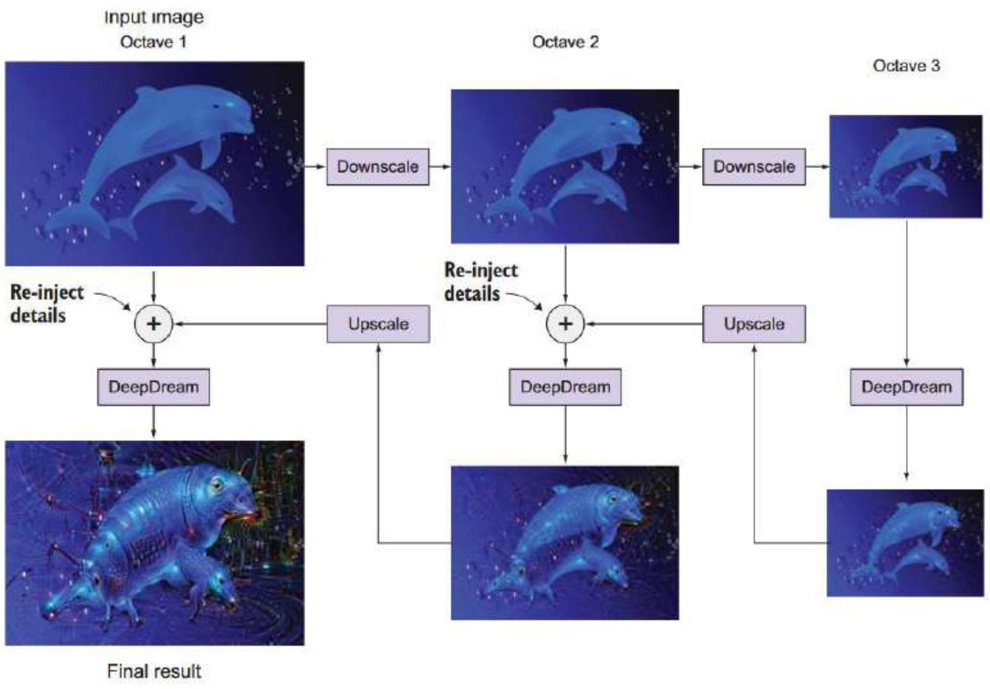


Figure 9.12 The DeepDream process: successive image downscales called octaves, detail re-injection, and then upscaling to the next octave

Neural style transfer

we have learned how to visualize specific filters in a network. We also learned how to manipulate features of an input image to create dream-like hallucinogenic images using the DeepDream algorithm.

A new type of artistic image that ConvNets can create using neural style transfer the technique of transferring the style from one image to another. The goal of the neural style transfer algorithm is to take the style of an image (style image) and apply it to the content of another image (content image). Style in this context means texture, colors, and other visual patterns in the image. And content is the higher-level macrostructure of the image. The result is a combined image that contains both the content of the content image and the style of the style image.
For example, let's look at figure 9.16. The objects in the content image (like dolphins, fish, and plants) are kept in the combined image but with the specific texture of the style image (blue and yellow brushstrokes).

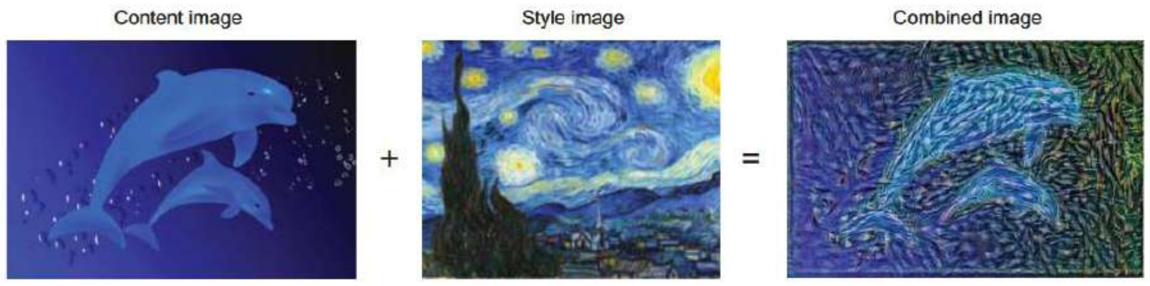


Figure 9.16 Example of neural style transfer

The idea of neural style transfer was introduced by Leon A. Gatys et al. in 2015. The concept of style transfer, which is tightly related to texture generation, had a long history in the image-processing community prior to that; but as it turns out, the DL-based implementations of style transfer offer results unparalleled by what had been previously achieved with traditional CV techniques, and they triggered an amazing renaissance in creative CV applications.

The main idea behind implementing style transfer is the same as the one central to all DL algorithms, we first define a loss function to define what we aim to achieve, and then we work on optimizing this function. In style-transfer problems, we know what we want to achieve: conserving the content of the original image while adopting the style of the reference image. Now all we need to do is to define both content and style in a mathematical representation, and then define an appropriate loss function to minimize.

The key notion in defining the loss function is to remember that we want to preserve content from one image and style from another:

 Content loss—Calculate the loss between the content image and the combined image. Minimizing this loss means the combined image will have more content from the original image.

 Style loss—Calculate the loss in style between the style image and the combined image. Minimizing this loss means the combined image will have style similar to the style image.

 Variation /Noise loss —This is called the total variation loss. It measures the noise in the combined image. Minimizing this loss creates an image with a higher spatial smoothness.

CNNs learn the information in the training set through successive filters. Each layer of the network deals with features at a different level of abstraction, so the complexity of the features generated depends on the layer's location in the network. Earlier layers learn low-level features; the deeper the layer is in the network, the more identifiable the extracted features are.

- Once a network is trained, we can run it in reverse to adjust the original image slightly so that a given output neuron (such as the one for faces or certain animals) yields a higher confidence score. This technique can be used for visualizations to better understand the emergent structure of the neural network and is the basis for the DeepDream concept.

- DeepDream processes the input image at different scales called octaves. We pass each scale, re-inject image details, pass it through the DeepDream algorithm, and then upscale the image for the next octave.

- The DeepDream algorithm is similar to the filter-visualization algorithm. It runs the ConvNet in reverse to generate output based on the representations extracted by the network.

- DeepDream differs from filter-visualization in that it needs an input image and maximizes the entire layer, not specific filters within the layer. This allows DeepDream to mix together a large number of features at once.

- DeepDream is not specific to images—it can be used for speech, music, and more.

- Neural style transfer is a technique that trains the network to preserve the style(texture, color, patterns) of the style image and preserve the content of the content image. The network then creates a new combined image that combines the style of the style image and the content from the content image.

- Intuitively, if we minimize the content, style, and variation losses, we get a new image that contains low variance in content and style from the content and style images, respectively, and low noise.

- Different values for content weight, style weight, and total variation weight will give you different results.

Overview of deep learning on Edge Devices:

In terms of computers, the edge is the very end device that sees things and measures parameters.

Deep learning on edge devices implies injecting AI into the edge device so that along with seeing, it can also analyze an image and report its content. An example of an edge device for computer vision is a camera.

Edge computing makes image recognition on premises quick and efficient. The AI component inside a camera consists of a powerful yet tiny processor that has deep learning capabilities. This AI on the edge can perform a mix of three separate functions, depending on the choice of hardware and software platforms you use:

- ➢ Hardware acceleration to make the device run faster
- ➢ Software optimization to reduce the model size and remove unnecessary components
- ➢ Interacting with the cloud to batch process image and tensors.

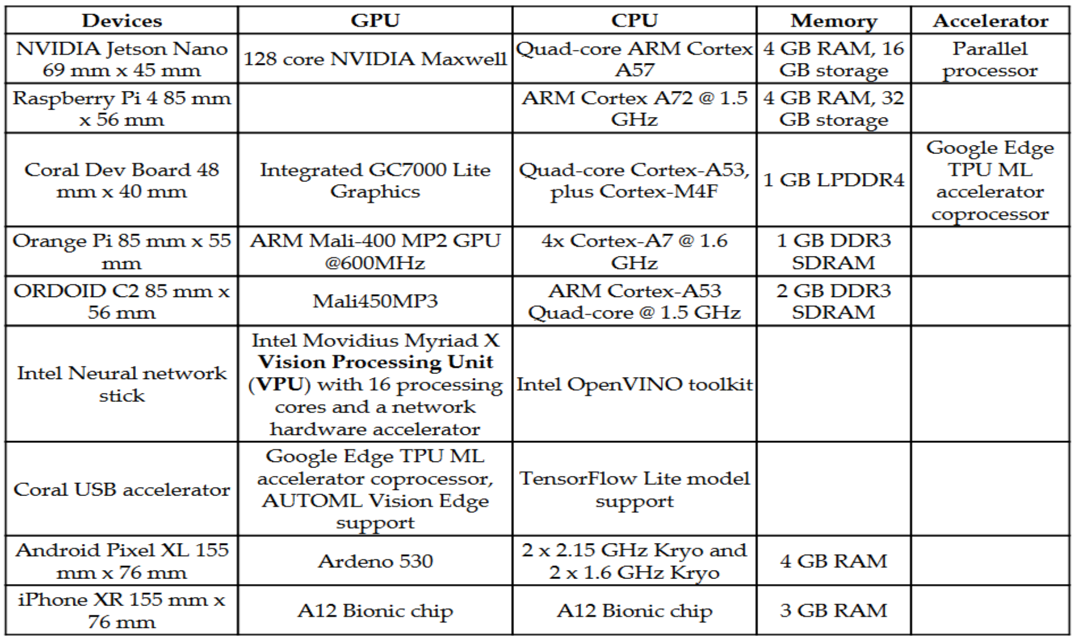
The benefit of this is increased speed, reduced bandwidth, increased data privacy, and network scalability. This is done by embedding a controller inside the camera to give the camera the processing power it needs to work.

Edge computing means the workload is transferred from the cloud to the device. This requires highly efficient edge devices, optimized software to perform detection without significant lag, and an efficient data transfer protocol to send select data to the cloud for processing and then feed the output back to the edge device for real time decision making. Selecting the correct edge device depends on your application requirements and how it interfaces with the rest of the subsystem.

Some examples of edge devices are as follows:

- ➢ NVIDIA Jetson Nano
- ➢ Raspberry Pi + Intel neural network stick
- ➢ Coral Dev board + Coral USB accelerator
- ➢ Orange Pi + Intel neural network stick
- ➢ ORDOID C2
- ➢ Android phone
- ➢ iOS phone

The following table summarizes the performance specifications of the various edge devices listed previously. You can use this table to determine your selection process:



Devices	GPU	CPU	Memory	Accelerator
NVIDIA Jetson Nano 69 mm x 45 mm	128 core NVIDIA Maxwell	Quad-core ARM Cortex A57	4 GB RAM, 16 GB storage	Parallel processor
Raspberry Pi 4 85 mm x 56 mm		ARM Cortex A72 @ 1.5 GHz	4 GB RAM, 32 GB storage	
Coral Dev Board 48 mm x 40 mm	Integrated GC7000 Lite Graphics	Quad-core Cortex-A53, plus Cortex-M4F	1 GB LPDDR4	Google Edge TPU ML accelerator coprocessor
Orange Pi 85 mm x 55 mm	ARM Mali-400 MP2 GPU @600MHz	4x Cortex-A7 @ 1.6 GHz	1 GB DDR3 SDRAM	
ORDOID C2 85 mm x 56 mm	Mali450MP3	ARM Cortex-A53 Quad-core @ 1.5 GHz	2 GB DDR3 SDRAM	
Intel Neural network stick	Intel Movidius Myriad X **Vision Processing Unit** (**VPU**) with 16 processing cores and a network hardware accelerator	Intel OpenVINO toolkit		
Coral USB accelerator	Google Edge TPU ML accelerator coprocessor, AUTOML Vision Edge support	TensorFlow Lite model support		
Android Pixel XL 155 mm x 76 mm	Ardeno 530	2 x 2.15 GHz Kryo and 2 x 1.6 GHz Kryo	4 GB RAM	
iPhone XR 155 mm x 76 mm	A12 Bionic chip	A12 Bionic chip	3 GB RAM	

2. Techniques used for GPU/CPU optimization:

a) Choose the right framework
b) Optimize the data pipeline
c) Use mixed precision training
d) Apply model pruning and quantization
e) Leverage parallelism and distribution

a.Choose the right framework

Different AI and machine learning frameworks have different strengths and weaknesses when it comes to GPU and CPU optimization. For example, TensorFlow and PyTorch are popular frameworks that support both GPU and CPU devices, but they have different ways of handling data parallelism, distributed training, and memory management. You should choose the framework that best suits your needs, goals, and preferences, and also consider the availability and compatibility of the hardware and software platforms you are using.

b.Optimize the data pipeline

One of the common bottlenecks in machine learning is the data pipeline, which refers to the process of loading, preprocessing, augmenting, and feeding data to the model. If the data pipeline is slow or inefficient, it can limit the performance of the GPU and CPU devices, and waste valuable resources. To optimize the data pipeline, you should use techniques such as caching, prefetching, batching, sharding, and compression, depending on the characteristics and size of your data. You should also monitor the data throughput and utilization of the GPU and CPU devices, and adjust the data pipeline accordingly.

c. Use mixed precision training

Mixed precision training is a technique that uses both 16-bit and 32-bit floating point numbers to represent the model parameters, gradients, and activations. This can reduce the memory usage and bandwidth requirements of the GPU and CPU devices, and also increase the speed and accuracy of the model. However, mixed precision training also requires careful tuning and testing, as it can introduce numerical instability and overflow issues. You should use mixed precision training only when it is supported by your framework and hardware, and when it can benefit your model performance.

d. Apply model pruning and quantization

Model pruning and quantization are techniques that reduce the size and complexity of the model, by removing or compressing some of the model parameters or operations. This can improve the efficiency and portability of the model, and also reduce the computational and memory demands of the GPU and CPU devices. However, model pruning and quantization can also affect the quality and accuracy of the model, so you should balance the trade-offs between speed and performance. You should apply model pruning and quantization only when it is necessary or beneficial for your use case and deployment scenario.

e. Leverage parallelism and distribution

Parallelism and distribution are techniques that use multiple GPU and CPU devices to perform the model training or inference tasks concurrently or collaboratively. This can increase the scalability and throughput of the model, and also overcome the limitations of a single device. However, parallelism and distribution also introduce challenges such as communication overhead, synchronization issues, and load balancing problems. You should leverage parallelism and distribution only when you have access to sufficient and compatible hardware resources, and when you can implement them effectively and efficiently with your framework.

3. Overview of MobileNet ,

MobileNets, a class of light weight deep convolutional neural networks that are vastly smaller in size and faster in performance than many other popular models. we can work with MobileNets using TensorFlow's Keras API.

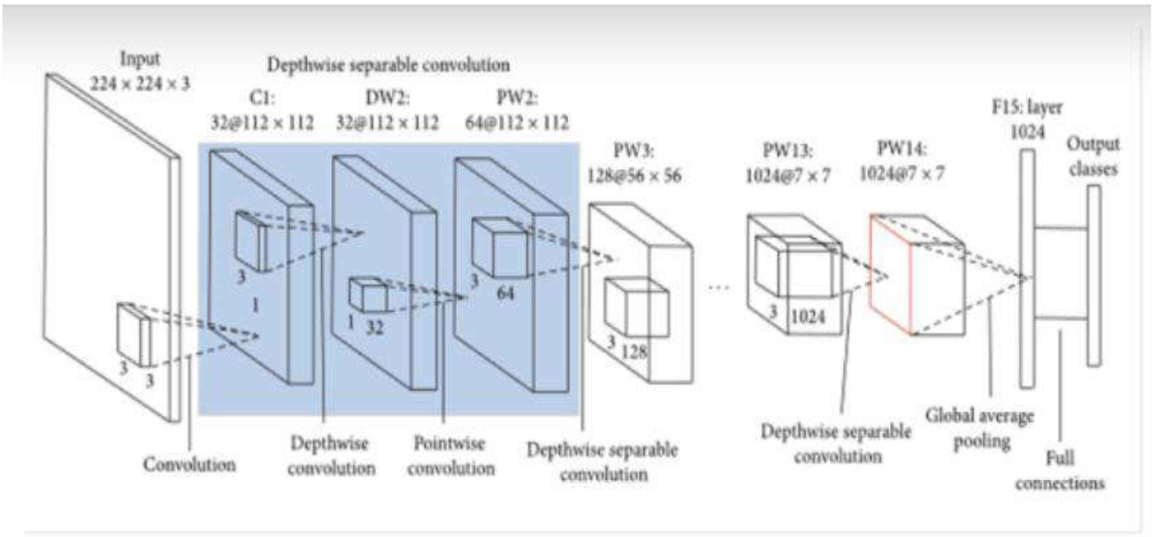
MobileNets are a class of small, low-latency, low-power models that can be used for classification, detection, and other common tasks convolutional neural networks are good for. Because of their small size, these are considered great deep learning models to be used on mobile devices.

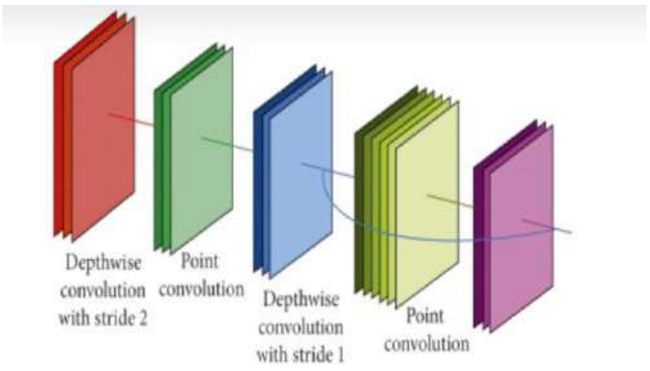
The size of the full VGG16 network on disk is about 553 megabytes. The size of one of the currently largest MobileNets is about 17 megabytes, so that is a huge difference, especially when you think about deploying a model to a mobile app or running it in the browser.

Model	Size	Parameters
VGG16	553 MB	138,000,000
Mobile Net	17 MB	4,200,000

This vast size difference is due to the number of parameters within these networks. For example, VGG16 has 138 million parameters, while the 17-megabyte Mobile Net has only 4.2 million. Aside from the size of the networks on disk, the size of the networks in memory also grows as the number of network parameters grow. Now, while MobileNets are faster and smaller than other major networks, like VGG16, for example, there is a tradeoff. That tradeoff is accuracy, but don't let this discourage you.

MobileNets Architecture:





MobileNet V2 model has 53 convolution layers and 1 AvgPool with nearly 350 GFLOP. It has two main components:

- Inverted Residual Block
- Bottleneck Residual Block

There are two types of Convolution layers in MobileNet V2 architecture:

- 1x1 Convolution
- 3x3 Depthwise Convolution

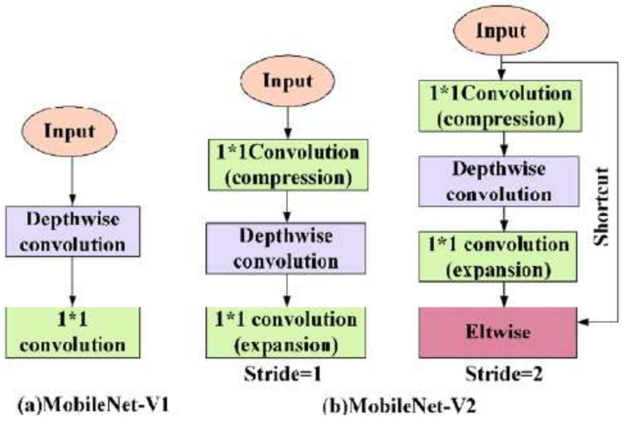
These are the two different components in MobileNet V2 model:

Each block has 3 different layers:

- 1x1 Convolution with Relu6
- Depthwise Convolution
- 1x1 Convolution without any linearity

There are Stride 1 Blocks and Stride 2 Blocks.

The internal components of the two blocks are as follows:



Stride 1 Block:

- Input ,1x1 Convolution with Relu6, Depthwise Convolution with Relu6, 1x1 Convolution without any linearity, Add
-

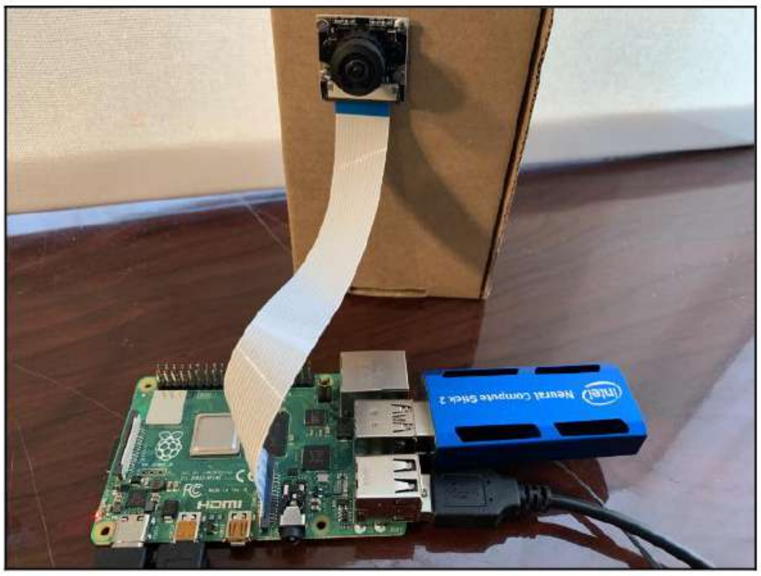
Stride 2 Block:

- Input. 1x1 Convolution with Relu6 ,Depthwise Convolution with stride=2 and Relu6, 1x1 Convolution without any linearity

4. Image processing with a Raspberry Pi

Raspberry Pi is a single-board tiny computer without a GPU that can be connected to an external camera and other sensor modules and can be programmed in Python to perform computer vision work such as object detection. Raspberry Pis have built-in Wi-Fi, so they can be connected to the internet seamlessly to receive and transfer data. Because of its tiny shape and powerful computing, the Raspberry Pi is a perfect example of an edge device for IoT and computer vision work. Detailed information on Raspberry Pi can be found at https://www.raspberrypi.org/products/.

The following photo shows the complete setup for a Raspberry Pi:



Step1 : Raspberry Pi hardware setup

Step 2 : Raspberry Pi camera software setup

Step3 : OpenCV installation in Raspberry Pi

1. Enter the following commands in the Terminal to install the necessary components for OpenCV:

```
$sudo su
$apt update && apt upgrade -y
$apt install build-essential
```

2. Install CMake to manage the build process:

```
$wget
https://github.com/Kitware/CMake/releases/download/v3.14.4/cmake-3.
14.4.tar.gz
$tar xvzf cmake-3.14.4.tar.gz
$cd ~/cmake-3.14.4
```

3. Install any additional dependencies, such as `bootstrap`:

```
$./bootstrap
$make -j4
$make install
```

4. Install OpenCV from the source in the Raspberry Pi drive:

```
$git clone https://github.com/opencv/opencv.git
$cd opencv && mkdir build && cd build
$cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/usr/local
..
$make -j4
$make install
```

The preceding command will install OpenCV.

5. To verify the installation process, open another Terminal while keeping the current one open, and enter the following command:

```
$python3
>>> import cv2
>>> cv2.__version__
```

This should display the latest OpenCV version that's installed on your system.

OpenVINO installation in Raspberry Pi

OpenVINO is an Intel trademark and stands for Open Visual Inference and Neural Network Optimization toolkit. It provides developers with a deep learning acceleration toolkit on Intel hardware such as CPUs (Intel Core and Xeon processors), GPUs (Intel Iris Pro graphics and HD graphics), VPUs (Intel Movidius neural computing stick), and FPGAs (Intel Arria 10GX).

To download OpenVINO on your desktop, go to https://software.intel.com/en-us/openvino-toolkit/choose-download.

After downloading OpenVINO, you will have to enter your name and email. Then, you will be able to download OpenVINO. Follow the instructions to unzip the file and install the dependencies. Note that this process will not work for Raspberry Pi. So, for Raspberry Pi, use the following commands

```
$cd ~/dldt/inference-engine
$mkdir build && cd build
$cmake -DCMAKE_BUILD_TYPE=Release \
-DCMAKE_CXX_FLAGS='-march=armv7-a' \
-DENABLE_MKL_DNN=OFF \
-DENABLE_CLDNN=OFF \
-DENABLE_GNA=OFF \
-DENABLE_SSE42=OFF \
-DTHREADING=SEQ \
..
$make
```

The preceding commands will return a series of displays in the Terminal.

5. Model conversion and inference using OpenVINO

To use a pre-trained model or a custom trained model to make inference using openVINO. Inference is the process of using a model to perform object detection or classification and is divided into three steps:

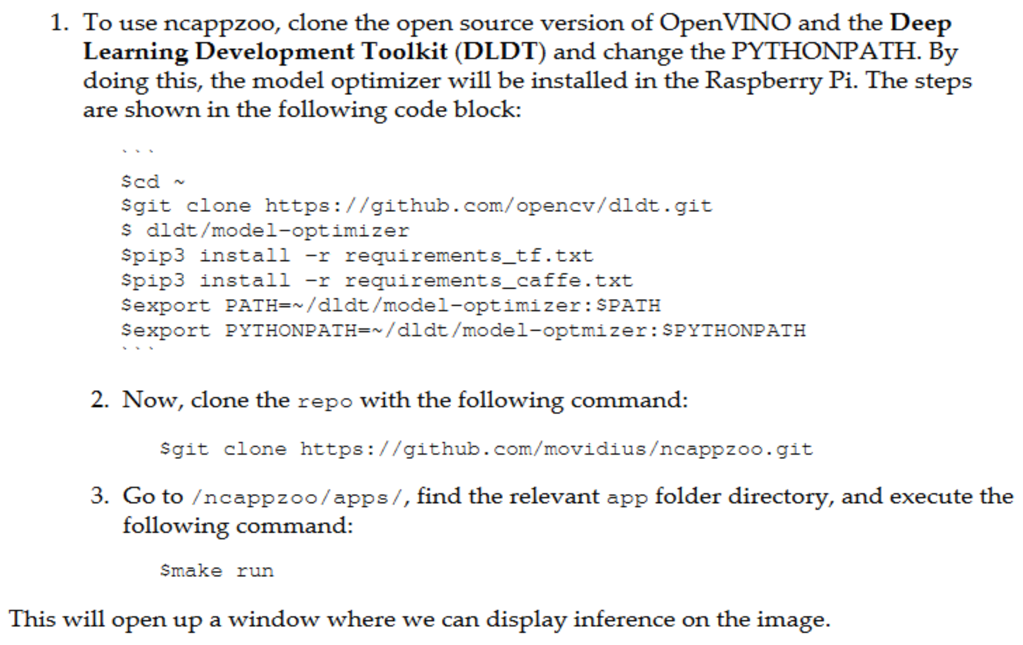
i. Running inference using a pre-trained model from ncappzoo.
ii. Converting the custom model into IR format for inference.
iii. Summary of all the steps involved in a flowchart.

These steps will be discussed in detail in the following subsections

i. Running inference in a Terminal using ncappzoo

OpenVINO's toolkit installation for Raspberry Pi is different to how it's done on a regular PC. The installation for Raspberry Pi does not include the Model Optimizer. Neural Compute Application Zoo (ncpappzoo) is an open source repository.

To use ncappzoo:



1. To use ncappzoo, clone the open source version of OpenVINO and the **Deep Learning Development Toolkit** (**DLDT**) and change the PYTHONPATH. By doing this, the model optimizer will be installed in the Raspberry Pi. The steps are shown in the following code block:

```
```
$cd ~
$git clone https://github.com/opencv/dldt.git
$ dldt/model-optimizer
$pip3 install -r requirements_tf.txt
$pip3 install -r requirements_caffe.txt
$export PATH=~/dldt/model-optimizer:$PATH
$export PYTHONPATH=~/dldt/model-optmizer:$PYTHONPATH
```
```

2. Now, clone the `repo` with the following command:

```
$git clone https://github.com/movidius/ncappzoo.git
```

3. Go to `/ncappzoo/apps/`, find the relevant `app` folder directory, and execute the following command:

```
$make run
```

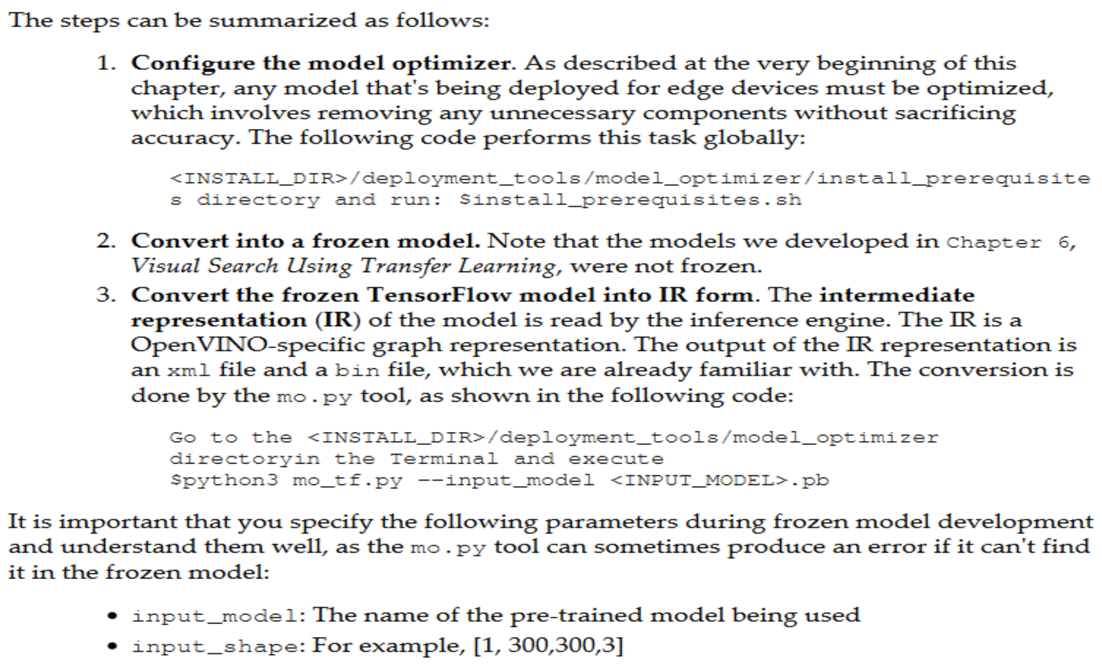
This will open up a window where we can display inference on the image.

ii) Converting the pre-trained model for inference

This section describes the steps involved in converting a custom TensorFlow model created using the TensorFlow Keras object classification model we developed in Chapter 6, Visual Search Using Transfer Learning, or using a model created using the TensorFlow Object

Detection API, as we did in the previous chapter. The steps described in the previous section will work if you already plan on using a pre-trained optimized model from Intel Open Source Technology Center. In the next section, we'll describe how to perform conversion using two types of TensorFlow models.

iii) Converting from a TensorFlow model developed using Keras


The steps can be summarized as follows:

1. **Configure the model optimizer.** As described at the very beginning of this chapter, any model that's being deployed for edge devices must be optimized, which involves removing any unnecessary components without sacrificing accuracy. The following code performs this task globally:

   ```
   <INSTALL_DIR>/deployment_tools/model_optimizer/install_prerequisite
   s directory and run: $install_prerequisites.sh
   ```

2. **Convert into a frozen model.** Note that the models we developed in `Chapter 6`, *Visual Search Using Transfer Learning*, were not frozen.
3. **Convert the frozen TensorFlow model into IR form.** The **intermediate representation (IR)** of the model is read by the inference engine. The IR is a OpenVINO-specific graph representation. The output of the IR representation is an `xml` file and a `bin` file, which we are already familiar with. The conversion is done by the `mo.py` tool, as shown in the following code:

   ```
   Go to the <INSTALL_DIR>/deployment_tools/model_optimizer
   directoryin the Terminal and execute
   $python3 mo_tf.py --input_model <INPUT_MODEL>.pb
   ```

It is important that you specify the following parameters during frozen model development and understand them well, as the `mo.py` tool can sometimes produce an error if it can't find it in the frozen model:

- `input_model`: The name of the pre-trained model being used
- `input_shape`: For example, [1, 300,300,3]

The preceding diagram indicates that the process can be divided into three key segments:

Preprocessing step: Here, we install OpenCV and set the environmental variable.

Model preparation: Here, we convert the model into the optimized model format.

Inference: This can be divided into two independent methods – by performing inference over the Terminal by going to the appropriate director and executing make run or performing inference using the Python code. I found that on an Intel PC, all these steps are easy to execute. However, in a Raspberry Pi environment, working in the Terminal using the make Run command causes different types of errors; for example, sometimes, it cannot find the .bin or .xml file. Other times, the environmental variable isn't initialized or the CMakeLists.txt file couldn't be found. Executing the Python code provided does not create any of those issues in Raspberry Pi. This also gives us a better understanding of the computer vision environment as all we are doing is getting the model, understanding the input and output, and then generating some code so that we can display our results.

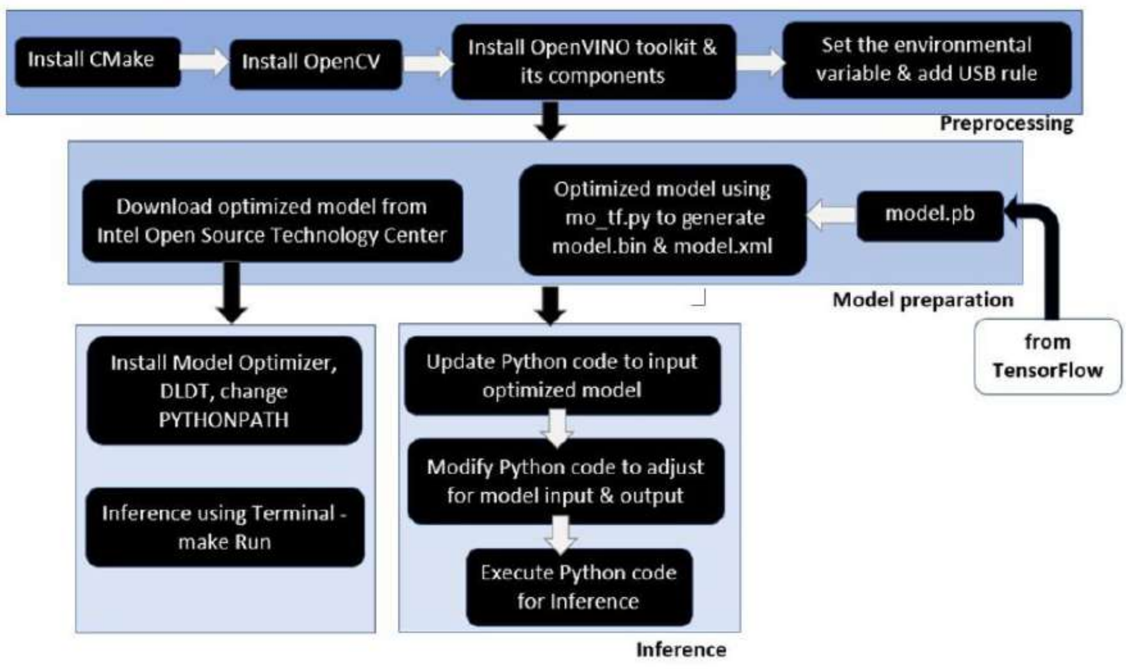

Install CMake
Install OpenCV
Install OpenVINO toolkit & its components
Set the environmental variable & add USB rule
Preprocessing
Download optimized model from Intel Open Source Technology Center
Optimized model using mo_tf.py to generate model.bin & model.xml
model.pb
Model preparation
from TensorFlow
Install Model Optimizer, DLDT, change PYTHONPATH
Inference using Terminal - make Run
Update Python code to input optimized model
Modify Python code to adjust for model input & output
Execute Python code for Inference
Inference

Fig : The entire model inference process flow chart

The model optimization techniques we learned so far:
1) Batch Normalization operation fused with convolution operations -OpenVINO uses this
2) Move convolution with stride greater than 1 with filter size 1 to upperconvolution layers. Add pooling layer to align input shape – OpenVINO uses this
3) Replace large filter by two small filters, example replace 3 x 3 x 3 with #32 with 3 x 3 x 1 with #3 and 1 x 1 x 3 with #32 - MobileNet uses this

6. Converting a TensorFlow model developed using the TensorFlowObject Detection API ,

The model that we obtained in Chapter 10, Object Detection Using R-CNN, SSD, and R-FCN,using the TensorFlow Object Detection API is already frozen. Follow these steps to convert it:

1. Before converting, refer to the following link to configure the model optimizer: `https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_Config_Model_Optimizer.html`.
2. Now, we are ready for conversion. Prepare three files, as follows:
    - The frozen inference graph for the model: This is a file with a `.pb` extension that's generated by training the model (R-CNN, SSD, or R-FCN) with your custom image. In this example, it is `frozen_inference_graph_fasterRCNN.pb`.
    - The configuration `json` file: This is the corresponding `json` file that describes the custom attributes, nodes, ports, endpoints, and starting points of your frozen TensorFlow graph.
    - The config file used to generate the model: This is the same file that you used to generate the model using the TensorFlow Object Detection API in `Chapter 10`, *Object Detection Using R-CNN, SSD, and R-FCN*. For example, for R-CNN, we used `faster_rcnn_inception_v2_pets.config`.

After you've completed the previous steps, execute the following code in the Terminal:

```
$python3 mo_tf.py --input_model frozen_inference_graph_fasterRCNN.pb --
transformations_config faster_rcnn_support_api_v1.14.json --
tensorflow_object_detection_api_pipeline_config
faster_rcnn_inception_v2_pets.config
```

Here, you need to replace the file with `.pb` extension with your specific model file name with `.pb` extension. The result of the conversion will be an `xml` and a `bin` file.

7. Application of TensorFlow Lite ,

TensorFlow Lite is the TensorFlow deep learning framework for inference on edge devices. Similar to OpenVINO, TensorFlow Lite has built-in pre-trained deep learning modules. Alternatively, an existing model can be converted into TensorFlow Lite format for on-device inference. Currently, TensorFlow Lite provides inference support for PCs with a built-in or external camera, Android devices, iOS devices, Raspberry Pis, and tiny microcontrollers. Visit https://www.tensorflow.org/lite for details on TensorFlow Lite.

The TensorFlow Lite converter takes a TensorFlow model and generates a FlatBuffer tflite file. A FlatBuffer file is an efficient cross-platform library that can be used to access binary serialized data without the need for parsing. Serialized data is usually a text string. Binary serialized data is binary data written in string format. For details on FlatBuffer, please refer to the following link: https://google.github.io/flatbuffers/.

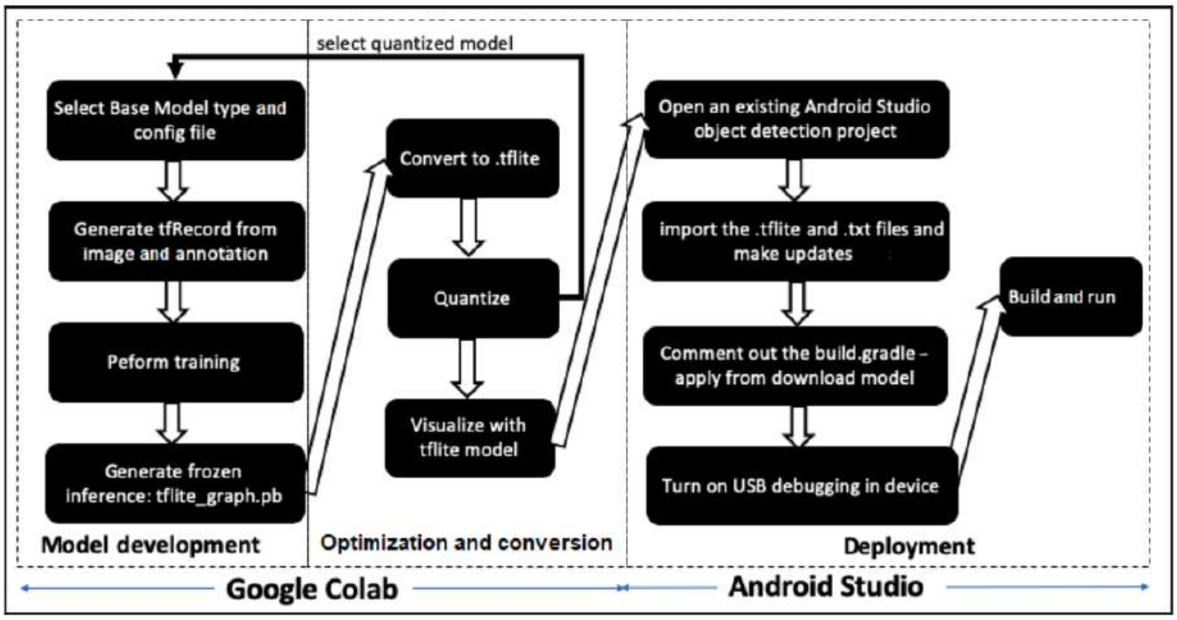
The TensorFlow output model can be of these types:

- **SavedModel format**: A saved model is executed by tf.saved_model and the output is saved_model.pb. It is a complete TensorFlow format that includes learned weights and a graph structure.
- **tf.keras models format:** The tf.kears models are tf.keras.model.compile files
- **Concrete functions**: A TensorFlow graph with a single input and output

Object detection on Android phones using TensorFlow Lite:

The steps required to deploy the TensorFlow lite converted model will be described in this section. Alternatively, you can follow the instructions at https://github.com/tensorflow/examples/tree/master/lite/examples/image_classification/android to build a sample app.

A detailed flow chart regarding object detection on Android phones is as follows:



select quantized model
Select Base Model type and config file
Generate tfRecord from image and annotation
Peform training
Generate frozen inference: tflite_graph.pb
Convert to .tflite
Quantize
Visualize with tflite model
Open an existing Android Studio object detection project
import the .tflite and .txt files and make updates
Comment out the build.gradle - apply from download model
Turn on USB debugging in device
Build and run
Model development
Optimization and conversion
Deployment
Google Colab
Android Studio

We need two files:

- The TensorFlow Lite converted file in `.tflite` form
- An updated labelmap `.txt` file showing the class

The `.tflite` file comes directly from Google Colab if we export it, as explained in the *TensorFlow Object Detection API – toco* section. The `lablemap.txt` file comes from the `label_map.pbtxt` file by listing only the names of the class.

We need two files:

- The TensorFlow Lite converted file in `.tflite` form
- An updated labelmap `.txt` file showing the class

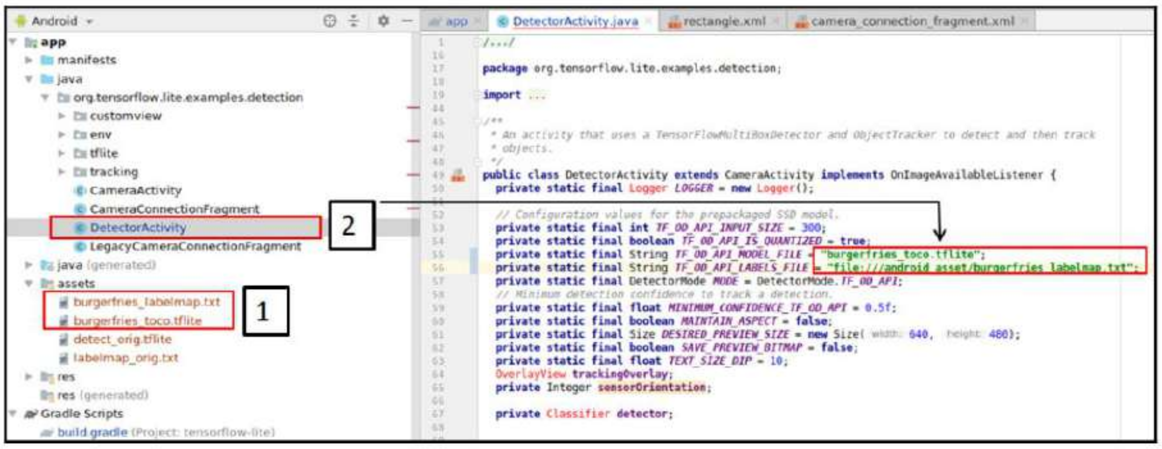
The `.tflite` file comes directly from Google Colab if we export it, as explained in the *TensorFlow Object Detection API – toco* section. The `lablemap.txt` file comes from the `label_map.pbtxt` file by listing only the names of the class.

The steps for taking a `tflite` model and generating inference in an Android phone are as follows:

1. In the burger and fries example, the `.txt` file will have one column and two rows, as follows:

    ```
    burger
    fries
    ```

2. Place both of those files in your PC in the same directory. Open Android Studio. If you have never used Android Studio before, you will have to download it from `https://developer.android.com/studio`. Follow the download instructions given at the site.
3. After downloading it, on Mac or Windows, double-click to open it. For Linux, you'll have to go to a Terminal and navigate to the `android-studio/bin` directory and type `./studio.h`.
4. Download some sample examples by typing the following in the Terminal: `git clone https://github.com/tensorflow/examples`.
5. Open Android Studio and open an existing project, setting the folder to `examples/lite/examples/object_detection/android`.
6. Within the Android Studio project, go to the app and then go to **assets**, as shown in the following screenshot:



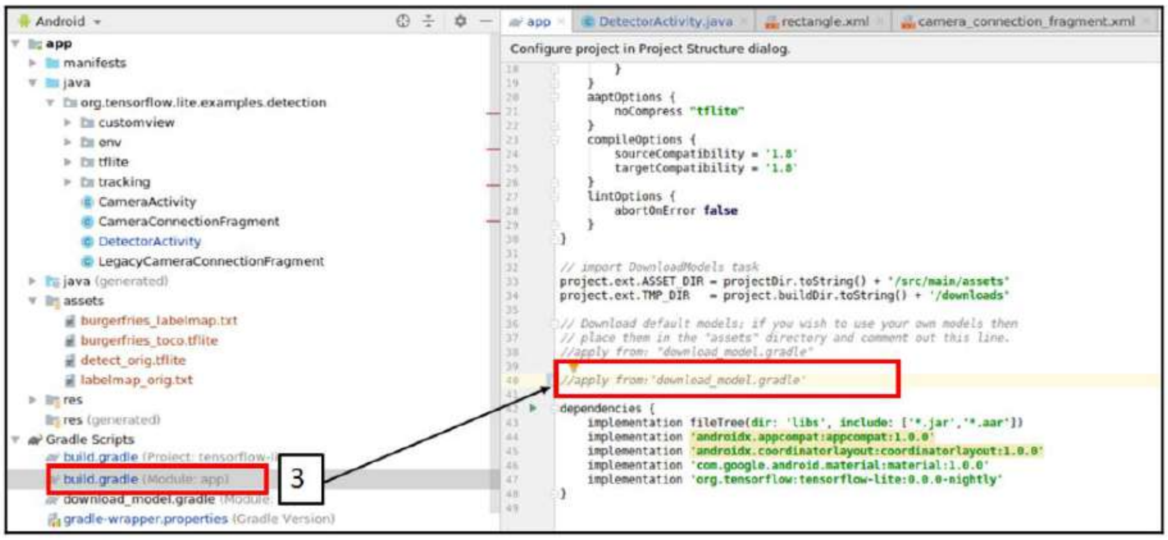
```
/.../

package org.tensorflow.lite.examples.detection;

import ...

/**
 * An activity that uses a TensorFlowMultiBoxDetector and ObjectTracker to detect and then track
 * objects.
 */
public class DetectorActivity extends CameraActivity implements OnImageAvailableListener {
  private static final Logger LOGGER = new Logger();

  // Configuration values for the prepackaged SSD model.
  private static final int TF_OD_API_INPUT_SIZE = 300;
  private static final boolean TF_OD_API_IS_QUANTIZED = true;
  private static final String TF_OD_API_MODEL_FILE = "burgerfries_toco.tflite";
  private static final String TF_OD_API_LABELS_FILE = "file:///android_asset/burgerfries_labelmap.txt";
  private static final DetectorMode MODE = DetectorMode.TF_OD_API;
  // Minimum detection confidence to track a detection.
  private static final float MINIMUM_CONFIDENCE_TF_OD_API = 0.5f;
  private static final boolean MAINTAIN_ASPECT = false;
  private static final Size DESIRED_PREVIEW_SIZE = new Size( width: 640,  height: 480);
  private static final boolean SAVE_PREVIEW_BITMAP = false;
  private static final float TEXT_SIZE_DIP = 10;
  OverlayView trackingOverlay;
  private Integer sensorOrientation;

  private Classifier detector;
```

7. Right-click on **assets** and select **Show in Files** from the menu list. Drag and drop the `.tflite` and `.txt` files we created in the very first step into the **assets** directory. Close the folder and go back to **Android Studio**.
8. Double-click on the `.txt` file to open it and add a new line at the top. Fill it with `???`. So, the `.txt` file will have three lines for the two classes:

```
???
Burger
fries
```

9. Select **Java**, then **Tracking**, and double-click on **DetectorActivity**. Change the names of the `.tflite` and `.txt` files to their actual names, as shown in the following screenshot. Then, click on **build gradle**:

Note that for the `.txt` file, keep the path, that is, `file:///android_asset/burgerfries_labelmap.txt`. Later, we will mention that if the `.tflite` file is not generated using `toco`, then keeping the previous path will cause the app to crash. To prevent a crash, you can just keep the filename (for example, `burgerfries_labelmap.txt`). However, note that this doesn't create a bounding box for the detected image.

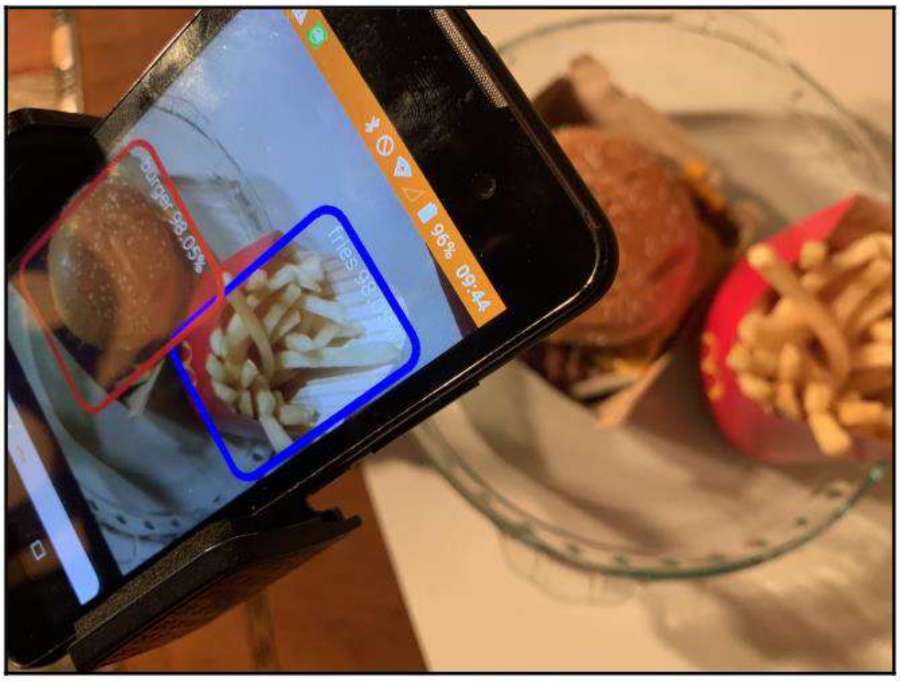


Configure project in Project Structure dialog.

```
        }
    }
    aaptOptions {
        noCompress "tflite"
    }
    compileOptions {
        sourceCompatibility = '1.8'
        targetCompatibility = '1.8'
    }
    lintOptions {
        abortOnError false
    }
}

// import DownloadModels task
project.ext.ASSET_DIR = projectDir.toString() + '/src/main/assets'
project.ext.TMP_DIR   = project.buildDir.toString() + '/downloads'

// Download default models; if you wish to use your own models then
// place them in the "assets" directory and comment out this line.
//apply from: "download_model.gradle"

//apply from:'download_model.gradle'

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar','*.aar'])
    implementation 'androidx.appcompat:appcompat:1.0.0'
    implementation 'androidx.coordinatorlayout:coordinatorlayout:1.0.0'
    implementation 'com.google.android.material:material:1.0.0'
    implementation 'org.tensorflow:tensorflow-lite:0.0.0-nightly'
}
```

10. Comment out `///apply from download_model.gradle`. Verify that the dependencies appear, as shown in the preceding screenshot.
11. Now, connect your Android device to your PC using a USB cable. Go to your device and, under settings, click on **Developer options** to make sure it is on. Then, turn on USB debugging. For many Android phones, this option comes up by itself.

12. Click on **Build** at the top and then **Make Project**. After Android Studio has finished compiling (look at the bottom of the screen to see if all activities have been completed), click on **Run** and then **Run app**. The app will be downloaded to your device. An option box will appear on your device; select `ok` to be able to run the app. The following is an image of the app working:



The phone is able to clearly detect a real picture of both burgers and fries with very high accuracy. This concludes the Android app deployment exercise.

11. Object detection on Raspberry Pi using TensorFlow Lite

The Python `quickstart` package listed under TensorFlow Lite (`https://www.tensorflow.org/lite/guide/python`) describes how to install the TensorFlow Lite package for Raspberry Pi. However, there are several notable exceptions. Due to this, we have listed the entire process here:

1. First and foremost, install the TensorFlow Lite interpreter. Raspberry Pi has ARM7 and Python3.7 installed, so run the following two commands in the Terminal:

```
$sudo su
$pip3 install tflite_runtime-1.14.0-cp37-cp37m-linux_armv7l.whl
```

2. As per the official TensorFlow Lite documentation, some changes need to be made in the `label_image.py` file (`https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/examples/python`):

```
$import tflite_runtime.interpreter as tflite,
$interpreter = tf.lite.Interpreter(model_path=args.model_file)
```

3. Next, follow the steps provided at the following link to install the TensorFlow Lite directory and files: `https://github.com/tensorflow/examples.git`.
4. If everything goes well, you should have a directory in the Raspberry Pi called `pi/examples/lite/examples`. Within this folder, you should have directories such as `image_classification`, `object_detection`, `image_segmentation`, `posenet`, `style_transfer`, and so on.
5. Next, we will perform two examples on Raspberry Pi – one for image classification and another for object detection.

To perform the following steps for image classification:

1. Go to the `image_classification` directory, that is, `pi/examples/lite/examples/image_classification/raspberry_pi`, using the File Manager. You will see a file named `classify_picamera.py`. Now, go to `https://www.tensorflow.org/lite/guide/hosted_models` and download the object detection models and `label` files folder named `mobilenet_v2_1.0_224.tflite` and the `labels_mobilenet_v2_1.0_224.txt` file. Copy those files into `pi/examples/lite/examples/image_classification/raspberry_pi`.

2. Next, go to the directory using the Terminal using `pi/examples/lite/examples/image_classification/raspberry_pi` and execute the following command:

```
$Python3 classify_picamera.py –model mobilenet_v2_1.0_224.tflite
–labels labels_ mobilenet_v2_1.0_224.txt
```

3. You should see the Raspberry Pi camera module light up and start classifying images.

To perform object detection. Follow these steps:

1. Go to the object detection directory, that is, `pi/examples/lite/examples/object_detection/raspberry_pi`, using the File Manager. You will see a file called `detect_picamera.py`.
2. Now, go to `https://www.tensorflow.org/lite/guide/hosted_models` and download the object detection models and label files folder named as `coco_ssd_mobilenet_v1_1.0_quant_2018_06_29`. Within this folder, you will see two files: `detect.tflite` and `labelmap.txt`.
3. Copy those files into `pi/examples/lite/examples/object_detection/raspberry_pi`.
4. Next, go to the object detection directory using a Terminal using `pi/examples/lite/examples/object_detection/raspberry_pi` and execute the following command:

```
$Python3 detect_picamera.py –model detect.tflite –labels
labelmap.txt
```

Now, you should see the Raspberry Pi camera module light up and start showing bounding boxes around the images.

5. Next, copy the `burgerfries.tflite` and `labelmap` files into the folder. Then, change the Python path shown in the preceding command line to reflect your new filename and execute it. The following image is the image being used for `object_detection`: