In The name of God the Merciful

Introduction to Neural Networks with OpenCV

Chapter 10

▶ 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:

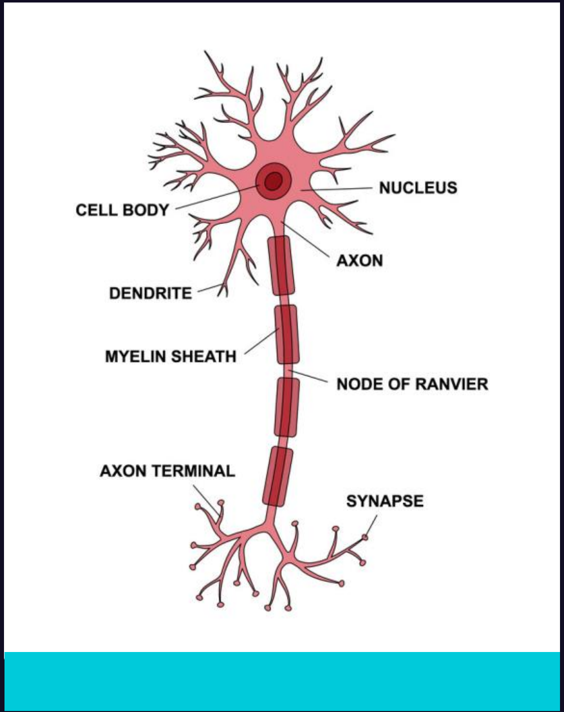
- Understanding ANNs as a statistical model and as a tool for supervised machine learning.
- Understanding ANN topology or, in other words, the organization of an ANN into layers of interconnected neurons. Particularly, we will consider the topology that enables an ANN to act as a type of classifier 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.



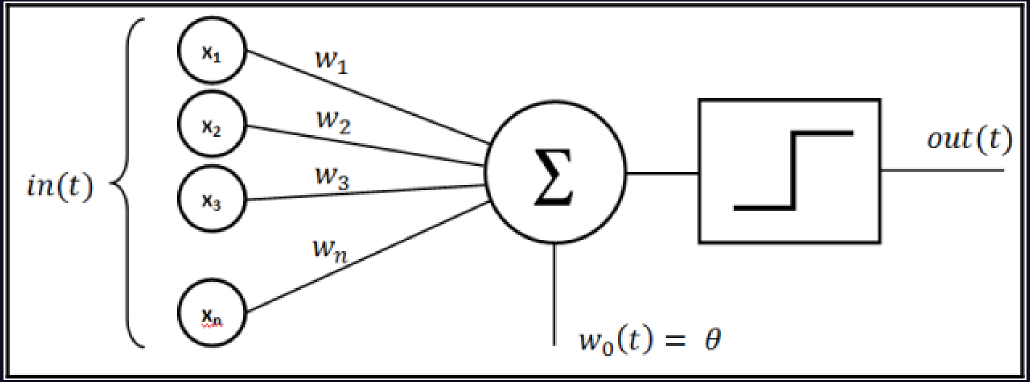
ANNs, like other types of machine learning models, can learn from observations in one of the following ways:

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 neurons and perceptrons

What do the input weights represent, and how are they determined?

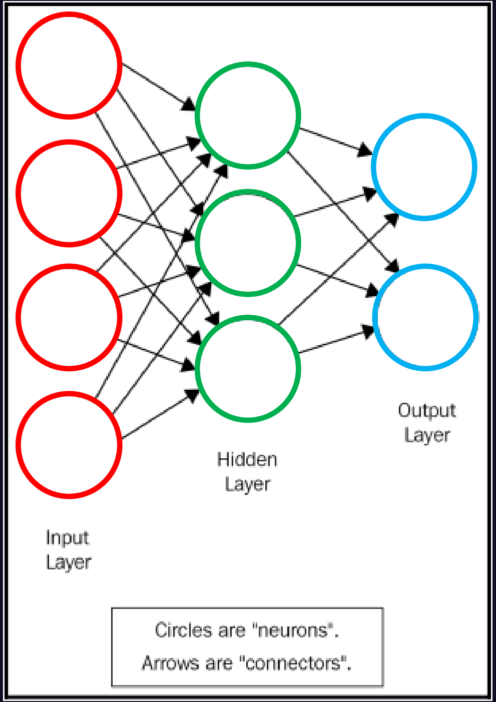
Neurons are interconnected, insofar as one neuron's output can be an input for many other neurons. Each input weight defines the strength of the connection between two neurons. These weights are adaptive, meaning that they change in time according to a learning algorithm.

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):

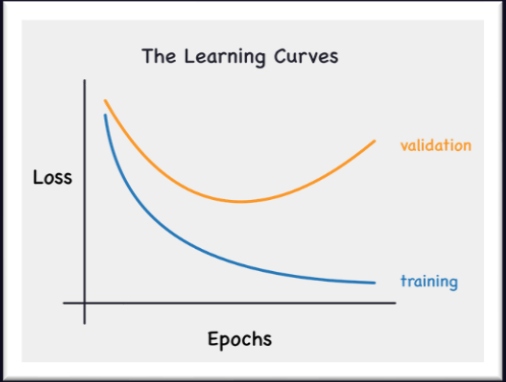
- If the input layer is large, the number of hidden neurons should be between the size of the input layer and the size of the output layer – and, typically, closer to the size of the output layer.
- On the other hand, if the input and output layers are both small, the hidden layer should be the largest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

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

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

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 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.