Chapter 6
Perceptron and Neural Networks

6.1 Introduction

For most people, the area of machine learning and artificial intelligence starts and ends with neural networks. From the name itself, it suggests an implicit resemblance with human brain and the neurons inside it. Common sense dictates: as the biological neurons are responsible for the human intelligence, these artificial neurons must be the building blocks of artificial intelligence. Incidently, in this case, the common sense is not far from the truth. The invention of such an artificial neuron (technically called as perceptron in the literature) indeed marked the beginning of modern neural networks.

6.1.1 Biological Neuron

A biological neuron looks like the illustration shown in Fig. 6.1. It is composed of three main parts that are unique to it along with other typical cellular objects:

1. Soma: This is the nucleus of the neuron.
2. Dendrites: They exhibit a complex treelike structure with branches ranging up to tens of thousands. These are responsible for receiving messages from other neurons.
3. Axon: This is a long stemlike structure that carries the incoming signals from the dendrites to send it to other neurons that may be physically farther away. It ends in axon terminals that look similar to dendrites.

The activation of neurons and the transmission of the messages to and from neurons are controlled by electrochemical processes. Each neuron is continuously getting activation signals from the other connecting neurons. The information is

A. V. Joshi, *Machine Learning and Artificial Intelligence*,
https://doi.org/10.1007/978-3-031-12282-8_6

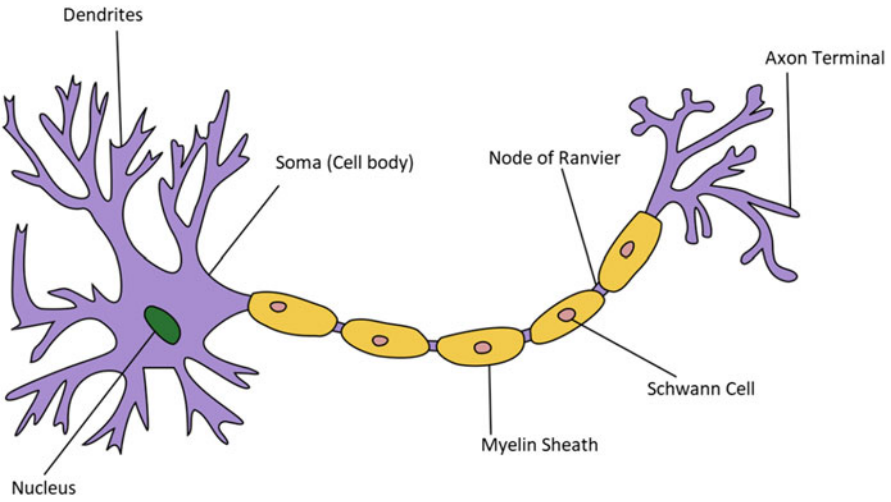


Fig. 6.1 Biological neuron [25]

gathered in the neuron's cell body where it gets processed and then sent out to axon terminals.

6.2 Perceptron

Perceptron was introduced by Rosenblatt [64] in 1957 with an architecture that was strongly influenced by that of a biological neuron. It was used to craft a generalized computation framework for solving linear problems. It was quite effective, one of a kind machine at the time and seemingly had unlimited potential. In the early days after its successful implementation to solve certain types of problems, some critical flaws were detected in the theory of perceptron that limited their scope significantly (e.g., problem of implementing XOR operation). However, these issues were overcome in time with addition of multiple layers in the architecture of the perceptron giving rise to the concept of artificial neural networks (ANNs). The addition of nonlinear kernel functions like sigmoid further extended the scope of ANNs. We will study the concept of perceptron and its evolution into modern ANNs in this chapter. However, we will restrict the scope to traditional neural networks and will not delve into the deep networks. We will study those in Chap. 13.

Geometrically a single-layer perceptron with linear mapping represents a linear plane in n-dimensions. In n-dimensional space, the input vector is represented as $(x_1, x_2, \ldots, x_n)$ or $\mathbf{x}$. The coefficients or weights in n-dimensions are represented as $(w_1, w_2, \ldots, w_n)$ or $\mathbf{w}$. The equation of perceptron in the n-dimensions is then written in vector form as

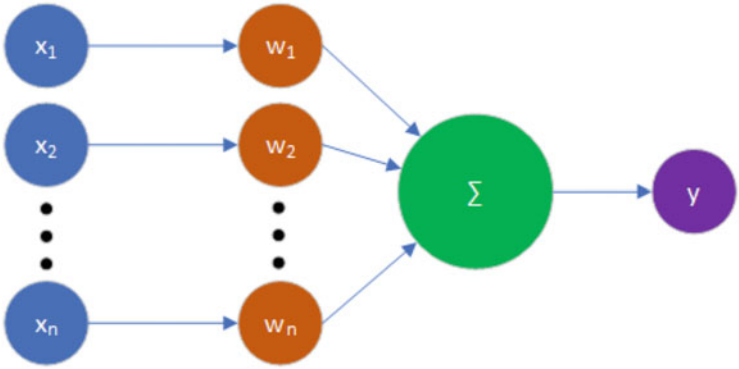


Fig. 6.2 Perceptron

$$\mathbf{x}.\mathbf{w} = y \tag{6.1}$$

Figure 6.2 shows an example of a n-dimensional perceptron. This equation looks a lot like the linear regression equation that we studied in Chap. 5, which is essentially true, as perceptron represents a similar computational architecture to solve this problem. However, the differences lie in the way the weights are computed or learned.

6.2.1 Implementing Perceptron

Let's use the *sklearn* library and *Google Colab* as explained in the earlier chapter to implement the perceptron. As perceptron architecture is primarily used for classification, let's use the *Iris* dataset to illustrate the use of perceptron. Iris dataset contains 150 samples corresponding to three types of flowers. Each type of flower has 50 samples. There are $four$ features for each samples making the data four dimensional.

Figure 6.3 shows how we can import the perceptron model from sklearn and train it on Iris data. In Fig. 6.4 we apply the trained model on the same data to see how well it is trained. As can be seen, the overall prediction accuracy of this classifier is only at 0.48. As the original data is not linearly separable, the accuracy of simple perceptron is not great.

Ideally, we should use separate datasets for training and predicting, but we are using the same dataset here for the purpose of illustrating how well the model is learning the patterns and also to emphasize the fact that the model is not just remembering the training patterns showing 100% accuracy on them. We will also use these scores as reference to compare with other models that we will learn in the chapter.

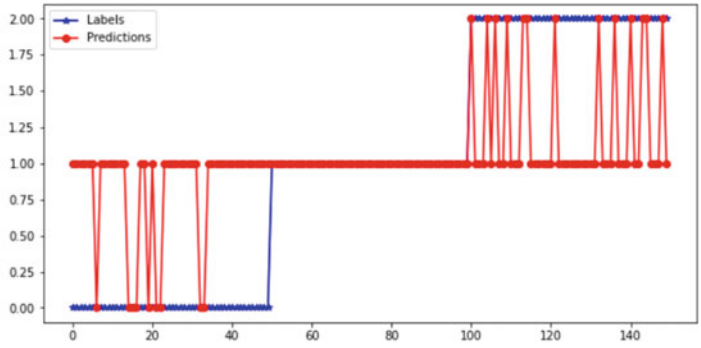
```
from sklearn import datasets
from sklearn.linear_model import Perceptron
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

```
[2] iris.feature_names, iris.target_names
```

```
(['sepal length (cm)',
  'sepal width (cm)',
  'petal length (cm)',
  'petal width (cm)'],
 array(['setosa', 'versicolor', 'virginica'], dtype='<U10'))
```

Fig. 6.3 Implementing perceptron using Google Colab and sklearn

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax1 = fig.add_subplot(111)

ax1.plot(y, c='b', marker="*", label='Labels')
ax1.plot(y_pred, c='r', marker="o", label='Predictions')
plt.legend(loc='upper left');
fig.set_size_inches(10,5)
plt.show()
```




```
[6] ptron.score(X,y)
```

```
0.48
```

Fig. 6.4 Visualizing the training performance with perceptron on Iris data

If there are multiple variables to be predicted using the same set of inputs, one can have a series of perceptrons in parallel to generate those outputs. This architecture is called as single-layer perceptron.

6.3 Multilayered Perceptron or Artificial Neural Network

Multilayered perceptron or (*MLP*) is a logical extension of the single-layer architecture, again following the functioning of biological neural network, where there are multiple layers of perceptrons chained in series. Each layer can contain an arbitrary number of nodes or perceptrons as needed. Figure 6.1 shows an illustration of a generic MLP with 3 layers. Let $n1$ be the number of nodes in layer 1, which is the same as the input dimensionality. The subsequent layers have $n2$ and $n3$ number of nodes. The layers of nodes between the input and output layers are called as hidden layers, as their size is independent of the input and output, and are not directly visible from that perspective. The number of nodes in hidden layers can have any arbitrary value. Typically the more complex the relation between input and output, the more and bigger hidden layers are used. Also, all the consecutive layers in MLP are fully connected, meaning each of the internal layers needs to be fully connected to all the nodes in the previous and next layer. It is important to note that as long as we are using linear mapping (also called as activation function) at each node, single-layer perceptron and multilayered perceptron are mathematically equivalent. In other words, having multiple layers does not really improve the capabilities of the model, and it can be proved mathematically using rigorous calculations. Thus, the real benefits of MLP architecture start to surface with the use of nonlinear activation functions.

6.3.1 Feedforward Operation

The network shown in Fig. 6.5 also emphasizes another important aspect of MLP called as feedforward operation. The information that is entered from the input propagates through each layer towards the output. There is no feedback of

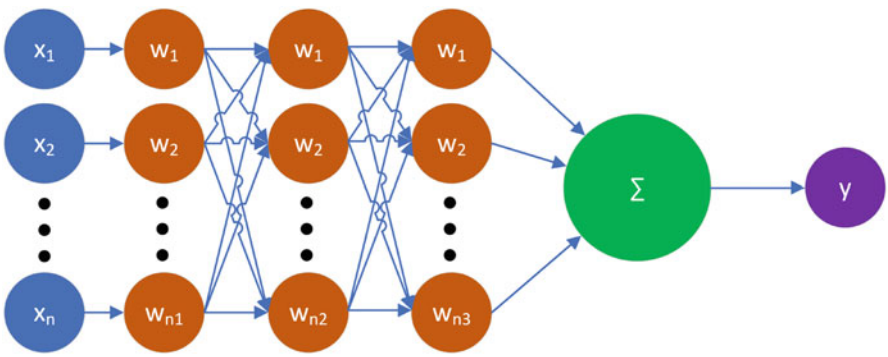


Fig. 6.5 Multilayered perceptron

information from any layer backwards when the network is used for predicting the output in the form of regression or classification. This process also closely resembles the operation of human brain.

6.3.2 Nonlinear MLP or Nonlinear ANN

The major improvement in the MLP architecture comes in the way of using nonlinear mapping. Instead of using simple dot product of the input and weights, a nonlinear function, called as activation function, is used.

6.3.2.1 Activation Functions

The most simple activation function is a step function, also called as a *sign* function, as shown in Fig. 6.6. This activation function is suited for applications like binary classification. However, as this is not a continuous function, it is not suitable for most training algorithms as we will see in the next section.

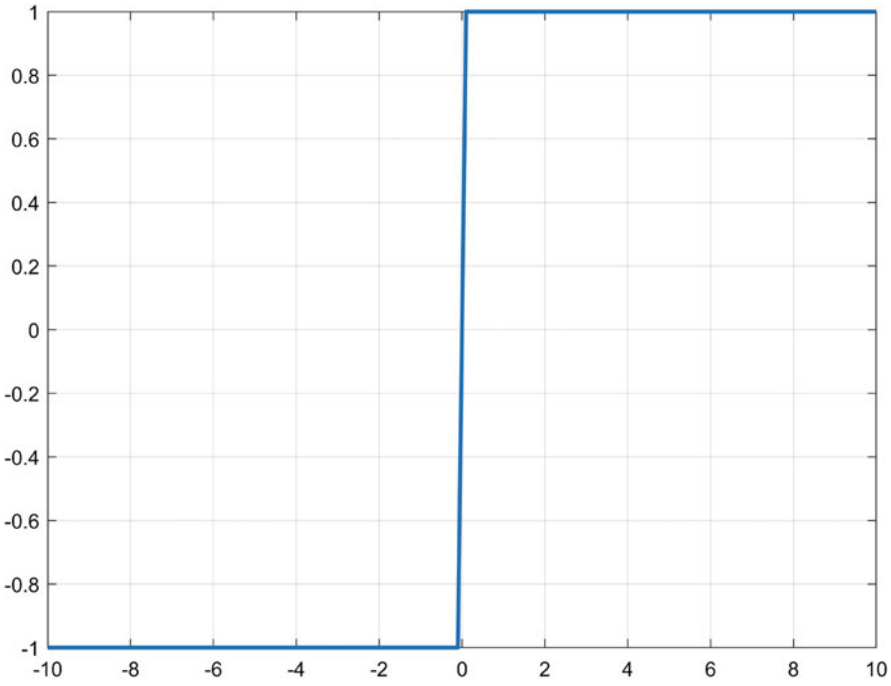


Fig. 6.6 Activation function *sign*

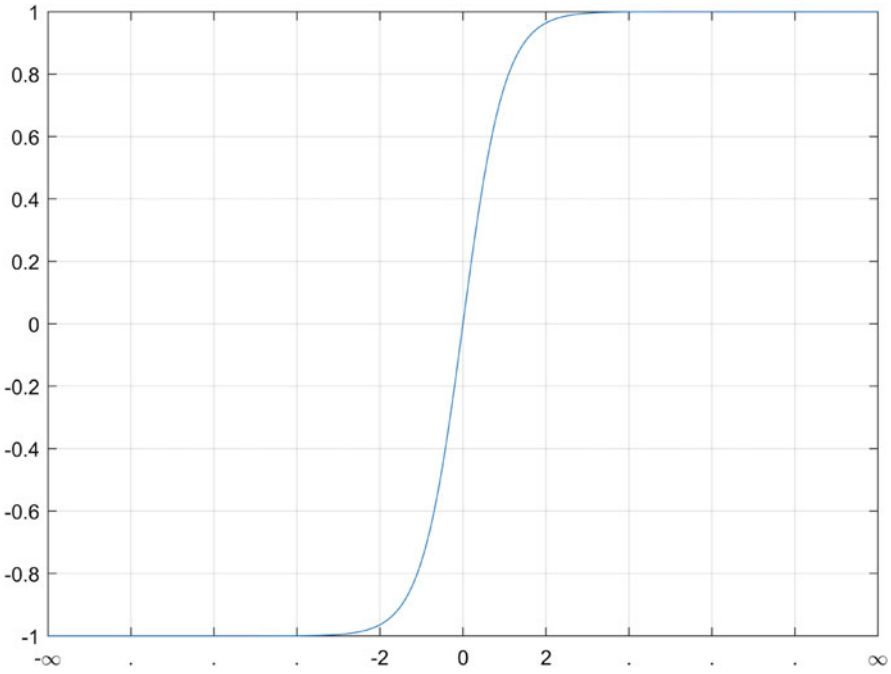


Fig. 6.7 Activation function *tanh*

The continuous version of step function is called as a hyperbolic tan or *tanh* function as shown in Fig. 6.7. Sometimes a variation of *tanh* function called as sigmoid function is used. Sigmoid function has exactly the same shape, but its value ranges from 0 to 1, instead of from -1 to 1 in the case of *tanh* function.

The relationship between input $\mathbf{x}$ and output Y at a given node that uses sigmoid activation function $S(x)$ can be written as

$$S(\mathbf{x}.\mathbf{w}) = y \tag{6.2}$$

where w represents weights for each input coming to the given node. This mapping ensures that the value of y will always be bounded between 0 and 1 irrespective of the values of inputs and the weights.

With the use of such nonlinear activation functions, MLP architecture is no more equivalent to the single layer and can now deal with more complex and nonlinear input-output mappings.

6.3.3 Training MLP

During the training process, the weights of the network are learned from the labelled training data using the process of backpropagation. Conceptually the process can be described as

1. Present the input to the neural network.
2. All the weights of the network are assigned some default value.
3. The input is transformed into output by passing through each node or neuron in each layer.
4. The output generated by the network is then compared with the expected output or label.
5. The error between the prediction and label is then used to update the weights of each node.
6. The error is then propagated in backward direction through every layer, to update the weights in each layer such that they minimize the error.

Reference [65] summarizes various backpropagation training algorithms commonly used in the literature along with their relative performances. We are not going to go into the mathematical details of these algorithms here, as the theory quickly becomes quite advanced and can make the topic very hard to understand. Also, as we look at the implementation of this algorithm using sklearn, we will see that with conceptual understanding of the training framework, one is sufficiently armed to apply these concepts to solve real problems.

Thus, backpropagation algorithm for training and feedforward operation for prediction mark the two phases in the operation of neural network. Backpropagation-based training can be done in two different methods.

1. Online or stochastic method
2. Batch method

6.3.3.1 Online or Stochastic Learning

In this method a single sample or a small subset of an entire training set (drawn randomly) is sent as input to the network, and based on the output error, the weights are updated. The optimization method most commonly used to update the weights in this setup is called stochastic gradient descent or *SGD* method. The use of stochastic here implies that the samples are drawn randomly from the whole dataset rather than using them sequentially. The process can converge to the desired accuracy level even before all the samples are used. It is important to understand that in stochastic learning process, a small set of samples is used in each iteration, and the learning path is more noisy. The set of samples used in each iteration is called a mini-batch. SGD is beneficial when the expected learning path can contain multiple local minima and/or the size of training data is too large that using all the data in each iteration is not feasible.

6.3.3.2 Batch Learning

In batch method the entire training dataset is used and divided into small and deterministic set of batches (unlike stochastic method where samples are drawn randomly). The entire batch of samples is sent to the network before computing the error and updating the weights. After an entire batch is processed, the weights are updated. The training process using each batch is called as one iteration. When all the samples are used once, it is considered as one epoch in the training process. Typically multiple epochs are used before the algorithm fully converges. As the batch learning uses a batch of samples in each iteration, it reduces the overall noise, and the learning path is cleaner. However, the process is a lot more computation heavy and needs more memory and computation resources. Batch learning is preferred when computer hardware permits, and the learning path is expected to be relatively smooth.

6.3.4 Hidden Layers

The concept of hidden layers needs a little more explanation. As such they are not directly connected with inputs and outputs, and there is no theory around how many such layers are optimal in a given application. Each layer in MLP transforms the input to a new dimensional space. The hidden layers can have a higher dimensionality than the actual input, and thus they can transform the input into an even higher dimensional space. Sometimes, if the distribution of input in its original space has some nonlinearities and is ill conditioned, the higher dimensional space can help improve the distribution and as a result improve the overall performance. These transformations also depend on the activation function used. Increasing the dimensionality of hidden layer also makes the training process much more complicated, and one needs to carefully trade between the added complexity and performance improvement. Also, how many such hidden layers should be used is another variable where there are no theoretical guidelines. Both these parameters are called as hyperparameters, and one needs to do an open-ended exploration using a grid of possible values for them and then choose the combination that gives the best possible results within the constraints of the training resources.

6.3.5 Implementing MLP

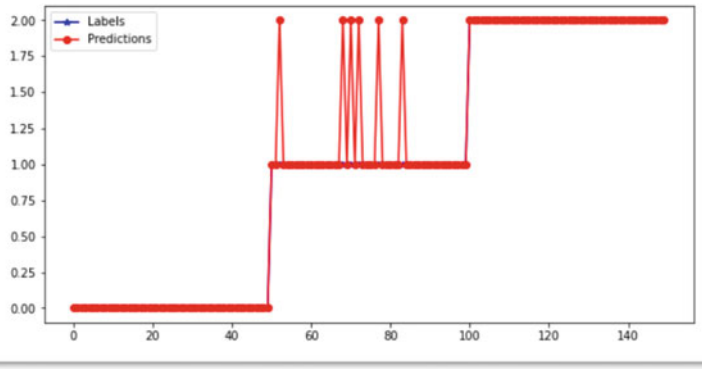
We revisit the problem of classifying the Iris data, now with MLP architecture with logistic activation function and a single hidden layer with 50 nodes (the number 50 is selected as fairly arbitrary) (Fig. 6.8). We use all the other default parameters to train the model using sklearn library. We then apply on the same training data as before to see how the algorithm is able to model the data (Fig. 6.9).

```
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(activation='logistic', hidden_layer_sizes=(50,))
mlp.fit(X,y)
y_pred = mlp.predict(X)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:69
  ConvergenceWarning,
```

Fig. 6.8 Implementing MLP using Google Colab and sklearn

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax1 = fig.add_subplot(111)

ax1.plot(y, c='b', marker="*", label='Labels')
ax1.plot(y_pred, c='r', marker="o", label='Predictions')
plt.legend(loc='upper left');
fig.set_size_inches(10,5)
plt.show()
```




```
[21] mlp.score(X,y)

0.96
```

Fig. 6.9 Visualizing the training performance with MLP on Iris data

As you can see, the classification accuracy is order of magnitude better compared to a simple perceptron using linear mappings.

6.4 Radial Basis Function Networks

Radial basis function networks *RBFN* or radial basis function neural networks *RBFNN* are a variation of the feedforward neural networks (we will call them as RBF networks to avoid confusion). Although their architecture as shown in

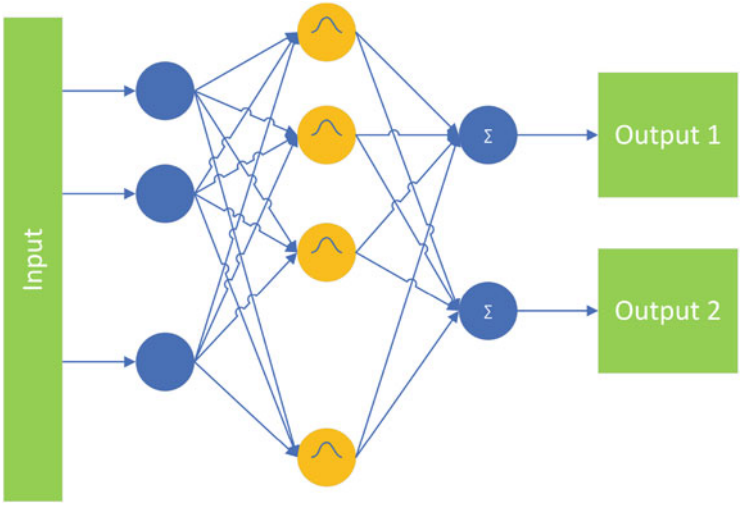


Fig. 6.10 Architecture of radial basis function neural network

Fig. 6.10 looks similar to MLP as described above, functionally they are more close to the support vector machines with radial kernel functions. The RBF networks are characterized by three layers: an input layer, a single hidden layer, and an output layer. The input and output layers are linear weighing functions, and the hidden layer has a radial basis activation function instead of sigmoid-type activation function that is used in traditional MLP. The basis function is defined as

$$f_{RBF}(x) = e^{-\beta\|x-\mu\|^2)} \tag{6.3}$$

The above equation is defined for a scalar input, but without lack of generality, it can be extended for multivariate inputs. μ is called as center, and β represents the spread or variance of the radial basis function. It lies in the input space. Figure 6.11 shows the plot of the basis function. This plot is similar to Gaussian distribution.

6.4.1 Interpretation of RBF Networks

Aside from the mathematical definition, RBF networks have a very interesting interpretability that regular MLP does not have. Consider that the desired values of output form n number of clusters for the corresponding clusters in the input space. Each node in the hidden layer can be thought of as a representative of each transformation from the input cluster to the output cluster. As can be seen from Fig. 6.11, the value of radial basis function reduces to 0 rather quickly as the distance between the input and the center of the radial basis function μ increases with respect

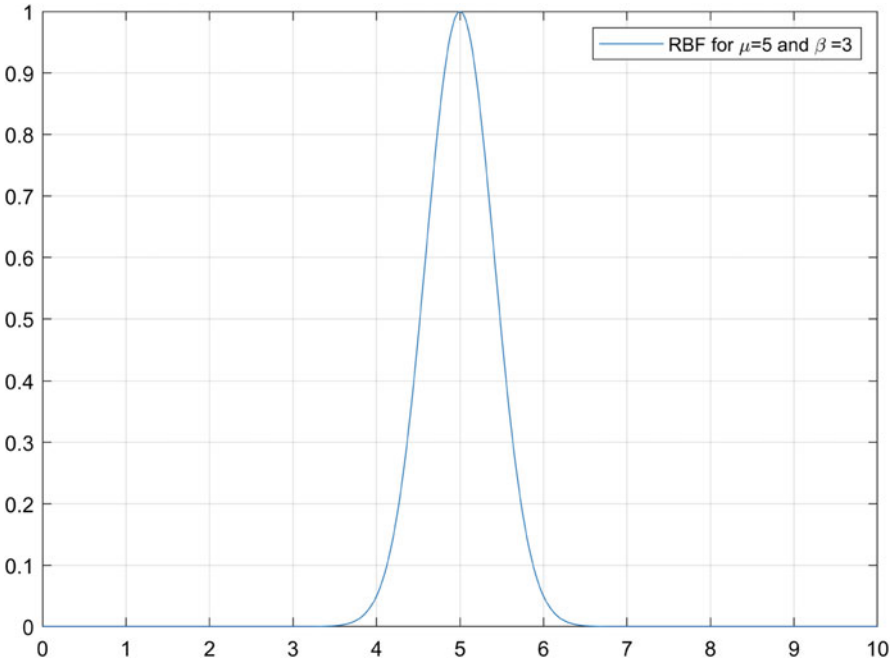


Fig. 6.11 Plot of radial basis function

to the spread β. Thus, RBF network as a whole maps the input space to the output space by linear combination of outputs generated by each hidden RBF node. It is important to choose these cluster centers carefully to make sure the input space is mapped uniformly and there are no gaps. The training algorithm is capable of finding the optimal centers, but the number of clusters to use is a hyperparameter (in other words it needs to be tuned by exploration). If an input is presented to RBF network that is significantly different than the one used in training, the output of the network can be quite arbitrary. In other words the generalization performance of RBF networks in extrapolation situations is not good. However, if requirements for the RBF network are followed, it produces accurate predictions.

6.4.2 Implementing RBF Networks

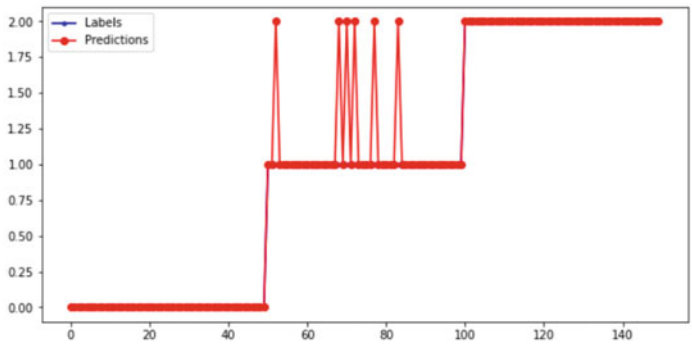
We can implement RBF network with sklearn using *GaussianProcessClassifier* and selecting the kernel as *RBF*. Figure 6.12 shows the code for implementation. The following figures show the code for implementation. We will use all the default parameters for the model to focus on the concept. As can be seen, the accuracy is better than MLP, but marginally, and with some tuning both models can essentially produce similar results (Fig. 6.13).

```
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF

kernel = 1.0 * RBF(1.0)
rbfn = GaussianProcessClassifier(kernel=kernel, random_state=0)
rbfn.fit(X, y)
y_pred = rbfn.predict(X)
```

```
GaussianProcessClassifier(kernel=1**2 * RBF(length_scale=1), random_state=0)
```

Fig. 6.12 Implementing RBF network using Google Colab and sklearn

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax1 = fig.add_subplot(111)

ax1.plot(y, c='b', marker=".", label='Labels')
ax1.plot(y_pred, c='r', marker="o", label='Predictions')
plt.legend(loc='upper left');
fig.set_size_inches(10,5)
plt.show()
```




```
rbfn.score(X,y)
```

```
0.9866666666666667
```

Fig. 6.13 Visualizing the training performance with RBF network on Iris data

6.5 Overfitting

Neural networks open up a feature-rich framework with practically unlimited scope to improve the performance for the given training data by increasing the complexity of the network. Complexity can be increased by manipulating various factors as follows:

1. Increasing the number of hidden layers
2. Increasing the nodes in hidden layers
3. Using complex activation functions

4. Increasing the training epochs

Such improvements in training performance with arbitrary increase in complexity typically lead to overfitting. Overfitting is a phenomenon where we try to model the training data so accurately that in essence we just memorize the training data rather than identifying the generic patterns and structure in it. Such memorization leads to significantly worse performance on unseen data. However, determining the optimal threshold where the optimization should be stopped to keep the model generic enough but also accurate is not trivial. Multiple approaches are proposed in the literature, e.g., *Optimal Brain Damage* [67] or *Optimal Brain Surgeon* [66]. Before delving into the regularization techniques specifically tailored for neural networks, let's look at the concept of regularization in general.

6.5.1 Concept of Regularization

When we are dealing with any prediction problem, we have a loss function that we want to optimize using labelled training data. This optimization process leads to finding the values of the parameters of the equation and concludes the training. If we start with overly complex functional, the process can lead to what we just defined as memorization. In order to restrict the memorization, we can use a mathematical trick, by altering the optimization functional with some added terms. These added factors limit the range of values that the parameters of the algorithm can take and ultimately ensure that the memorization is avoided.

6.5.2 L1 and L2 Regularization

$$C(x) = L(x) + \lambda \sum \|\mathbf{W}\| \tag{6.4}$$

$$C(x) = L(x) + \lambda \sum \|\mathbf{W}\|^2 \tag{6.5}$$

The process of regularization that alters the optimization function (or loss function), also called as technique of Lagrangian multipliers, typically adds another term in the optimization problem that restricts the complexity of the network and is weighted by Lagrangian weighing factor λ. Equations 6.4 and 6.5 show the updated cost function $C(x)$ use of L1 and L2 types of regularizations to reduce the overfitting.

$L(x)$ is the loss function that is dependent on the error in prediction, while $\mathbf{W}$ stands for the vector of weights in the neural network. The L1 norm tries to minimize the sum of absolute values of the weights, while the L2 norm tries to minimize the sum of squared values of the weights. Each type has some pros and cons. The L1 regularization requires less computation but is less sensitive

to strong outliers, as well as prone to making the weights zero. In other words, L1 regularization tends to reduce the overall dimensionality of the problem by dropping some weights altogether. L2 regularization is typically overall a better metric and provides slow weight decay towards zero, but is more computation intensive. However, depending on the problem at hand, either one can be a better choice. Linear methods (regression/classification) that use L1 regularization are also called as Lasso methods, and the ones that use L2 regularization are also called as Ridge methods.

6.5.3 Dropout Regularization

This is an interesting method and is only applicable to the case of neural networks, while the L1 and L2 regularization can be applied to any algorithm. In dropout regularization, the neural network is considered as an ensemble of neurons in sequence, and instead of using a fully populated neural network, some neurons are randomly dropped from the path. The effect of each dropout on overall accuracy is considered, and after some iterations, an optimal set of neurons is selected in the final models. As this technique actually makes the model simpler rather than adding more complexity like L1 and L2 regularization techniques, this method is quite popular, specifically in the case of more complex and deep neural networks that we will study in later chapters.

6.6 Conclusion

In this chapter, we studied the machine learning model based on simple neural network. We studied the concept of single perceptron and its evolution into full-fledged neural network. We also studied the variation of the neural networks using radial basis function kernels. In the end we studied the effect of overfitting and how to reduce it using regularization techniques.

6.7 Exercises

1. Play with the parameters of MLP, e.g., the number of hidden layers, the number of nodes in each hidden layer, and activation function, and see the effect of the changes on the accuracy.
2. Separate the training and test data using *sklearn.model_selection.train_test_split* function. Use 70% training and 30% test, and repeat MLP training to see how the test accuracy compares with the earlier method of using the same data for training and test for MLP.

3. Play with all the parameters for RBF Network, and see the impact on the accuracy. Split the training and test data as before, and see the effect of that.
4. After fine-tuning all the models, see which model gives the absolute best accuracy, and also compare which model is easier to tune.
5. Ridge and Lasso regression models are offered in sklearn in *sklearn.linear_model*. Try these two alternatives for perceptron in classification of Iris data, and compare the results.