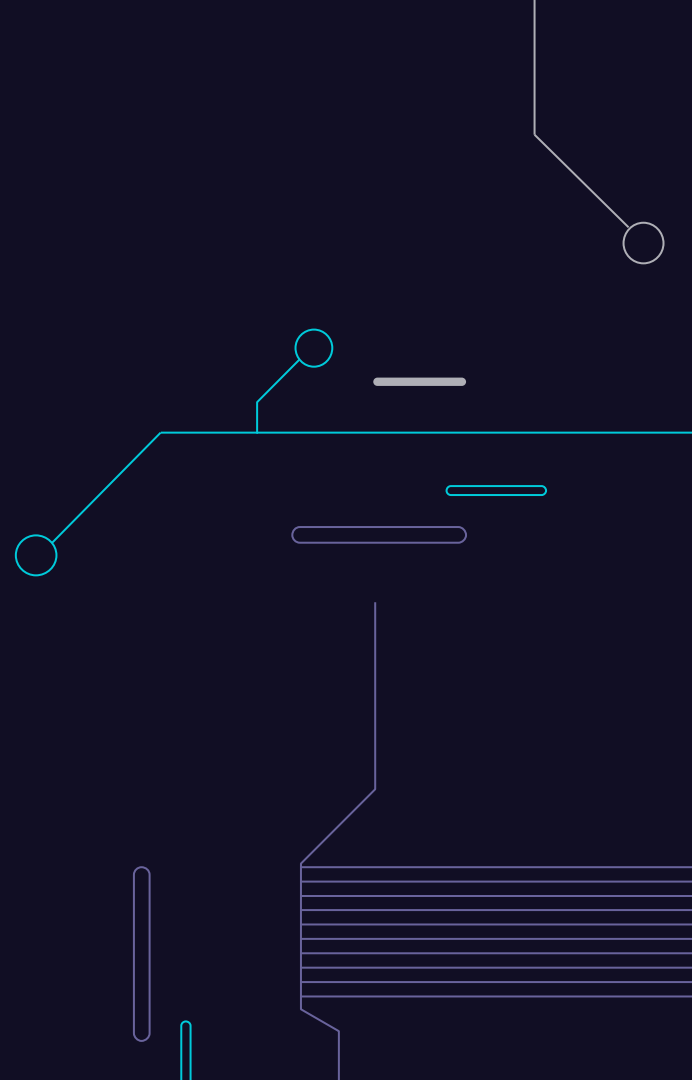


► **In The name
of God
the Merciful**



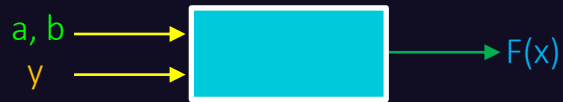
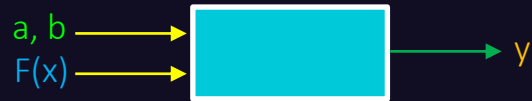


► Introduction to Neural Networks with OpenCV

Chapter 10

► What is AI & ML :

```
def Add(a, b):  
    return a + b
```



► artificial neural networks (ANNs)

- This chapter introduces a family of machine learning models called artificial neural networks (ANNs), or sometimes just neural networks.
- Recent versions of OpenCV contain an increasing amount of functionality related to ANNs – and, in particular, ANNs with many layers, called deep neural networks (DNNs).

► ANNs aim to provide superior accuracy in the following circumstances :

- There are many **input** variables, which may have **complex**, **nonlinear** relationships to each other.
- There are many **output** variables, which may have **complex**, **nonlinear** relationships to the **input** variables. (Typically, the output variables in a classification problem are the confidence scores for the classes, so if there are many classes, there are many output variables.)
- There are many **hidden** (unspecified) variables that may have **complex**, **nonlinear** relationships to the **input** and **output** variables. **DNNs** even aim to model multiple layers of **hidden** variables, which are interrelated primarily to each other rather than being related primarily to input or **output** variables.



These circumstances exist in many – perhaps most – real-world problems.

► in this chapter, we will cover the following topics:

- Understanding **ANNs** as a statistical model and as a tool for supervised machine learning.
- Understanding **ANN** topology or, in other words, the organization of an **ANN** into layers of interconnected neurons. Particularly, we will consider the topology that enables an **ANN** to act as a type of classifier known as a **multi-layer perceptron (MLP)**.
- Training and using **ANNs** as classifiers in **OpenCV**.
- Building an application that detects and recognizes **handwritten digits (0 to 9)**. For this, we will train an **ANN** based on a widely used dataset called **MNIST**, which contains samples of **handwritten digits**.
- Loading and using pre-trained **DNNs** in **OpenCV**. We will cover examples of object classification, face detection, and gender classification with **DNNs**.

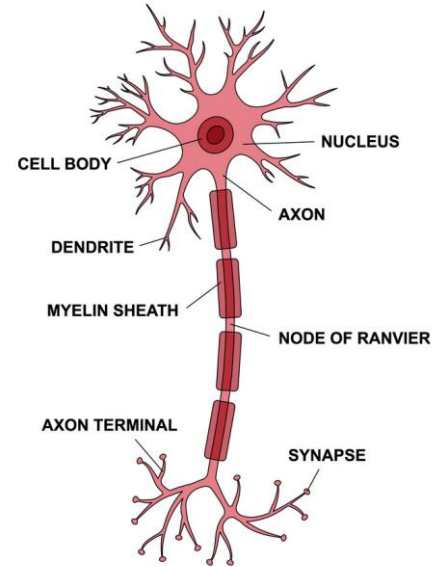


By the end of this chapter, you will be in a good position to train and use **ANNs** in **OpenCV**, to use pre-trained **DNNs** from a variety of sources, and to start exploring other libraries that allow you to train your own **DNNs**.

► WHAT IS A **ANNs**?

Let's define **ANNs** in terms of their basic role and components. Although much of the literature on **ANNs** emphasizes the idea that they are biologically inspired by the way neurons connect in a brain, we don't need to be biologists or neuroscientists to understand the fundamental concepts of an **ANN**.


Thus, **ANNs** are models that take a **complex reality**, **simplify it**, and deduce a function to (approximately) represent the statistical observations we would expect from that reality, in a mathematical form.





► **ANNs**, like other types of machine learning models, can learn from observations in one of the following ways:

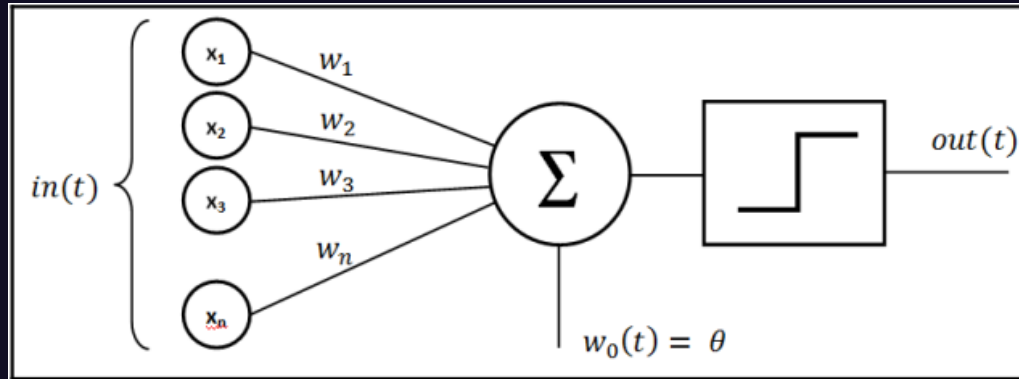
- 
- 01** Supervised learning
 - 02** Unsupervised learning
 - 03** Reinforcement learning



Throughout the remainder of this chapter, we will confine our discussions to **supervised learning**, as this is the most common approach to machine learning in the context of **computer vision**.

► Understanding neurons and perceptrons

Often, to solve a classification problem, an ANN is designed as a multi-layer perceptron (MLP), in which each neuron acts as a kind of binary classifier called a perceptron. The perceptron is a concept that dates back to the 1950s. To put it simply, a perceptron is a function that takes a number of inputs and produces a single value. Each of the inputs has an associated weight that signifies its importance in an activation function. The activation function should have a nonlinear response; for example, a sigmoid function (sometimes called an S-curve) is a common choice. A threshold function, called a discriminant, is applied to the activation function's output to convert it into a binary classification of 0 or 1. Here is a visualization of this sequence, with inputs on the left, the activation function in the middle, and the discriminant on the right:

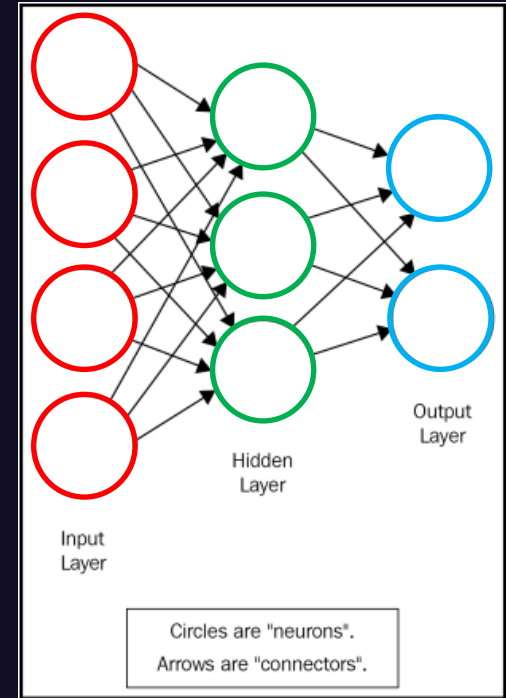


► Understanding the layers of a neural network

Here is a visual representation of a **neural network** :

As the preceding diagram shows, there are **at least** three distinct layers in a **neural network**: the **input layer**, the **hidden layer**, and the **output layer**. There can be more than one **hidden layer**; however, one **hidden layer** is enough to resolve many **real-life** problems. A **neural network** with multiple **hidden layers** is sometimes called a **deep neural network (DNN)**.

How do we determine the network's topology, and how many neurons do we need to create for each layer? Let's make this determination layer by layer.



► Choosing the size of the **input layer**

The number of nodes in the **input** layer is, by definition, the number of **inputs** into the network. For example, let's say you want to create an **ANN** to help you determine an animal's species based on measurements of its physical attributes. In principle, we can choose any measurable attributes. If we choose to classify animals based on **weight**, **length**, and **number of teeth**, that is a set of three attributes, and thus our network needs to contain three **input** nodes.

Are these three **input** nodes an adequate basis for species classification? Well, in a **real-life** problem, **surely not** – but in a toy problem, it depends on the **output** we are trying to achieve, and this is our next consideration.

► Choosing the size of the output layer

For a classifier, the number of nodes in the output layer is, by definition, the number of classes the network can distinguish. Continuing with the preceding example of an animal classification network, we can use an output layer of four nodes if we know we are going to deal with the following animals: dog, condor, dolphin, and dragon(!). If we try to classify data for an animal that is not in one of these categories, the network will predict the class that is most likely to resemble this unrepresented animal.

Now, we come to a difficult problem – the size of the hidden layer.

► Choosing the size of the hidden layer

There are no agreed-upon rules of thumb for choosing the size of the hidden layer; it must be chosen based on experimentation. For every real-world problem where you want to apply ANNs, you will need to train, test, and retrain your ANN until you find a number of hidden nodes that yield acceptable [accuracy](#).

Of course, even when choosing a parameter's value by experimentation, you might wish for the experts to suggest a starting value, or a range of values, for your tests. Unfortunately, there is no expert consensus on these points either.

► Choosing the size of the **hidden layer**

Some experts offer rules of thumb based on the following broad suggestions (these should be taken with a grain of salt):

- If the **input** layer is **large**, the number of **hidden** neurons should be between the size of the input layer and the size of the output layer – and, typically, closer to the size of the output layer.
- On the other hand, if the **input** and **output** layers are **both small**, the hidden layer should be the largest layer.
- If the **input** layer is small but the **output** layer is large, the hidden layer should be closer to the size of the input layer.

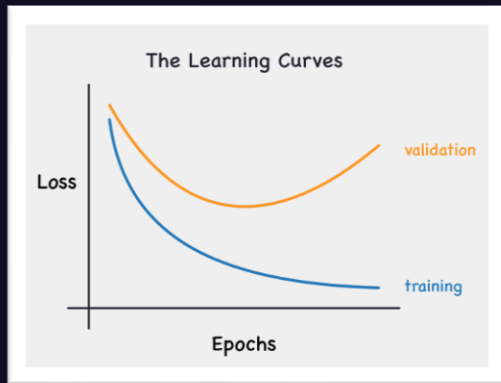
Other experts suggest that the number of training samples also needs to be taken into account; a greater number of training samples implies that a greater number of **hidden** nodes might be useful.

► Choosing the size of the hidden layer

One critical factor to keep in mind is **overfitting**. Overfitting occurs when there is such an inordinate amount of pseudo-information contained in the **hidden** layer, compared to the information actually provided by the training data, that the classification is not very meaningful. The larger the **hidden** layer, the **more training data** it requires in order for it to learn properly. Of course, as the size of the training dataset grows, so does the **training time**.

For some of the **ANN** sample projects in this chapter, we will use a **hidden** layer size of **60** as a starting point. Given a large training set, **60 hidden** nodes can yield decent **accuracy** for a variety of classification problems.

Now that we have a general idea of what **ANNs** are, let's see how **OpenCV** implements them, and how to put them to good use. We will start with a minimal code example. Then, we will flesh out the animal-themed classifier that we discussed in the previous two sections. Finally, we will work our way up to a more realistic application, in which we will classify **handwritten digits** based on image data.



► Training a basic ANN in OpenCV

OpenCV provides a class, `cv2.ml_ANN_MLP`, that implements an ANN as a multi-layer perceptron (MLP). This is exactly the kind of model we described earlier, in the Understanding neurons and perceptrons section.

To create an instance of `cv2.ml_ANN_MLP`, and to format data for this ANN's training and use, we rely on functionality in OpenCV's machine learning module, `cv2.ml`

Let's examine a dummy example as a gentle introduction to ANNs. This example will use completely meaningless data, but it will show us the basic API for training and using an ANN in OpenCV:

1. To begin, we import OpenCV and NumPy as usual:

```
import cv2
import numpy as np
```

2. Now, we create an untrained ANN:

```
ann = cv2.ml.ANN_MLP_create()
```


► Training a basic ANN in OpenCV

3. After creating the ANN, we need to configure its number of layers and nodes:

```
ann.setLayerSizes(np.array([9, 15, 9], np.uint8))
```

The layer sizes are defined by the NumPy array that we pass to the `setLayerSizes` method. The first element is the size of the input layer, the last element is the size of the output layer, and all the in-between elements define the sizes of the hidden layers. For example, `[9, 15, 9]` specifies 9 input nodes, 9 output nodes, and a single hidden layer with 15 nodes. If we changed this to `[9, 15, 13, 9]`, it would specify two hidden layers with 15 and 13 nodes, respectively.

4. We can also configure the activation function, the training method, and the training termination criteria, as follows:

```
ann.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, 0.6, 1.0)  
ann.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP, 0.1, 0.1)  
ann.setTermCriteria( (cv2.TERM_CRITERIA_MAX_ITER | cv2.TERM_CRITERIA_EPS, 100, 1.0))
```

Here, we are using a symmetrical sigmoid activation function (`cv2.ml.ANN_MLP_SIGMOID_SYM`)

and a backpropagation training method (`cv2.ml.ANN_MLP_BACKPROP`)

Backpropagation is an algorithm that calculates errors of predictions at the output layer, traces the sources of the errors backward through previous layers, and updates the weights in order to reduce errors.

► Training a basic ANN in OpenCV

5. Let's train the ANN. We need to specify training inputs (or samples, in OpenCV's terminology), the corresponding correct outputs (or responses), and whether the data's format (or layout) is one row per sample or one column per sample. Here is an example of how we train the model with a single sample:

```
training_samples = np.array( [[1.2, 1.3, 1.9, 2.2, 2.3, 2.9, 3.0, 3.2, 3.3]], np.float32)
layout = cv2.ml.ROW_SAMPLE
training_responses = np.array( [[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]], np.float32)
data = cv2.ml.TrainData_create( training_samples, layout, training_responses) ann.train(data)
```



Realistically, we would want to train any ANN with a larger dataset that contains far more than one sample. We could do this by extending `training_samples` and `training_responses` so that they contain multiple rows, representing multiple samples and their corresponding responses. Alternatively, we could call the ANN's `train` method multiple times, with new data each time. The latter approach requires some additional arguments for the `train` method, and it is demonstrated in the next section, Training an ANN- classifier in multiple epochs.

Note that in this case, we are training the ANN as a classifier. Each response is a confidence score for a class, and in this case, there are nine classes. We will refer to them by their 0-based indices, as classes 0 to 8. Our training sample in this case has a response of `[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]`, meaning that it is an instance of class 5 (with confidence 1.0), and it is definitely not an instance of any other class (as the confidence is 0.0 for every other class).

► Training a basic ANN in OpenCV

6. To complete our minimal tour of the ANN's API, let's make another sample, classify it, and print the result:

```
test_samples = np.array( [[1.4, 1.5, 1.2, 2.0, 2.5, 2.8, 3.0, 3.1, 3.8]], np.float32)
prediction = ann.predict(test_samples) print(prediction)
```

This will print the following result:

```
(5.0, array([[ -0.08763029, -0.01616517, 0.13196233,
              0.0402631,   0.05711843,  1.1642447 ,
              0.18130444,   0.1857026 , -0.07486832 ]], dtype=float32))
```

As you may have guessed, the output of a prediction is a **tuple**, with the first value being the class and the second being an array containing the probabilities for each class. The predicted class will have the highest value.

Let's move on to a slightly more believable example - animal classification.

► Training an ANN classifier in multiple epochs

Let's create an ANN that attempts to classify animals based on three measurements: weight, length, and number of teeth. This is, of course, a mock scenario. Realistically, no one would describe an animal with just these three statistics. However, our intent is to improve our understanding of ANNs before we start applying them to image data.

Compared to the minimal example in the previous section, our animal classification mockup will be more sophisticated in the following ways:

- We will increase the number of neurons in the hidden layer.
- We will use a larger training dataset. For convenience, we will generate this dataset pseudorandomly.
- We will train the ANN in multiple epochs, meaning that we will train and retrain it multiple times with the same dataset each time.



The number of neurons in the hidden layer is an important parameter that needs to be tested in order to optimize the accuracy of any ANN. You will find that a larger hidden layer may improve accuracy up to a point, and then it will overfit, unless you start compensating with an enormous training dataset. Likewise, up to a point, a greater number of epochs may improve accuracy, but too many will result in overfitting.

► Training an ANN classifier in multiple epochs

Let's go through the implementation step by step:

1. First, we import **OpenCV** and **NumPy** as usual. Then, from the Python standard library, we import the **randint** function to generate pseudorandom integers and the **uniform** function to generate pseudorandom floating-point numbers:

```
import cv2
import numpy as np
from random import randint, uniform
```

2. Next, we create and configure the **ANN**. This time, we use a three-neuron **input** layer, a 50-neuron **hidden** layer, and a four-neuron **output** layer, as highlighted in bold in the following code:

```
animals_net=cv2.ml.ANN_MLP_create()
animals_net.setLayerSizes(np.array([3, 50, 4]))
animals_net.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM,0.6,1.0)
animals_net.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP, 0.1, 0.1)
animals_net.setTermCriteria( (cv2.TERM_CRITERIA_MAX_ITER | cv2.TERM_CRITERIA_EPS, 100, 1.0))
```

► Training an ANN classifier in multiple epochs

3. Now, we need some data. We aren't really interested in representing animals accurately; we just require a bunch of records to be used as training data. Thus, we define four functions in order to generate random samples of different classes, along with another four functions to generate the correct classification results for training purposes:

Input arrays : weight, length, teeth

Output arrays : dog, condor, dolphin, dragon

```
def dog_sample():
    return [uniform(10.0, 20.0), uniform(1.0, 1.5), randint(38, 42)]
def dog_class():
    return [1, 0, 0, 0]
def condor_sample():
    return [uniform(3.0, 10.0), randint(3.0, 5.0), 0]
def condor_class():
    return [0, 1, 0, 0]
def dolphin_sample():
    return [uniform(30.0, 190.0), uniform(5.0, 15.0), randint(80, 100)]
def dolphin_class():
    return [0, 0, 1, 0]
def dragon_sample():
    return [uniform(1200.0, 1800.0), uniform(30.0, 40.0), randint(160, 180)]
def dragon_class():
    return [0, 0, 0, 1]
```

► Training an ANN classifier in multiple epochs

4. We also define the following helper function in order to convert a sample and classification into a pair of NumPy arrays:

```
def record(sample, classification):  
    return (np.array([sample], np.float32), np.array([classification], np.float32))
```

5. Let's proceed with the creation of our fake animal data. We will create 20,000 samples per class:

```
RECORDS = 20000  
records = []  
for x in range(0, RECORDS):  
    records.append(record(dog_sample(), dog_class()))  
    records.append(record(condor_sample(), condor_class()))  
    records.append(record(dolphin_sample(), dolphin_class()))  
    records.append(record(dragon_sample(), dragon_class()))
```

► Training an ANN classifier in multiple epochs

6. Now, let's train the ANN. As we discussed at the start of this section, we will use multiple training epochs. Each epoch is an iteration of a loop, as shown in the following code:

```
EPOCHS = 10
for e in range(0, EPOCHS):
    print("epoch: %d" % e)
    for t, c in records:
        data=cv2.ml.TrainData_create(t,cv2.ml.ROW_SAMPLE,c)
        if animals_net.isTrained():
            animals_net.train(data,cv2.ml.ANN_MLP_UPDATE_WEIGHTS|cv2.ml.ANN_MLP_NO_INPUT_SCALE|
                               cv2.ml.ANN_MLP_NO_OUTPUT_SCALE)
        else:
            animals_net.train(data, cv2.ml.ANN_MLP_NO_INPUT_SCALE | cv2.ml.ANN_MLP_NO_OUTPUT_SCALE)
```



For real-world problems with large and diverse training datasets, an ANN can potentially benefit from hundreds of training epochs. For the best results, you may wish to keep training and testing an ANN until you reach convergence, which means that further epochs no longer produce a noticeable improvement in the accuracy of the results.

Note that we must pass the `cv2.ml.ANN_MLP_UPDATE_WEIGHTS` flag to the ANN's `train` function to update the previously trained model rather than training a new model from scratch. This is a critical point to remember whenever you are training a model incrementally, as we are doing here.

► Training an ANN classifier in multiple epochs

7. Having trained our ANN, we should test it. For each class, let's generate 100 new random samples, classify them using the ANN, and keep track of the number of correct classifications:

```
TESTS = 100

dog_results = 0
for x in range(0, TESTS):
    clas = int(animals_net.predict( np.array([dog_sample()]), np.float32))[0])
    print("class: %d" % clas)
    if clas == 0: dog_results += 1
condor_results = 0
for x in range(0, TESTS):
    clas = int(animals_net.predict( np.array([condor_sample()]), np.float32))[0])
    print("class: %d" % clas)
    if clas == 1: condor_results += 1
dolphin_results = 0
for x in range(0, TESTS):
    clas = int(animals_net.predict( np.array([dolphin_sample()]), np.float32))[0])
    print("class: %d" % clas)
    if clas == 2: dolphin_results += 1
dragon_results = 0
for x in range(0, TESTS):
    clas = int(animals_net.predict( np.array([dragon_sample()]), np.float32))[0])
    print("class: %d" % clas)
    if clas == 3: dragon_results += 1
```

► Training an ANN classifier in multiple epochs

8. Finally, let's print the accuracy statistics:

```
print("dog accuracy: %.2f%%" % (100.0 * dog_results / TESTS))  
print("condor accuracy: %.2f%%" % (100.0 * condor_results / TESTS))  
print("dolphin accuracy: %.2f%%" % (100.0 * dolphin_results / TESTS))  
print("dragon accuracy: %.2f%%" % (100.0 * dragon_results / TESTS))
```

When we run the script, the preceding code block should produce the following output:

```
dog accuracy: 100.00%  
condor accuracy: 100.00%  
dolphin accuracy: 100.00%  
dragon accuracy: 100.00%
```

Since we are dealing with random data, the results may vary each time you run the script. Typically, the accuracy should be high or even perfect because we have set up a simple classification problem with non-overlapping ranges of input data. (The range of random weight values for a dog does not overlap with the range for a dragon, and so forth.)

► Training an ANN classifier in multiple epochs

You may wish to take some time to experiment with the following modifications (one at a time) so that you can see how the ANN's accuracy is affected:

- Change the number of training samples by modifying the value of the `RECORDS` variable.
- Change the number of training epochs by modifying the value of the `EPOCHS` variable.
- Make the ranges of input data partially overlapping by editing the parameters of the `uniform` and `randint` function calls in our `dog_sample`, `condor_sample`, `dolphin_sample`, and `dragon_sample` functions.

When you are ready, we will proceed with an example containing **real-life** image data. With this, we will train an ANN to recognize **handwritten digits**.

► Recognizing handwritten digits with an ANN

A **handwritten digit** is any of the 10 Arabic numerals (0 to 9), written manually with a pen or pencil, as opposed to being printed by a machine. The appearance of **handwritten digits** can vary significantly. Different people have different handwriting, and – with the possible exception of a skilled calligrapher – a person does not produce identical digits every time he or she writes. This variability means that the visual recognition of **handwritten digits** is a non-trivial problem for **machine learning**. Indeed, students and researchers in machine learning often test their skills and new algorithms by attempting to train an accurate recognizer for **handwritten digits**. We will approach this challenge in the following manner:

1. Load data from a Python-friendly version of the **MNIST** database. This is a widely used database containing images of **handwritten digits**.
 2. Using the **MNIST** data, train an **ANN** in multiple **epochs**.
 3. Load an image of a sheet of paper with many **handwritten digits** on it.
 4. Based on contour analysis, detect the individual digits on the paper.
 5. Use our **ANN** to classify the detected digits.
 6. Review the results in order to determine the accuracy of our detector and our **ANN**-based classifier.
- ❓ Before we delve into the implementation, let's review some information about the **MNIST** database.

► Understanding the **MNIST** database of handwritten digits

The **MNIST** database (or **M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology database) is publicly available at <http://yann.lecun.com/exdb/mnist/>. The database includes a training set of 60,000 images of **handwritten digits**. Half of these were written by employees of the United States Census Bureau, while the other half were written by high school students in the **United States**. The database also includes a test set of 10,000 images, gathered from the same writers. All the training and test images are in grayscale format, with dimensions of 28 x 28 pixels. The digits are white (or shades of gray) on a black background. For example, here are three of the **MNIST** training samples:



As an alternative to using **MNIST**, you could, of course, build a similar database yourself. This would involve collecting a large number of images of **handwritten digits**, converting the images into grayscale, cropping them so that each image contains a single digit in a standardized position, and scaling the images so that they are all the same size. You would also need to label the images so that a program could read the correct classification for the purpose of training and testing a classifier.

Now that we know something about the **MNIST** database, let's consider what **ANN** parameters are appropriate for this training set.

► Choosing training parameters for the **MNIST** database

Each **MNIST** sample is an image containing **784** pixels (that is, **28 x 28** pixels). Thus, our **ANN's** **input** layer will have **784** nodes. The **output** layer will have **10** nodes because there are 10 classes of digits (**0 to 9**).

We are free to choose the values of other parameters, such as the number of nodes in the **hidden** layer, the number of training samples to use, and the number of training **epochs**. As usual, experimentation can help us find values that offer acceptable training time and accuracy, without overfitting the model to the training data. Based on some experimentation that the authors of this book have done, we will use **60 hidden** nodes, **50,000** training samples, and **10 epochs**. These parameters will be good enough for a preliminary test, keeping the training time down to a few minutes (depending on the processing power of your machine).

► Implementing a module to train the ANN

Training an ANN based on MNIST is something you might want to do in future projects as well. To make our code more reusable, we can write a Python module that is solely dedicated to this training process. Then (in the next section, Implementing the main module), we will import this training module into a main module, where we will implement our demonstration of digit detection and classification.

Let's implement the training module in a file called `digits_ann.py`:

1. To begin, we will import the `gzip` and `pickle` modules from the Python standard library. As usual, we will also import `OpenCV` and `NumPy`:

```
import gzip
import pickle
import cv2
import numpy as np
```

We will use the `gzip` and `pickle` modules to decompress and load the MNIST data from the `mnist.pkl.gz` file. We briefly mentioned this file earlier, in the Understanding the MNIST database of handwritten digits section. It contains the MNIST data in nested tuples, in the following format:

```
( (training_images, training_ids), (test_images, test_ids) )
```

► Implementing a module to train the ANN

2. Let's write the following helper function to decompress and load the contents of `mnist.pkl.gz`:

```
def load_data():  
    mnist = gzip.open('./digits_data/mnist.pkl.gz', 'rb')  
    training_data, test_data = pickle.load(mnist)  
    mnist.close()  
    return (training_data, test_data)
```

Note that in the preceding code, `training_data` is a tuple, equivalent to `(training_images, training_ids)`, and `test_data` is also a tuple, equivalent to `(test_images, test_ids)`.

► Implementing a module to train the ANN

3. We must reformat the raw data in order to match the format that OpenCV expects. Specifically, when we provide sample output to train the ANN, it must be a vector with 10 elements (for 10 classes of digits), rather than a single digit ID. For convenience, we will also apply Python's built-in zip function to reorganize the data in such a way that we can iterate over matching pairs of input and output vectors as tuples. Let's write the following helper function to reformat the data:

```
def wrap_data():  
    tr_d, te_d = load_data()  
    training_inputs = tr_d[0]  
    training_results = [vectorized_result(y for y in tr_d[1])]  
    training_data = zip(training_inputs, training_results)  
    test_data = zip(te_d[0], te_d[1])  
    return (training_data, test_data)
```

► Implementing a module to train the ANN

4. Note that the preceding code calls `load_data` and another helper function, `vectorized_result`. The latter function converts an ID into a classification vector, as follows:

```
def vectorized_result(j):  
    e = np.zeros((10,), np.float32)  
    e[j] = 1.0  
    return e
```

For example, the ID 1 is converted into a NumPy array containing the values `[0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]`. This 10-element array, as you may have guessed, corresponds to the ANN's output layer, and we can use it as a sample of correct output when we train the ANN.

► Implementing a module to train the ANN

5. So far, we have written functions that load and reformat **MNIST** data. Now, let's write a function that will create an untrained **ANN**:

```
def create_ann(hidden_nodes=60):  
    ann=cv2.ml.ANN_MLP_create()  
    ann.setLayerSizes(np.array([784, hidden_nodes, 10]))  
    ann.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, 0.6, 1.0)  
    ann.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP,0.1,0.1)  
    ann.setTermCriteria( (cv2.TERM_CRITERIA_MAX_ITER | cv2.TERM_CRITERIA_EPS, 100, 1.0))  
    return ann
```

Note that we have hardcoded the sizes of the **input** and **output** layers, based on the nature of the **MNIST** data. However, we have allowed the caller of this function to specify the number of nodes in the **hidden** layer.

► Implementing a module to train the ANN

6. Now, we need a training function that allows the caller to specify the number of **MNIST** training samples and the number of **epochs**. Much of the training functionality should be familiar from our previous **ANN** samples, so let's look at the implementation in its entirety and then discuss some details afterward:

```
def train(ann, samples=50000, epochs=10):
    tr, test = wrap_data()
    tr = list(tr)
    for epoch in range(epochs):
        print("Completed %d/%d epochs" % (epoch, epochs))
        counter = 0
        for img in tr:
            if (counter > samples):
                break
            if (counter % 1000 == 0):
                print("Epoch %d: Trained on %d/%d samples" % \
                    (epoch, counter, samples))
                counter += 1
            sample, response = img
            data = cv2.ml.TrainData_create(
                np.array([sample], dtype=np.float32), cv2.ml.ROW_SAMPLE, np.array([response], dtype=np.float32))
            if ann.isTrained():
                ann.train(data, cv2.ml.ANN_MLP_UPDATE_WEIGHTS | cv2.ml.ANN_MLP_NO_INPUT_SCALE |
                    cv2.ml.ANN_MLP_NO_OUTPUT_SCALE)
            else:
                ann.train(data, cv2.ml.ANN_MLP_NO_INPUT_SCALE | cv2.ml.ANN_MLP_NO_OUTPUT_SCALE)
        print("Completed all epochs!")
    return ann, test
```

► Implementing a module to train the ANN

7. Of course, the purpose of a trained ANN is to make predictions, so we will provide the following predict function in order to wrap the ANN's own predict method:

```
def predict(ann, sample):  
    if sample.shape != (784,):  
        if sample.shape != (28, 28):  
            sample = cv2.resize(sample, (28, 28), interpolation=cv2.INTER_LINEAR)  
            sample = sample.reshape(784,)  
        return ann.predict(np.array([sample], dtype=np.float32))
```

This function takes a trained ANN and a sample image; it performs a minimal amount of data sanitization by making sure the sample image is 28 x 28 and by resizing it if it isn't. Then, it flattens the image data into a vector before giving it to the ANN for classification.

► Implementing a module to train the ANN

That's all the ANN-related functionality we will need to support our demo application. However, let's also implement a **test** function that measures a trained ANN's accuracy by classifying a given set of test data, such as the **MNIST** test data. Here is the relevant code:

```
def test(ann, test_data):
    num_tests = 0
    num_correct = 0
    for img in test_data:
        num_tests += 1
        sample, correct_digit_class = img
        digit_class = predict(ann, sample)[0]
        if digit_class == correct_digit_class:
            num_correct += 1
    print('Accuracy: %.2f%%' % (100.0 * num_correct / num_tests))
```

Now, let's take a short detour and write a minimal test that leverages all the preceding code and the **MNIST** dataset. After that, we will proceed to implement the main module of our demo application.

► Implementing a minimal **test** module

Let's make another script, `test_digits_ann.py`, in order to test the functions from our `digits_ann` module. The test script is quite trivial; here it is:

```
from digits_ann import create_ann, train, test

ann, test_data = train(create_ann())
test(ann, test_data)
```

Note that we haven't specified the number of **hidden** nodes, so `create_ann` will use its default parameter value: 60 **hidden** nodes. Similarly, `train` will use its default parameter values: 50,000 samples and 10 epochs.

When we run this script, it should print training and test information similar to the following:

```
Completed 0/10 epochs
Epoch 0: Trained on 0/50000 samples
Epoch 0: Trained on 1000/50000 samples
... [more reports on progress of training] ...
Completed all epochs! Accuracy: 95.39%
```

Here, we can see that the **ANN** achieved 95.39% accuracy when classifying the 10,000 test samples in the **MNIST** dataset. This is an encouraging result, but let's see how well the **ANN** can generalize. Can it accurately classify data from an entirely different source, unrelated to **MNIST**? Our main application, which detects digits from our own image of a sheet of paper, will provide this kind of challenge to the classifier.

► Implementing the **main** module

Our demo's main script takes everything we have learned in this chapter about **ANNs** and **MNIST** and combines it with some of the object detection techniques that we studied in previous chapters. Thus, in many ways, this is a capstone project for us.

Let's implement the main script in a new file called `detect_and_classify_digits.py`:

1. To begin, we will import **OpenCV**, **NumPy**, and our `digits_ann` module:

```
import cv2
import numpy as np
import digits_ann
```

2. Now, let's write a couple of helper functions to analyze and adjust the bounding rectangles of digits and other contours. As we have seen in previous chapters, overlapping detections are a common problem. The following function, called `inside`, will help us determine whether one bounding rectangle is entirely contained inside another:

```
def inside(r1, r2):
    x1, y1, w1, h1 = r1
    x2, y2, w2, h2 = r2
    return (x1 > x2) and (y1 > y2) and (x1+w1 < x2+w2) and (y1+h1 < y2+h2)
```

With the help of the `inside` function, we will be able to easily choose only the outermost bounding rectangle for each digit. This is important because we do not want our detector to miss any extremities of a digit; such a mistake in detection could make the classifier's job impossible. For example, if we detected only the bottom half of a digit, **8**, the classifier might reasonably see this region as a **0**.

► Implementing the **main** module

With the help of the `inside` function, we will be able to easily choose only the outermost bounding rectangle for each digit. This is important because we do not want our detector to miss any extremities of a digit; such a mistake in detection could make the classifier's job impossible. For example, if we detected only the bottom half of a digit, **8**, the classifier might reasonably see this region as a **0**.

To further ensure that the bounding rectangles meet the classifier's needs, we will use another helper function, called `wrap_digit`, to convert a tightly-fitting bounding rectangle into a square with padding around the digit. Remember that the **MNIST** data contains **28 x 28** pixel square images of digits, so we must rescale any region of interest to this size before we attempt to classify it with our **MNIST** trained **ANN**. By using a padded bounding square instead of a tightly-fitting bounding rectangle, we ensure that skinny digits (such as a **1**) and fat digits (such as a **0**) are not stretched differently.

► Implementing the **main** module

3. Let's look at the implementation of `wrap_digit` in multiple stages. First, we modify the rectangle's smaller dimension (be it width or height) so that it equals the larger dimension, and we modify the rectangle's x or y position so that the center remains unchanged:

```
def wrap_digit(rect, img_w, img_h):  
    x, y, w, h = rect  
    x_center = x + w//2  
    y_center = y + h//2  
    if (h > w):  
        w = h  
        x = x_center - (w//2)  
    else:  
        h = w  
        y = y_center - (h//2)
```

4. Next, we add 5-pixel **padding** on all sides:

```
padding = 5  
x -= padding  
y -= padding  
w += 2 * padding  
h += 2 * padding
```

At this point, our modified rectangle could possibly extend outside the image.

► Implementing the **main** module

5. To avoid out of bounds problems, we crop the rectangle so that it lies entirely within the image. This could leave us with non-square rectangles in these edge cases, but this is an acceptable compromise; we would prefer to use a non-square region of interest rather than having to entirely throw out a detected digit just because it is at the edge of the image. Here is the code for bounds-checking and cropping the rectangle:

```
if x < 0:
    x = 0
elif x > img_w:
    x = img_w
if y < 0:
    y = 0
elif y > img_h:
    y = img_h
if x+w > img_w:
    w = img_w - x
if y+h > img_h:
    h = img_h - y
```

6. Finally, we return the modified rectangle's coordinates:
return x, y, w, h

► Implementing the **main** module

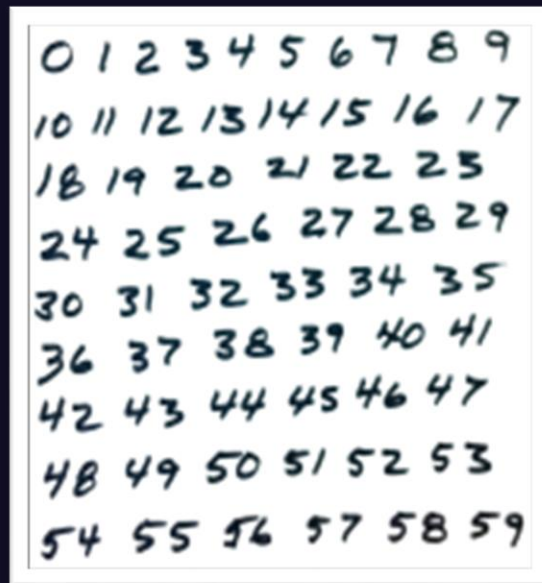
- Now, let's proceed to the main part of the program. Here, we start by creating an **ANN** and training it on **MNIST** data:

```
ann, test_data = digits_ann.train( digits_ann.create_ann(60), 50000, 10)
```

- Now, let's load a test image that contains many **handwritten digits** on a white sheet of paper:

```
img_path = "./digit_images/digits_0.jpg"  
img = cv2.imread(img_path, cv2.IMREAD_COLOR)
```

We are using the following image of Joe Minichino's handwriting (but, ofcourse, you could substitute another image if you prefer):



► Implementing the **main** module

9. Let's convert the image into grayscale and blur it in order to remove noise and make the darkness of the ink more uniform:

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2.GaussianBlur(gray, (7, 7), 0, gray)
```

10. Now that we have a smoothened grayscale image, we can apply a threshold and some morphology operations to ensure that the numbers stand out from the background and that the contours are relatively free of irregularities, which might throw off the prediction. Here is the relevant code:

```
ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY_INV)
erode_kernel = np.ones((2, 2), np.uint8)
thresh = cv2.erode(thresh, erode_kernel, thresh, iterations=2)
```



Note the threshold flag, `cv2.THRESH_BINARY_INV`, which is for an inverse binary threshold. Since the samples in the **MNIST** database are white on black (and not black on white), we turn the image into a black background with white numbers. We use the thresholded image for both detection and classification.

► Implementing the **main** module

11. After the morphology operation, we need to separately detect each digit in the picture. As a step toward this, first, we need to find the contours:

```
contours, hier = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

12. Then, we iterate through the contours and find their bounding rectangles. We discard any rectangles that we deem too large or too small to be digits. We also discard any rectangles that are entirely contained in other rectangles. The remaining rectangles are appended to a list of good rectangles, which (we believe) contain individual digits. Let's look at the following code snippet:

```
rectangles = []
img_h, img_w = img.shape[:2]
img_area = img_w * img_h
for c in contours:
    a = cv2.contourArea(c)
    if a >= 0.98 * img_area or a <= 0.0001 * img_area:
        continue
    r = cv2.boundingRect(c)
    is_inside = False
    for q in rectangles:
        if inside(r, q):
            is_inside = True
            break
    if not is_inside:
        rectangles.append(r)
```

► Implementing the **main** module

13. Now that we have a list of good rectangles, we can iterate through them, sanitize them using our `wrap_digit` function, and classify the image data inside them:

for r in rectangles:

 x, y, w, h = wrap_digit(r, img_w, img_h)

 roi = thresh[y:y+h, x:x+w]

 digit_class = int(digits_ann.predict(ann, roi)[0])

14. Moreover, after classifying each digit, we draw the sanitized bounding rectangle and the classification result:

cv2.rectangle(img, (x,y), (x+w, y+h), (0, 255, 0), 2)

cv2.putText(img, "%d" % digit_class, (x, y-5), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)

► Implementing the **main** module

15. After processing all the regions of interest, we save the thresholded image and the fully annotated image and display them until the user hits any key to end the program:

```
cv2.imwrite("detected_and_classified_digits_thresh.png", thresh)
cv2.imwrite("detected_and_classified_digits.png", img)
cv2.imshow("thresh", thresh)
cv2.imshow("detected and classified digits", img)
cv2.waitKey()
```

That is the end of the script. When running it, we should see the thresholded image as well as a visualization of detection and classification results. (The two windows may overlap initially, so you might need to move one to see the other.) Here is the thresholded image:



► Implementing the **main** module

Here is the visualization of the results:



This image contains 110 sample digits: 10 digits in the single-digit numbers from 0 to 9, plus 100 digits in the double-digit numbers from 10 to 59. Out of these 110 samples, the bounds are correctly detected for 108 samples, meaning that the detector's accuracy is 98.18%. Then, out of these 108 correctly detected samples, the classification result is correct for 80 samples, meaning that the ANN classifier's accuracy is 74.07%. This is a lot better than a random classifier, which would correctly classify a digit only 10% of the time.



► Trying to improve the ANN's training

We could apply a number of potential improvements to the problem of training our ANN. We have already mentioned some of these potential improvements, but let's review them here:

- You could experiment with the size of your training dataset, the number of hidden nodes, and the number of epochs until you find a peak level of accuracy.
- You could modify our `digits_ann.create_ann` function so that it supports more than one hidden layer.
- You could also try different activation functions. We have used `cv2.ml.ANN_MLP_SIGMOID_SYM`, but it isn't the only option; the others include `cv2.ml.ANN_MLP_IDENTITY`, `cv2.ml.ANN_MLP_GAUSSIAN`, `cv2.ml.ANN_MLP_RELU`, and `cv2.ml.ANN_MLP_LEAKYRELU`.
- Similarly, you could try different training methods. We have used `cv2.ml.ANN_MLP_BACKPROP`. The other options include `cv2.ml.ANN_MLP_RPROP` and `cv2.ml.ANN_MLP_ANNEAL`.

Aside from experimenting with parameters, think carefully about your application requirements. For example, where and by whom will your classifier be used? Not everyone draws digits the same way. Indeed, people in different countries tend to draw numbers in slightly different ways. The MNIST database was compiled in the United States, where the digit 7 is handwritten like the typewritten character 7. However, in Europe, the digit 7 is often handwritten with a small horizontal line halfway through the diagonal portion of the number. This stroke was introduced to help distinguish the handwritten digit 7 from the handwritten digit 1.

► Save the trained model

Specifically, you can use code such as the following to save a trained ANN to an XML file:

```
ann = cv2.ml.ANN_MLP_create()
data = cv2.ml.TrainData_create( training_samples, layout, training_responses)
ann.train(data)
ann.save('my_ann.xml')
```

Subsequently, you can reload the trained ANN using code such as the following:

```
ann = cv2.ml.ANN_MLP_create()
ann.load('my_ann.xml')
```

Now that we have learned how to create a reusable ANN for handwritten digit classification, let's think about the use cases for such a classifier.

► Finding other potential applications

The preceding demonstration is only the foundation of a handwriting recognition application. You could readily extend the approach to videos and detect **handwritten digits** in real-time, or you could train your **ANN** to recognize the entire alphabet for a full-blown **optical character recognition (OCR)** system.

Detection and recognition of **car registration plates** would be another useful extension of the lessons we have learned up to this point. The characters on registration plates have a consistent appearance (at least, within a given country), and this should be a simplifying factor in the **OCR** part of the problem.

You could also try applying **ANNs** to problems where we have previously used SVMs, or vice versa. This way, you could see how their accuracy compares for different kinds of data. Recall that in Chapter 7, Building Custom Object Detectors, we used SIFT descriptors as inputs for SVMs. Likewise, **ANNs** are capable of handling high-level descriptors and not just plain old pixel data.

As we have seen, the `cv2.ml_ANN_MLP` class is quite versatile, but in truth, it covers only a small subset of the ways an **ANN** can be designed. Next, we will learn about **OpenCV's** support for more complex **deep neural networks (DNNs)** that can be trained with a variety of other frameworks.

► Using DNNs from other frameworks in OpenCV

OpenCV can load and use DNNs that have been trained in any of the following frameworks:

Caffe (<http://caffe.berkeleyvision.org/>)

TensorFlow (<https://www.tensorflow.org/>)


Torch (<http://torch.ch/>)

Darknet (<https://pjreddie.com/darknet/>)

ONNX (<https://onnx.ai/>)

DLDLT (<https://github.com/opencv/dldt/>)

After we load a model, we need to preprocess the data we will use with the model. The necessary preprocessing is specific to the way the given DNN was designed and trained, so any time we use a third-party DNN, we must read about how that DNN was designed and trained. OpenCV provides a function, `cv2.dnn.blobFromImage`, that can perform some common preprocessing steps, depending on the parameters we pass to it. We can perform other preprocessing steps manually before passing data to this function.

 A neural network's input vector is sometimes called a tensor or blob hence the function's name, `cv2.dnn.blobFromImage`.

Let's proceed to a practical example where we'll see a third-party DNN in action.

► Detecting and classifying objects with third party DNNs

For this demo, we are going to capture frames from a webcam in **real-time** and use a **DNN** to detect and classify **20** kinds of objects that may be in any given frame. Yes, a single **DNN** can do all this in **real-time** on a typical laptop that a programmer might use!

Before delving into the code, let's introduce the **DNN** that we will use. It is a **Caffe** version of a model called **MobileNet-SSD**, which uses a hybrid of a framework from Google called **MobileNet** and another framework called **Single Shot Detector (SSD) MultiBox**.

► Detecting and classifying objects with third party DNNs

1. As usual, we begin by importing **OpenCV** and **NumPy**:

```
import cv2
import numpy as np
```

2. We proceed to load the **Caffe** model with **OpenCV** in the same manner that we described in the previous section:

```
model = cv2.dnn.readNetFromCaffe('objects_data/MobileNetSSD_deploy.prototxt',
                                'objects_data/MobileNetSSD_deploy.caffemodel')
```

3. We need to define some preprocessing parameters that are specific to this model. It expects the input image to be **300** pixels high. Also, it expects the pixel values in the image to be on a scale from **-1.0** to **1.0**. This means that, relative to the usual scale from **0** to **255**, it is necessary to subtract **127.5** and then divide by **127.5**. We define the parameters as follows:

```
blob_height = 300
color_scale = 1.0/127.5
average_color = (127.5, 127.5, 127.5)
```

► Detecting and classifying objects with third party DNNs

4. We also define a confidence threshold, representing the minimum confidence score that we require in order to accept a detection as a real object:

```
confidence_threshold = 0.5
```

5. The model supports 20 classes of objects, with IDs from 1 to 20 (not 0 to 19). The labels for these classes can be defined as follows:

```
labels = ['airplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car', 'cat', 'chair', 'cow', 'dining table', 'dog', 'horse',  
         'motorbike', 'person', 'potted plant', 'sheep', 'sofa', 'train', 'TV or monitor']
```

With the model and parameters at hand, we are ready to start capturing frames.

► Detecting and classifying objects with third party DNNs

6. For each frame, we begin by calculating the aspect ratio. Remember that this DNN expects the input to be based on an image that is 300 pixels high; however, the width can vary in order to match the original aspect ratio. The following code snippet shows how we capture a frame and calculate the appropriate input size:

```
cap = cv2.VideoCapture(0)
success, frame = cap.read()
while success:
    h, w = frame.shape[:2]
    aspect_ratio = w/h
    # Detect objects in the frame.
    blob_width = int(blob_height * aspect_ratio)
    blob_size = (blob_width, blob_height)
```

7. At this point, we can simply use the `cv2.dnn.blobFromImage` function, with several of its optional arguments, to perform the necessary preprocessing, including resizing the frame and converting its pixel data into a scale from -1.0 to 1.0:

```
blob = cv2.dnn.blobFromImage( frame, scalefactor=color_scale, size=blob_size, mean=average_color)
```

► Detecting and classifying objects with third party DNNs

8. We feed the resulting blob to the DNN and get the model's output:

```
model.setInput(blob)
results = model.forward()
```

The results are an array, in a format that is specific to the model we are using.

9. For this object detection DNN – and for other DNNs trained with the SSD framework – the results include a subarray of detected objects, each with its own confidence score, rectangle coordinates, and class ID. The following code shows how to access these, as well as how to use an ID to look up a label in the list we defined earlier:

```
# Iterate over the detected objects.
for object in results[0, 0]:
    confidence = object[2]
    if confidence > confidence_threshold:
        # Get the object's coordinates.
        x0, y0, x1, y1 = (object[3:7] * [w, h, w, h]).astype(int)
        # Get the classification result.
        id = int(object[1]) label = labels[id - 1]
```

► Detecting and classifying objects with third party DNNs

10. As we iterate over the detected objects, we draw the detection rectangles, along with the classification labels and confidence scores:

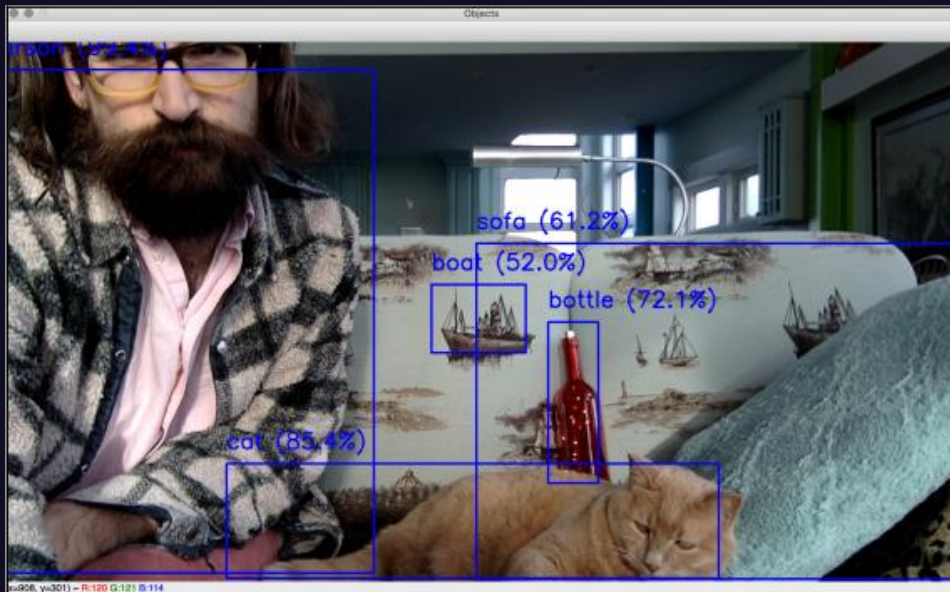
```
# Draw a blue rectangle around the
object. cv2.rectangle(frame, (x0, y0), (x1, y1), (255, 0, 0), 2)
# Draw the classification result and confidence.
text = '%s (%.1f%%)' % (label, confidence * 100.0)
cv2.putText(frame, text, (x0, y0 - 20), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
```

11. The last thing we do with the frame is show it. Then, if the user has hit the **Esc** key, we exit; otherwise, we capture another frame and continue to the next iteration of the loop:

```
cv2.imshow('Objects', frame)
k = cv2.waitKey(1)
if k == 27: # Escape
    Break
success, frame = cap.read()
```

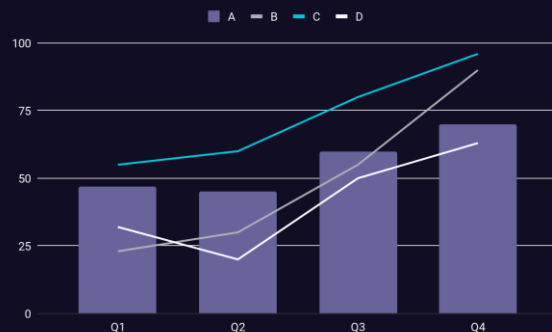
► Detecting and classifying objects with third party DNNs

If you plug in a webcam and run the script, you should see a visualization of detection and classification results, updated in **real-time**. Here is a screenshot showing Joseph Howse and Sanibel Delphinium Andromeda (a mighty, great, and righteous cat) in their living room in a Canadian fishing village:



► Detecting and classifying objects with third party DNNs

The DNN has correctly detected and classified a human person (with 99.4% confidence), a cat (85.4%), a decorative bottle (72.1%), part of a sofa (61.2%), and a woven picture of a boat (52.0%). Evidently, this DNN is well equipped to classify the contents of living rooms in nautical settings! This is only a first taste of the things that DNNs can do – and do in real time! Next, let's see what we can achieve by combining three DNNs in one application.





► **THANKS!**

Do you have any questions?

MrMrProgrammer.ir

mr.mr.programmer@gmail.com

