Chapter 7

SHALLOW ARTIFICIAL NEURAL NETWORK

CONTENTS

7.1 Perceptrons

The psychologist, Frank Rosenblatt (1958), was first inspired by the mechanism of the neurons in our brain to process information and then invented the very first version of Perceptron; it is one of the earliest linear binary classifiers, and it is generally regarded as ancestor of recently developing Neural Networks and Deep Learning. In layman term, each layer of neuron is connected to other layer of neurons, where information is passed from one layer to the latter. For the connection of one neuron to another, variation in the strength of impulse sent from the first to the latter, as neuroscientists believe, it is crucial for our brain to process information; while in the abstract perceptron model, the strength of connection can be modelled by different coefficient weights assigned. After the neuron sums up the information received from the few attached neurons, if the total sum reaches a certain threshold, the neuron fires to the next neuron with output value 1; and with output value 0 otherwise.

Rosenblatt also proposed that perceptron could perform visual tasks such as facial and object recognition. For instance, in recognizing the handwritten digit as introduced in Subsection **??**, a perceptron model can be used to differentiate whether a given handwritten digit is a certain number, with output value 1, or 0

otherwise.

Later, Minsky and Papert (1969) criticized the effectiveness of Perceptron by showing that it could only solve a limited types of question perfectly and gave their mathematical evidence through a book as *Perceptrons: an introduction to computational geometry*. For instance, perceptron is not able to handle the XOR (eXclusive OR) logic function, which receives two binary inputs, see Figure 7.1.1; in this figure the yellow points represent the situation that exactly either one takes value 0 and the another one takes value 1, while the red points represent the situation that both take value 0 or both take value 1; see Programme 7.1.3. Minsky and Papert suggested that this problem can be addressed by adding an additional "layer" of neurons such that the resulting multilayer neural network can deal with arbitrary Boolean function. However, tuning a suitable multilayer neural network model was not broadly studied during that time.

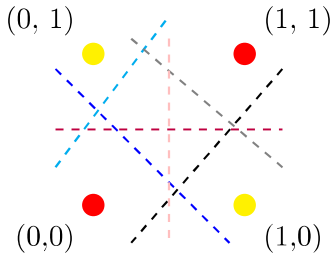


Figure 7.1.1: 2 yellow points and 2 red points which cannot be grouped by any seperating line.

In July 1971, Rosenblatt died in a boating accident in Chesapeake Bay in the United States at the age 43, together with the lack of government funding for neural network research in the late 1960s, the research on perceptrons and other neural networks were halted until 1976.

We first lay down the feature setting, so that $\boldsymbol{x}_n = (x_n^{(0)}, x_n^{(1)}, \cdots, x_n^{(D)}) \in \mathbb{R}^{D+1}$, where $x_n^{(0)} = 1$ and $(x_n^{(1)}, \cdots, x_n^{(D)}) \in \mathbb{R}^D$ is a D-dimensional feature vector, and $y_n = 1$ or -1 corresponding to its label value. For a better illustration of the mechanism behind perceptron, we use $+1$ and -1 to be the output values where the neuron is fired or not respectively, instead of $+1$ and 0. Perceptron is to learn a linear function in $\boldsymbol{x}_n$, f, which classifies a given datum $\boldsymbol{x}_n$ as the category 1 (predicted label, $\hat{y}_n = 1$) if $f(\boldsymbol{x}_n) > 0$; the category -1 ($\hat{y}_n = -1$) or otherwise if $f(\boldsymbol{x}_n) < 0$; further if $f(\boldsymbol{x}_n) = 0$, we consider it as undetermined, which almost surely barely happens in practice. That means, the learner is $f(\boldsymbol{x}) = \boldsymbol{\omega}^\top \boldsymbol{x}$, where $\boldsymbol{\omega}$ is a vector of parameters to be estimated. Then, a datum $\boldsymbol{x}_n$ is correctly classified if $f(\boldsymbol{x}_n) > 0$ with $y_n = 1$ or $f(\boldsymbol{x}_n) < 0$ with $y_n = -1$; then, it is always the case that $f(\boldsymbol{x}_n) \cdot y_n > 0$, and it is almost surely with a strict inequality. Similarly, a datum $\boldsymbol{x}_n$ is misclassified if $f(\boldsymbol{x}_n) \cdot y_n \leq 0$.

In the following, we illustrate with a planar example, in which a perceptron for two-dimensional feature vectors with a sample $\{(x_n^{(1)}, x_n^{(2)}; y_n)\}_{n=1}^N$ by learning a function

$$f(\boldsymbol{x}) = \boldsymbol{\omega}^\top \boldsymbol{x} = \omega_0 + \omega_1 x^{(1)} + \omega_2 x^{(2)}\,, \qquad \text{where} \quad \boldsymbol{\omega} = \begin{pmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{pmatrix} \quad \text{and} \quad \boldsymbol{x}_n = \begin{pmatrix} 1 \\ x_n^{(1)} \\ x_n^{(2)} \end{pmatrix}.$$

Geometrically, $(x_n^{(1)}, x_n^{(2)})$ is classified as category 1 (resp. 0) if it lies above (resp. below) the separation line

$$f(\boldsymbol{x}_n) = f(x^{(1)}, x^{(2)}) = 0\,. \tag{7.1.1}$$

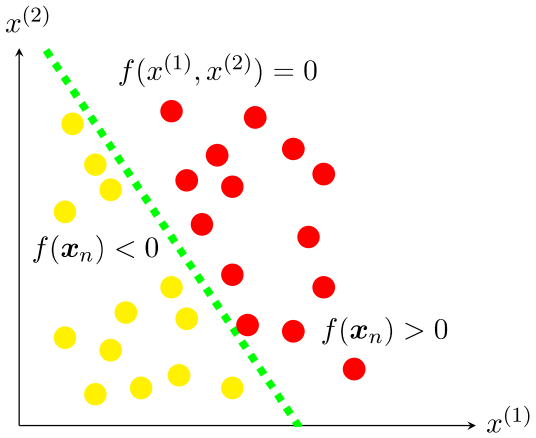


Figure 7.1.2: Perceptron: Category 1 (red); Category 2 (yellow)

To design a criterion to examine the suitable parameters, and hence the model, we observe from the separation line $f(x^{(1)}, x^{(2)}) = 0$ drawn in Figure 7.1.2 that the value $|f(x^{(1)}, x^{(2)})|$ increases if the point $(x^{(1)}, x^{(2)})$ lies further apart from the line. And so it is reasonable to think of the modulus of $f(x^{(1)}, x^{(2)})$ as an individual loss penalty corresponding to $\boldsymbol{x}_n$ to the overall misclassification, we therefore define the loss function:

$$L(\boldsymbol{\omega}) = \sum_{n \in I_N} |f(\boldsymbol{x}_n)|\,,$$

where $I_N := \{n \leq N : f(\boldsymbol{x}_n) \cdot y_n < 0\}$ is the set of indexes of all misclassified data; an equivalent expression can be given by:

$$L(\boldsymbol{\omega}) = -\sum_{n \in I_N} y_n \boldsymbol{\omega}^\top \boldsymbol{x}_n = \sum_{n=1}^{N} \max(0, -y_n \boldsymbol{\omega}^\top \boldsymbol{x}_n)\,, \tag{7.1.2}$$

and our goal now is to find the parameter $\boldsymbol{\omega} = (\omega_0, \omega_1, \cdots, \omega_D)^\top$, $D = 2$, that minimizes L. It is reminded that one of the difficulty encountered in this optimization problem is the dependence of L on the parameter $\boldsymbol{\omega}$ is implicitly involved in the index set I_N. **Gradient decent** method, see Section 6.1, will be used iteratively refine the parameters step by step against I_N.

To this end, we first initialize $\boldsymbol{\omega}^{(0)}$, and then a sequence of estimates $\{\boldsymbol{\omega}^{(t)}\}_{t=1}^{\infty}$ is constructed. For $t \in \mathbb{N}$, we further define $I_N^{(t)}$ to be the set of indexes of misclassified data given the parameter $\boldsymbol{\omega}^{(t)}$, and we also define recursively

$$L^{(t)}(\boldsymbol{\omega}) := \sum_{n \in I_N^{(t)}} |f(\boldsymbol{x}_n)| = -\sum_{n \in I_N^{(t)}} y_n \boldsymbol{\omega}^\top \boldsymbol{x}_n\,,$$

such that the iterated estimate is

$$\boldsymbol{\omega}^{(t+1)} = \boldsymbol{\omega}^{(t)} - \eta \nabla_{\boldsymbol{\omega}} L^{(t)}(\boldsymbol{\omega}^{(t)}) = \boldsymbol{\omega}^{(t)} + \eta \sum_{n \in I_N^{(t)}} y_n \boldsymbol{x}_n\,,$$

where, $\eta > 0$ is the hyperparameter **learning rate**. The iteration scheme will be terminated if $\|\boldsymbol{\omega}^{(t+1)} - \boldsymbol{\omega}^{(t)}\|_2$ is less than some prescribed threshold $\varepsilon > 0$, or if $|L^{(t+1)}(\boldsymbol{\omega}^{(\boldsymbol{t+1})}) - L^{(t)}(\boldsymbol{\omega}^{(t)})| < \varepsilon$. Geometrically,

see Figure 7.1.3, we update the separation line $f(\boldsymbol{x}) = 0$ by correcting its intercept and the normal vector $\boldsymbol{\omega}$ step by step.

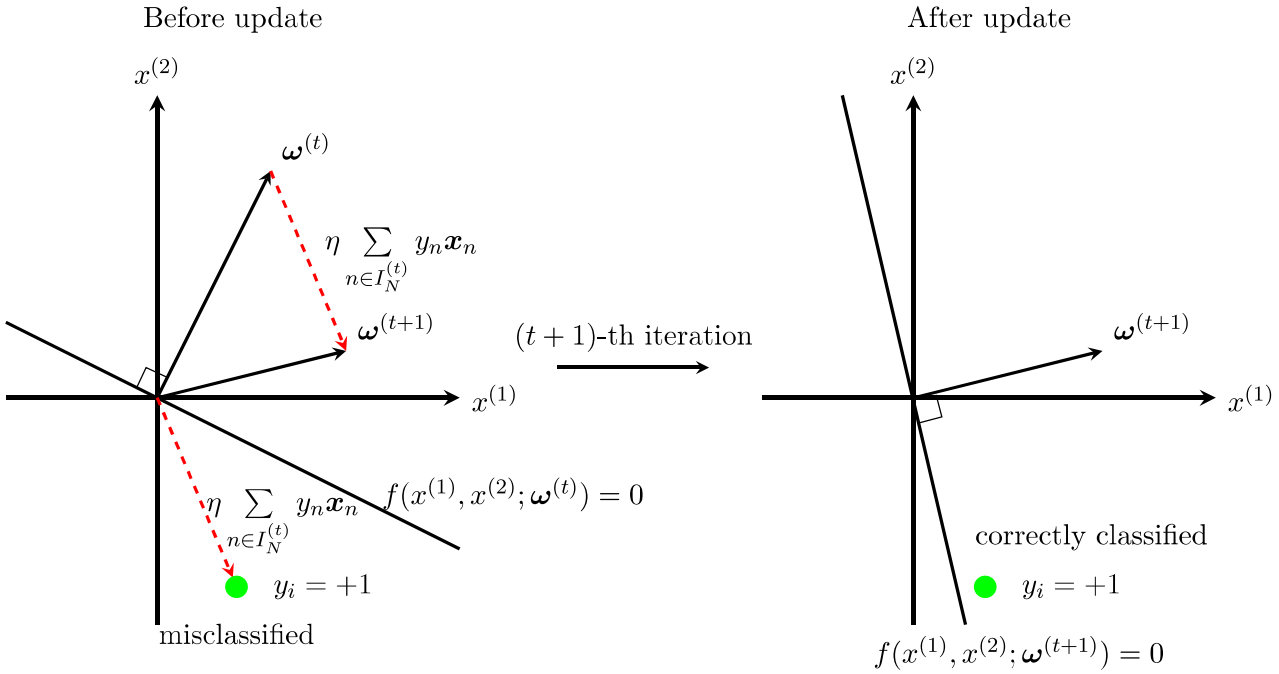


Figure 7.1.3: An update of the estimate for $\boldsymbol{\omega}$ under gradient descent method at the $(t+1)$-th iteration.

As explained in Section 6.1, this batch gradient descent method is not advocated especially when the sample size N is too large, as this results in high demand on computational effort for the sum $\sum_{n \in I_N^{(t)}} y_n \boldsymbol{x}_n$ in each single iteration; and this indetermined sum may bring in problem if all summands are dealt at a time. Therefore, one may adopt **stochastic gradient descent** (SGD) method instead, in which the parameter is updated by considering only one single datum in each iteration; and we repeat the procedures for N times to obtain the ultimate estimate for $\boldsymbol{\omega}$. If the data truely happens to be linearly separable, *i.e.* there exists $\boldsymbol{\omega}^* \in \mathbb{R}^{D+1}$ such that $\operatorname{sgn}((\boldsymbol{\omega}^*)^\top \boldsymbol{x}_{n_{t+1}}) = y_{n_{t+1}}$, then the convergence to optimal weights of $\boldsymbol{\omega}$ via stochastic gradient descent can be guaranteed. To this end, by comparing $\boldsymbol{\omega}^{(t)}$ and $\boldsymbol{\omega}^{(t+1)}$, *i.e.* the respective parameters before and after each update, the loss function for SGD at the $(t+1)$-th step is actually reduced; indeed, we have two scenarios:

1. if $(\boldsymbol{x}_{n_{t+1}}, y_{n_{t+1}})$ chosen is correctly classified, then $\boldsymbol{\omega}^{(t+1)} = \boldsymbol{\omega}^{(t)}$; otherwise
2. the loss function is $L^{(t+1)}(\boldsymbol{\omega}) = -y_{n_{t+1}} \boldsymbol{\omega}^\top \boldsymbol{x}_{n_{t+1}}$, where $(\boldsymbol{x}_{n_{t+1}}, y_{n_{t+1}})$ is the selected datum. Together with a learning rate η, the parameter $\boldsymbol{\omega}^{(t)}$ is updated by: $\boldsymbol{\omega}^{(t+1)} = \boldsymbol{\omega}^{(t)} + \eta y_{n_{t+1}} \boldsymbol{x}_{n_{t+1}}$. Since $\boldsymbol{x}_{n_{t+1}}$ is misclassified before the update, the loss $L^{(t+1)}(\boldsymbol{\omega}^{(t)})$ is positive, for instance, see Figure 7.1.3, $y_{n_{t+1}} = 1$ and there is an obscure angle between $\boldsymbol{\omega}^{(t)}$ and $\boldsymbol{x}_{n_{t+1}}$; and so

$$L^{(t+1)}(\boldsymbol{\omega}^{(t+1)}) = -y_{n_{t+1}} \left(\boldsymbol{\omega}^{(t+1)}\right)^\top \boldsymbol{x}_{n_{t+1}} = -y_{n_{t+1}} (\boldsymbol{\omega}^{(t)} + \eta y_{n_{t+1}} \boldsymbol{x}_{n_{t+1}})^\top \boldsymbol{x}_{n_{t+1}} = L^{(t+1)}(\boldsymbol{\omega}^{(t)}) - \underbrace{\eta \boldsymbol{x}_{n_{t+1}}^\top \boldsymbol{x}_{n_{t+1}}}_{\text{strictly} > 0},$$

the update corrects the weight vector in the direction of making the $(t+1)$-th individual loss less positive, *i.e.* apparent loss reduction, if it were $\boldsymbol{\omega}^{(t)}$ by augmenting the latter with this quantity of $\eta \boldsymbol{x}_{n_{t+1}}^\top \boldsymbol{x}_{n_{t+1}} > 0$.

We next discuss the convergence of the SGD scheme to an optimal perceptron classifier. For simplicity, we suppose that (i) all observation $\boldsymbol{x}_n$'s, for $n = 1, \cdots, N$, are uniformly bounded, *i.e.* $\|\boldsymbol{x}_n\|_2^2 \leq R$ for some $R > 0$; and (ii) there exists $\boldsymbol{\omega}^* \in \mathbb{R}^{D+1}$, see Figure 7.1.4, such that $\text{sgn}((\boldsymbol{\omega}^*)^\top \boldsymbol{x}_n) = y_n$ for all $n = 1, \cdots, N$, or equivalently, there exists a positive constant $\gamma > 0$ such that $y_n \left(\boldsymbol{\omega}^*\right)^\top \boldsymbol{x}_n \geq \gamma$ for all n; (iii) all the weights are initialized at 0, *i.e.* $\boldsymbol{\omega}^{(0)} = \mathbf{0}$. As a remark, the convergence to the optimal weight under SGD needs not to be equal to $\boldsymbol{\omega}^*$ here, and it can be a totally different normal vector. The convergence discussed here is totally different from the convergence result as introduced in Subsection 6.3.3.

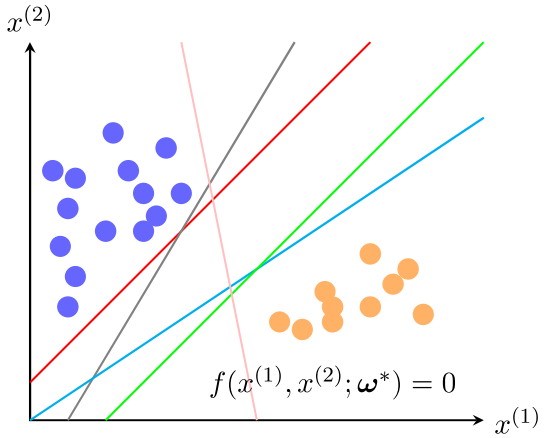


Figure 7.1.4: Some possible $\boldsymbol{\omega}^*$s all of which satisfy $\text{sgn}((\boldsymbol{\omega}^*)^\top \boldsymbol{x}_n) = y_n$ for all n.

It should be noted that if there is no more misclassified points, the numerical scheme may stop, and by definition, an exact classifier, which correctly classifies all data, is attained. Otherwise, if there are still misclassified points, modification of weights will still be carried on, and we now want to demonstrate that there is a ceiling for such a number of iterations, and hence the optimal classifier should be ultimately obtained. This kind of argument is essentially the same as the method of *infinite descent* in Number Theory, while this latter one is mostly used for the proof by contradiction. We first have some estimates, at the t-iteration step,

$$\begin{aligned}
\left(\boldsymbol{\omega}^*\right)^\top \boldsymbol{\omega}^{(t)} &= \left(\boldsymbol{\omega}^*\right)^\top \left(\boldsymbol{\omega}^{(t-1)} + \eta y_{n_t} \boldsymbol{x}_{n_t}\right) \\
&\geq \left(\boldsymbol{\omega}^*\right)^\top \boldsymbol{\omega}^{(t-1)} + \eta\gamma \\
&\geq \left(\boldsymbol{\omega}^*\right)^\top \boldsymbol{\omega}^{(t-2)} + 2\eta\gamma \qquad (\because \text{inductively}) \\
&\geq \cdots \\
&\geq \left(\boldsymbol{\omega}^*\right)^\top \boldsymbol{\omega}^{(0)} + t\eta\gamma = t\eta\gamma > 0\,. \qquad (\because \text{since } \boldsymbol{\omega}^{(0)} = \mathbf{0})
\end{aligned} \tag{7.1.3}$$

On the other hand, for its square norm:

$$\begin{aligned}
\|\boldsymbol{\omega}^{(t)}\|_2^2 &= \|\boldsymbol{\omega}^{(t-1)} + \eta y_{n_t} \boldsymbol{x}_{n_t}\|_2^2 = \|\boldsymbol{\omega}^{(t-1)}\|_2^2 + 2\eta y_{n_t} \left(\boldsymbol{\omega}^{(t-1)}\right)^\top \boldsymbol{x}_{n_t} + \eta \|\boldsymbol{x}_{n_t}\|_2^2 \\
&\leq \|\boldsymbol{\omega}^{(t-1)}\|_2^2 + \eta \|\boldsymbol{x}_{n_t}\|_2^2 \\
&\leq \|\boldsymbol{\omega}^{(t-1)}\|_2^2 + \eta R^2 \\
&\leq \cdots \\
&\leq \eta t R^2\,.
\end{aligned} \tag{7.1.4}$$

where the first inequality is due to the fact that $y_{n_t}\left(\boldsymbol{\omega}^{(t-1)}\right)^\top \boldsymbol{x}_{n_t} < 0$ especially when $\boldsymbol{x}_{n_t}$ is misclassified, the second inequality is due to our assumption on $\boldsymbol{x}_n$; and the last inequality is due to the repetitive application of the same argument. Finally, by the Cauchy-Schwartz inequality, $\|\boldsymbol{\omega}^*\|_2 \cdot \|\boldsymbol{\omega}^{(t)}\|_2 \geq (\boldsymbol{\omega}^*)^\top \boldsymbol{\omega}^{(t)} > 0$. Combining (7.1.3) with (7.1.4), we have the upper bound for the number of distinctive steps t, excluding those unchanged trivial step, so that misclassified datapoint can be identified out:

$$\|\boldsymbol{\omega}^*\|_2 \cdot \sqrt{\eta\,\mathsf{t}\,R^2} \geq \mathsf{t}\eta\gamma \qquad \text{which implies} \qquad \mathsf{t} \leq \frac{R^2\|\boldsymbol{\omega}^*\|_2^2}{\eta\gamma^2}, \tag{7.1.5}$$

and this is finite, that means no more misclassified points can there be beyond t steps.

Note that the learning rate η is inversely proportional to the upper bound of t in (7.1.5), and it hints the possiblility that a smaller learning rate η demands more iterations for the perceptron to learn the final optimal weights. In practice, we shall normalize the weights $\boldsymbol{\omega}^{(t)}$ to a unit vector, but not every step, after a certain number of iteration steps, such normalization will not affect the classification result of the dataset for each approximant $\boldsymbol{\omega}^{(t)}$.

The perceptron is an ancestor of *Artificial Neural Network* (ANN) and then *Deep Neural Network* (DNN),and particularly it is a simplified abstract model of the biological neurons in our brain, and it is also the simplest neural network, in the sense that it is just a single neuron. A perceptron $f(\boldsymbol{x}_n)$ can be visualized in the following Figure 7.1.5:

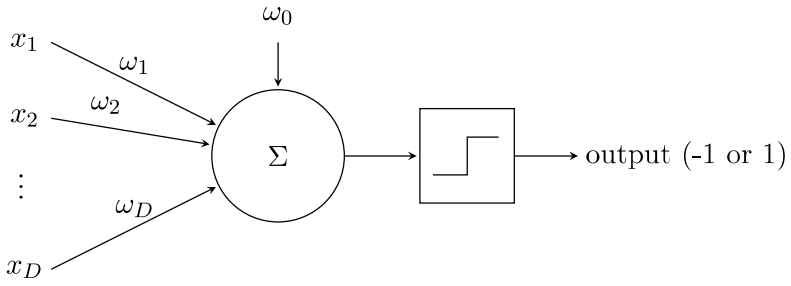


Figure 7.1.5: Perceptron as a neuron.

where Σ stands for the summation effect; and the so-called *activation function* is the sign function. Unfortunately, perceptron can only classify linearly for two clusters of data and these is a limitation, such as to explain the XOR (exclusive "OR") operation. By changing the choice of activation function, for example those other sigmoid functions in the next section, we can build up different neural networks; even further, if more layers of perceptron nodes are constructed a level by level, becoming a DNN now, the resulting structure can well explain the XOR operation.

Next, we shall illustrate the implementation of perceptron in **R**.

```
> perceptron <- function(X_train, X_test, y, eta=1e-3, epochs=1e5){
+     D <- ncol(X_train)
+     X_1 <- cbind(1, X_train)
+     # initialize weight vector
+     weight <- rep(1/(D + 1), D + 1)
```

```
+     loss_record <- c()
+     i <- 1
+     for (it in 1:epochs){
+         z <- weight%*%X_1[i,]
+         if(z < 0) {
+             y_pred <- -1
+         } else {
+             y_pred <- 1
+         }
+         loss_record <- c(loss_record, max(0, -y[i] * weight%*%X_1[i, ]))
+         weight <- weight + eta * (y[i]!=y_pred) * y[i] * X_1[i,]
+         if (i >= nrow(X_train)){
+             i <- 1
+         } else {
+             i <- i + 1
+         }
+     }
+     y_pred <- cbind(1, X_test)%*%weight
+     return(list(weight=weight, y_pred=y_pred, loss_record=loss_record))
+ }
```

Programme 7.1.1: Perceptron function in **R**.

We implement this `perceptron()` function in Programme 7.1.1 to the HSI financial ratio dataset.

```
> set.seed(4012)
>
> df <- read.csv("fin-ratio.csv")
> X <- df[-ncol(df)]
> y <- df[ncol(df)]
> y[y==0] <- -1                  # transform the output to {-1, 1}
>
> # training:testing = 7:3 and shuffle the training dataset
> id <- sample(1:nrow(df), size=round(0.7*nrow(df)))
> X_train <- as.matrix(X[id,])
> y_train <- y[id,]
> X_test <- as.matrix(X[-id,])
> y_test <- y[-id,]
>
> # prediction on testing dataset
> model <- perceptron(X_train, X_test, y_train, eta=0.05, epochs=1e5)
> y_pred <- model$y_pred
> y_pred[y_pred >= 0] <- 1
> y_pred[y_pred < 0] <- -1
> conf <- table(y_test, y_pred)            # confusion matrix
> conf
```

```
      y_pred
y_test  -1   1
    -1 196   1
     1   2   5
> sum(diag(conf))/length(y_test)
[1] 0.9852941
```

Programme 7.1.2: Perceptron using Programme 7.1.1 with HSI financial ratio dataset in **R**.

The accuracy rate is $(196 + 5)/204 = 98.53\%$.

Lastly, we shall revisit the XOR example in **R**.

```
> set.seed(4012)
>
> XOR <- function(X, y, eta=1e-1, max_epoch=1e5){
+     D <- ncol(X)
+     X_1 <- cbind(1, X)
+     # initialize weight vector
+     weight <- rep(1/(D + 1), D + 1)
+     loss <- c()
+     acc <- 0
+     j <- 1
+     best_weight <- weight
+     best_acc <- 0
+     for (i in 1:max_epoch){
+         for (j in sample(1:length(y), length(y))){
+             z <- weight%*%X_1[j,]
+             if(z < 0) {
+                 y_j <- -1
+             } else {
+                 y_j <- 1
+             }
+             loss <- c(loss, max(0, -y[j] * weight%*%X_1[j, ]))
+             weight <- weight + eta * (y[j]!=y_j) * y[j] * X_1[j,]
+
+             y_pred <- X_1%*%weight
+             y_pred[y_pred >= 0] <- 1
+             y_pred[y_pred < 0] <- -1
+             acc <- mean(y_pred==y)
+             if (best_acc < acc) {best_weight <- weight; best_acc <- acc}
+             if (acc > 0.75){break}
+         }
+     }
+     return(list(weight=weight, best_weight=best_weight, loss_record=loss))
+ }
```

```
>
> X <- matrix(c(0,0, 1,1, 0,1, 0,1), ncol=2)
> y <- c(1, 0, 0, 1)
> y[y==0] = -1
> model <- XOR(X, y, eta=0.1, max_epoch=1e5)
> y_pred <- cbind(1, X)%*%model$best_weight
> y_pred[y_pred >= 0] <- 1
> y_pred[y_pred < 0] <- -1
> conf <- table(y, y_pred)          # confusion matrix
> conf
    y_pred
y    -1 1
  -1  1 1
  1   0 2
> sum(diag(conf))/length(y)         # best accuracy
[1] 0.75
> model$best_weight
[1]  0.03333333  0.13333333 -0.06666667
> length(model$loss_record) # stop at max epoch, not converge with accuracy 1
[1] 400000
>
> par(mar=c(4.3,4.3,2,2))
> plot(X, col=y+2, cex=2, pch=16, xlim=c(-0.2, 1.2), ylim=c(-0.2,1.2),
+      main="XOR", xlab=expression(x^{(1)}), ylab=expression(x^{(2)}))
> w0 <- model$best_weight[1]
> w1 <- model$best_weight[2]
> w2 <- model$best_weight[3]
> abline(-w0/w2, -w1/w2)
> text(0.2, 0.4, expression(omega[0]+omega[1]*x^{(1)}+omega[2]*x^{(2)} == 0))
```

Programme 7.1.3: XOR in **R**.

The resulting plot is

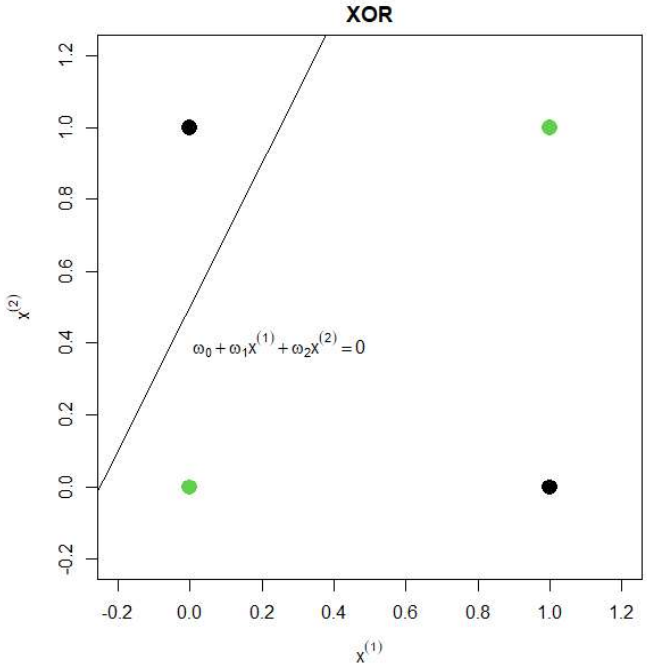


Figure 7.1.6: XOR plot from Programme 7.1.1 in **R**.

Here the `XOR()` function is an adjustment of Programme 7.1.1. Although the number of samples is only 4, we know in advance that the algorithm does not converge with an accuracy of 1 (versus best accuracy of 0.75), and we need to set a maximum number of epoch through the argument `max_epoch=1e5`. As usual, we set the seed by `set.seed(4012)` in order to replicate the result again. Finally, in line 53, the total number of iteration is `400000` which is four (=total number of datum) times the maximum number of epoch set in advance, which indicates that the `XOR()` function does not converge with an accuracy of 1 among all the iterations. The best accuracy of the model is 0.75!

McCulloch and Pitts (1943) postulated a simple mathematical model to explain how biological neurons work. There was not much progress in this area until 1970 when the computer was invented. In 1970's, Hopfield invented **back-propagation** algorithm to train neural networks so as to match the fitted output with the actual demanded response as much as possible. Since then, there are many successful examples and applications in the labs. In 1980's, research moved from the labs to commercial world, typical applications like detecting fraud credit card transactions, real estate appraisal, and data mining. Before we go into the topics of ANN, we need to learn a bit about the structure our human brains.

7.2 Human Brain and Artificial Neuron

Logistic regressions, EM Algorithms, Bayes classifiers, Classification Trees, and Random Forests are very commonly used statistical techniques for classification; another completely different approach is the Artificial Neural Network *ANN*. The latter approach is to mimic the functioning and mechanism of our human

brain. Although how our brain memorizes, recognizes, and generalizes pattern, some aspects of the network structure of brain are well-known at least to the neuroscientists. Their understanding and proposed theory actually provide a useful and workable model to lay down the architectures of ANNs and DNNs; indeed, one of the godfathers of Deep Learning, Yann LeCun[1] was inspired by physiologists on the mechansim behind vision of a cat, from light on retina to the cat's brain, he then used the same architecture to build up the ever first convolutional Neural Network (*CNN*).

Our brain consists of approximately 128 billions of some specific type of a cell, known as neuron. Each of these neurons is typically connected with thousands to 10,000 other neurons; and they are connected with each other by forming layers, and seems to have more connections among layers closer to the front head than at the back of a head. Most importantly, unlike other cells, these neurons will not regenerate. It is widely accepted that these neurons are responsible for our ability for memorizing, learning, generalizing, and thinking. Within a neuron, there are four main components: *Dendrites*, *cell body*, *Axon*, and *Synapse*; see Figure 7.2.1. These neurons are connected to form a huge and very complicated network. The exact functioning of these neurons is still a mystery, but a very simple mathematical model that mimics these neurons provides a surprising good performance in machine pattern recognition, classification, and prediction.

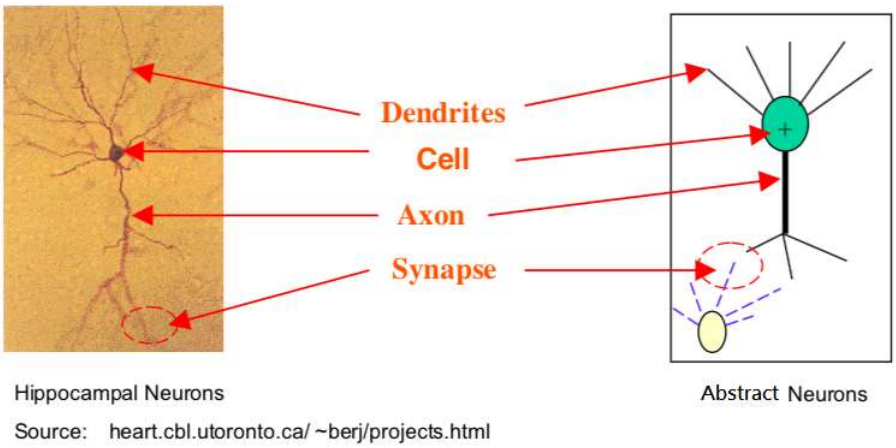


Figure 7.2.1: Hippocampal neuron.

In principle, dendrites are responsible for receiving information from nearly neurons or other sensors; cell body is for processing the collected information from dendrites; axon amplifies the signal carried by the processed information to be sent to other neurons; and synapse is the junction between axon end and dendrites of other neurons. To mimic the structure of a neuron, we have the following artificial neuron as depicted in Figure 7.2.2:

[1]He is also one of the main creators of DjVu image compression technology.

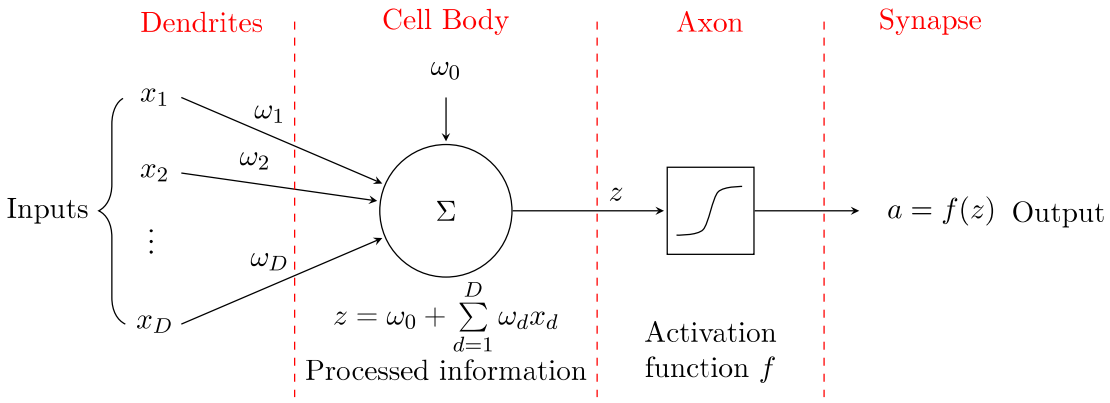
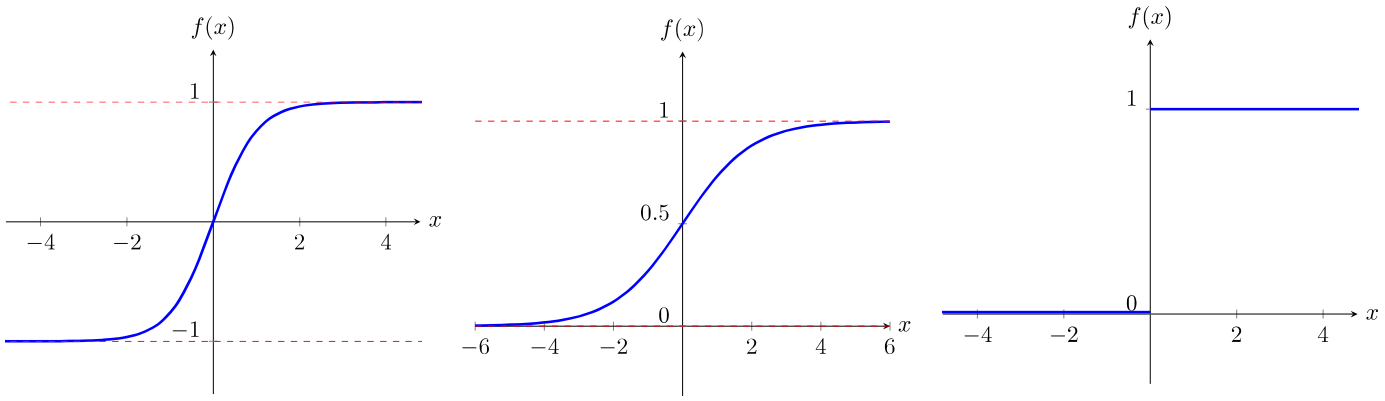


Figure 7.2.2: An artificial neuron.

Here $x_1, \cdots, x_D$ are the inputs received from other neurons or from the outside environment. The total balanced input z is formed from the linear combination of these inputs with the respective weights $\omega_1, \cdots, \omega_D$, together with another constant ω_0 known as the bias. The *transfer function* or *activation function* f converts the the input z to output, the predicted label $\hat{y} = a = f(z)$. The output $\hat{y}$ will go to other neurons as an input. Figure 7.2.3 illustrates some commonly used activation functions or sigmoid functions inside an axon:



Tanh (Hyperbolic Tangent)
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Logistic
$$f(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}$$

Zero-one (or Sign function)
$$f(x) = \begin{cases} 0 \text{ (or } -1), & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$$

Figure 7.2.3: Commonly used activation functions inside an axon.

Usually, logistic function is used, especially with continuous inputs without sharp changes; that is why artificial neural network is closely related to logistic regression, which is actually one single neuron. Besides, we also recall the nice property of logistic function, namely the derivative of the logistic function is $f'(x) = f(x)[1 - f(x)]$; meanwhile, the derivative of the hyperbolic tangent (tanh) function f is $f'(x) = 1 - f^2(x)$.

7.3 Feed-forward Network

In an ANN, artificial neurons are connected from one layer to other to form a network, and there is a certain amount of architecture but not completely random.

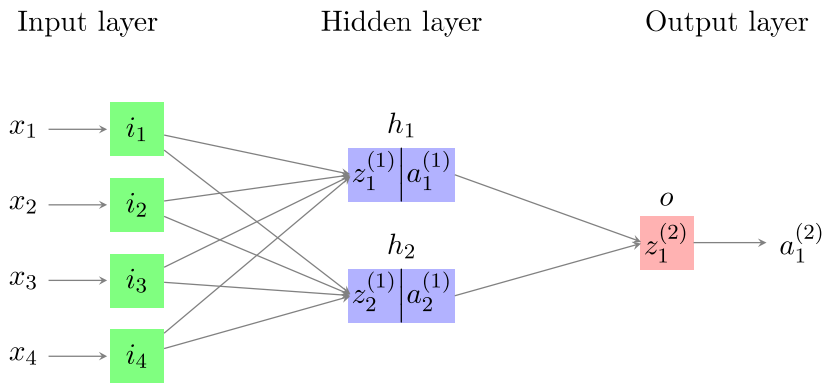


Figure 7.3.1: Illustration of a 4-2-1 ANN.

Referring to Figure 7.3.1, in this network, we can find three layers: input layer, hidden layer and output layer. The numbers of neurons in the input, hidden and output are respectively 4, 2, and 1; and so this ANN is called a 4-2-1 ANN.

1. The number of hidden layers can be zero, one, two, and so on; and whenever these is more than one layer, the neural network is already considered as a DNN. Yet, if there are not many hidden layers, we still call it as a shallow ANN;
2. The number of neurons in the input and output layer are determined by the nature of the problem; for instance, the inputs could be pixels of an image, and so higher the resolution, more the number of inputs; while for the number of outputs, it also varies depending on the purpose of usage. On the other hand, the number of neurons in the hidden layer is user-defined and this is a hyperparameter, indeed;
3. Within each layer, neurons are not connected with each other. Neurons in one layer are connected only to neurons in the forward next layer, and so it is feeding forward.
4. Each directed line joining two neurons, from i to j, is associated with a weight ω_{ji}. These weights are model parameters need to be estimated from the training dataset.

7.4 ANN using Python and R

Given the feature input $\boldsymbol{x}$ and the label $\boldsymbol{y}$ in a training set, we have to estimate the model parameters ω_{ji} such that the sum of square error:

$$\mathcal{E} = (\mathbf{y} - \mathbf{a}^{(L)})^\top (\mathbf{y} - \mathbf{a}^{(L)}), \tag{7.4.1}$$

is minimized, where $\mathbf{a}^{(L)}$ is the prediction from the ANN, that is $\boldsymbol{a}^{(L)} = \boldsymbol{f}(\boldsymbol{z}^{(L)})$ with $\boldsymbol{f}$ being an activation function, and L is the index of the output layer, in which the 0-th layer is the input layer, while the hidden layers are indexed by $\ell = 1, \cdots, L-1$. Generally, the algebraic form of $\boldsymbol{a}^{(L)}$ is too complex to write down explicitly. Usually, the loss function $\mathcal{E}$ has many local minima, and usually the *Back-propagation* algorithm is used to find various estimates of the model parameters ω_{ji} so as to achieve the minimum. Let us illustrate their applications to the classification problem for the Iris flower dataset (see Subsection **??**), using the 4-2-1 ANN in Figure 7.3.1, and we first transform the character labels to the numbers, *e.g.* via the `factor()`

function in **R**. Next, we implement the 4-2-1 ANN model with the `nnet()` function inside the `nnet` library. However, this library is only able to manage one single layer of ANN such that any additional hidden layers cannot be stacked, *i.e.* it does not support multi-layers ANN. Moreover, the optimization algorithm used in the `nnet()` function is the BFGS algorithm, see Subsection 6.2.3, instead of the stochastic, mini-batch, or batch Gradient Descent methods.

```
> library(nnet)
> set.seed(4012)
>
> df <- iris
> classes <- factor(df[,ncol(df)])
> levels(classes) <- 1:length(levels(classes))
> df[,ncol(df)] <- classes
> df <- apply(df, 2, as.numeric)
> # One hidden layer with size=2, indicating 2 neurons in the hidden layer,
> # and linout=TRUE, indicating a linear output unit is used in output layer
> # such that a_1^(2) is just a linear combination; ncol(df)=5
> iris.nn <- nnet(x=df[,-ncol(df)], y=df[,ncol(df)], size=2, linout=TRUE)
# weights:  13
initial  value 190.701697
iter  10 value 15.266521
iter  20 value 4.982184
iter  30 value 4.751855
iter  40 value 4.249477
iter  50 value 2.684285
iter  60 value 1.895233
iter  70 value 1.769411
iter  80 value 1.414936
iter  90 value 1.374228
iter 100 value 1.362985
final  value 1.362985
stopped after 100 iterations
> summary(iris.nn)
a 4-2-1 network with 13 weights
options were - linear output units
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1
-56.24 -23.96 -23.06  42.94  35.79
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2
 -3.82  12.86 -32.15   5.97   9.01
  b->o  h1->o  h2->o
  1.00   0.99   0.99
```

Programme 7.4.1: 4-2-1 ANN with Iris flower dataset in **R**.

Here, `b` stands for bias, `i` stands for input layer neurons, `h` stands for hidden layer neurons, and `o` stands for output layer neurons, as shown in Figure 7.3.1. Now, the interconnection between these variables are described by:

$$\begin{aligned} z_1^{(1)} &= -56.24 - 23.96x_1 - 23.06x_2 + 42.94x_3 + 35.79x_4\,; \\ z_2^{(1)} &= -3.82 + 12.86x_1 - 32.15x_2 + 5.97x_3 + 9.01x_4\,; \\ a_1^{(1)} &= \frac{\exp(z_1^{(1)})}{1+\exp(z_1^{(1)})}, \quad a_2^{(1)} = \frac{\exp(z_2^{(1)})}{1+\exp(z_2^{(1)})}\,; \\ a_1^{(2)} &= 1 + 0.99a_1^{(1)} - 0.99a_2^{(1)}\,. \end{aligned} \tag{7.4.2}$$

Let us use the 1st, 51th, and 101th observations from the original iris dataset, *i.e.* without shuffling, to illustrate how this ANN makes prediction in **R**, see Programme 7.4.2.

```
> df[c(1, 51, 101),]
     Sepal.Length Sepal.Width Petal.Length Petal.Width Species
[1,]          5.1         3.5          1.4         0.2       1
[2,]          7.0         3.2          4.7         1.4       2
[3,]          6.3         3.3          6.0         2.5       3
```

Programme 7.4.2: 1st, 51th and 101th observations from the iris dataset in **R**.

The 1st observation is $\boldsymbol{x}_1 = (x_1, x_2, x_3, x_4)^\top = (5.1, 3.5, 1.4, 0.2)^\top$. According to the formulae (7.4.2),

$$\begin{aligned} z_1^{(1)} &= -56.24 - 23.96(5.1) - 23.06(3.5) + 42.94(1.4) + 35.79(0.2) = -191.872\,, \\ z_2^{(1)} &= -3.82 + 12.86(5.1) - 32.15(3.5) + 5.97(1.4) + 9.01(0.2) = -40.599\,, \\ a_1^{(1)} &= \frac{\exp(z_1^{(1)})}{1+\exp(z_1^{(1)})} = 4.689 \times 10^{-84}\,, \quad a_2^{(1)} = \frac{\exp(z_2^{(1)})}{1+\exp(z_2^{(1)})} = 2.334 \times 10^{-18}\,, \\ a_1^{(2)} &= 1.99 + a_1^{(1)} - 0.99a_2^{(1)} = 1\,. \end{aligned}$$

Similarly for the 51st observation, $\boldsymbol{x}_{51} = (7, 3.2, 4.7, 1.4)^\top$, $z_1^{(1)} = -45.828$, $z_2^{(1)} = 23.993$, $a_1^{(1)} = 1.251 \times 10^{-20}$, $a_2^{(1)} = 1$, and $a_1^{(2)} = 1.99$; and the 101st observation, $\boldsymbol{x}_{101} = (6.3, 3.3, 6, 2.5)^\top$, the corresponding values are $z_1^{(1)} = 63.829$, $z_2^{(1)} = 29.448$, $a_1^{(1)} = 1$, $a_2^{(1)} = 1$ and $a_1^{(2)} = 2.98$.

For **R**, $a_1^{(2)}$ is stored in `iris.nn$fitted.values`, and we can produce the classification table according to Programme 7.4.3:

```
> pred <- round(iris.nn$fit)              # round the fitted values
> table(df[,ncol(df)], pred)              # confusion matrix
  pred
    1  2  3
1  50  0  0
2   0 49  1
3   0  0 50
```

Programme 7.4.3: Prediction of iris flower with 4-2-1 ANN in **R**.

In Python, the `MLPRegressor()` (Multiple Layer Perceptron Regressor) function in the `sklearn.neural_network` can implement single layer or even multi-layer ANN, which is specified by the parameter `hidden_layer_sizes`.

As a comparison with `nnet()` in **R**, L-BFGS (Limited memory BFGS) is used as the optimization algorithm with the argument `solver="lbfgs"`, which is the extension to the original BFGS algorithm, a version of quasi-Newton method, see Subsection 6.2.4. As usual, logistic activation function is used with the argument `activation="logistic"`, and 2 hidden neurons are used in the hidden layer with the parameter `hidden_layer_sizes=2` (If a 4-3-2-1 model is used, we can set `hidden_layer_sizes=(3,2)`, however, the activation function used is uniformly the same across all hidden layers.)

```
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np

iris = load_iris()
X, y = iris.data, iris.target

# One hidden layer with size=2, indicating 2 neurons in the hidden layer,
# and MLPRegressor, indicating a linear output unit is used in output layer
# such that a_1^(2) is just a linear combination;
from sklearn.neural_network import MLPRegressor
model = MLPRegressor(hidden_layer_sizes=2, max_iter=1000, solver="lbfgs",
                     activation="logistic", random_state=4012)
model.fit(X, y)
y_pred = model.predict(X)
y_pred = np.round(y_pred)   # round the fitted values to the nearest integers

from sklearn.metrics import classification_report
print(classification_report(y, y_pred))
print(pd.crosstab(y, y_pred))                # confusion matrix
print(model.coefs_)                          # get weights W^(1), W^(2)
print(model.intercepts_)                     # get bias b^(1), b^(2)
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      0.96      0.98        50
           2       0.96      1.00      0.98        50

    accuracy                           0.99       150
   macro avg       0.99      0.99      0.99       150
weighted avg       0.99      0.99      0.99       150

col_0  -0.0   1.0   2.0
row_0
0         50     0     0
1          0    48     2
2          0     0    50
```

```
[array([[ -2.25384929,  -6.88024598],
        [ 31.5264005 ,  -6.65706057],
        [-24.44260984,  11.87532164],
        [-13.66810773,  11.46066087]]), array([[-0.99138422],
        [ 1.00168146]])]
[array([ -3.46083348, -15.44372469]), array([0.9912316])]
```

Programme 7.4.4: 4-2-1 ANN with Iris flower dataset in Python.

From the outputs of Programme 7.4.4, the accuracy is $(52 + 48 + 50)/150 = 98.67\%$ and the 4-2-1 ANN model is

$$\begin{aligned} z_1^{(1)} &= -3.46083348 - 2.25384929x_1 + 31.5264005x_2 - 24.44260984x_3 - 13.66810773x_4\,; \\ z_2^{(1)} &= -15.44372469 - 6.88024598x_1 - 6.65706057x_2 + 11.87532164x_3 + 11.46066087x_4\,; \\ a_1^{(1)} &= \frac{\exp(z_1^{(1)})}{1+\exp(z_1^{(1)})}, \quad a_2^{(1)} = \frac{\exp(z_2^{(1)})}{1+\exp(z_2^{(1)})}\,; \\ a_1^{(2)} &= 0.9912316 - 0.99138422a_1^{(1)} + 1.00168146a_2^{(1)}\,. \end{aligned} \tag{7.4.3}$$

Next, we illustrate the use of a 6-2-1 ANN for HSI dataset in Programme 7.4.5 in **R**.

```
> library(nnet)
> set.seed(4012)
>
> df <- read.csv("fin-ratio.csv")
> fin.nn <- nnet(x=df[,-ncol(df)], y=df$HSI, size=2, linout=TRUE)
# weights:  17
initial  value 187.663107
iter  10 value 30.482958
iter  20 value 30.482482
iter  30 value 30.473251
final  value 30.471564
converged
> summary(fin.nn)
a 6-2-1 network with 17 weights
options were - linear output units
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1
  0.63  -1.23  -1.88   4.17   0.27   0.91  -0.09
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2
  1.21  -0.32   0.25   3.05   1.98   2.90   4.49
 b->o h1->o h2->o
 0.54 -0.54  0.05
>
> pred <- round(fin.nn$fit)
> table(df$HSI, pred)   # confusion matrix, this model predicts all as 0
 pred
    0
0 648
```

```
1   32
```

Programme 7.4.5: 6-2-1 ANN with HSI dataset in **R**

However, this ANN predicts all observations as non-HSI constituent stocks, *i.e.* $\hat{y} = 0$. This is a major problem of ANN that the final model heavily depends on the initial seeds for the estimated parameters. When we run `nnet()` at different times, we may obtain drastically different results; this drawback is illustrated in Figure 7.4.1 as below:

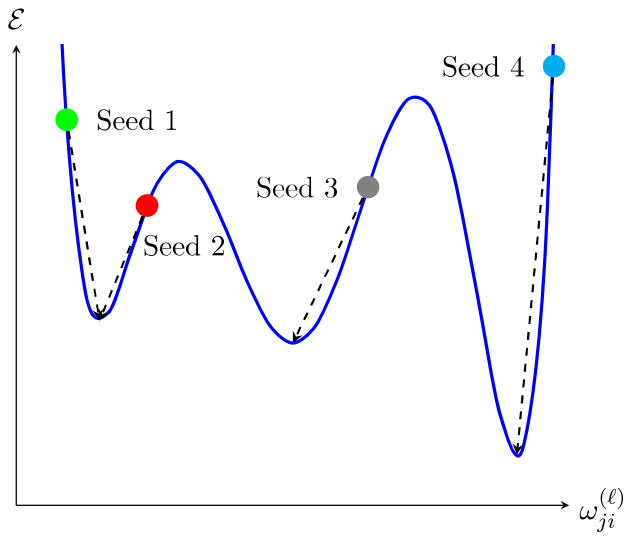


Figure 7.4.1: The 3 solutions of 4 different starting points in a parameter space.

The error function $\mathcal{E}$ in (7.4.1), while depending on the weights $\left(\omega_{ji}^{(\ell)}\right)$ of the ANN, consists of many local minima in the multi-dimensional space of $\omega_{ji}^{(\ell)}$. As `nnet()` in **R** randomly assign initial seeds for parameters $\omega_{ji}^{(\ell)}$ for the later iteration procedure, namely the backpropagation, see Section 7.6; the backpropagation method is essentially a gradient descent method as introduced in Section 6.1 for minimizing the error function $\mathcal{E}$. The resulting weights $\omega_{ji}^{(\ell)}$ of the ANN may only be the parameters at a local minimum of $\mathcal{E}$, therefore we need to run `nnet()` in **R** several times with different sets of initial parameters in order to propose the (relatively) global optimal weights. To this end, we provide an modified version of the `nnet()` function in as shown **R** in Programme 7.4.6:

```
> # Try nnet(x,y) K times and output the best trial
> # X is the matrix of input variable
> # y is the dependent value; which must be factor if linout=F
> library(nnet)
> ANNet <- function(x, y, size, maxit=100, linout=F, try=5){
+     ANN <- nnet(y~., data=x, size=size, maxit=maxit, linout=linout)
+     E <- ANN$value                    # value in the first trial
+     Best_ANN <- ANN
+
+     for (i in 2:try) {
+         ANN <- nnet(y~., data=x, size=size, maxit=maxit, linout=linout)
```

```
+          if (ANN$value < E){            # check if improved
+              E <- ANN$value             # save the best value
+              Best_ANN <- ANN            # save the best model
+          }
+      }
+      return (Best_ANN)
+ }
```

Programme 7.4.6: Improved `nnet()` in **R**.

This function allows users to specify number of trials with the parameter `try` and return the best possible model `Best_ANN` among these attempt. For instance, we have a much better performing ANN model as in Programme 7.4.7 in **R**:

```
> set.seed(4012)
> df <- read.csv("fin-ratio.csv")
> fin.nn <- ANNet(x=df[,-ncol(df)], y=df$HSI, size=2, linout=T, try=10)
> fin.nn$value
[1] 2.963041
> summary(fin.nn)
a 6-2-1 network with 17 weights
options were - linear output units
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1  i5->h1  i6->h1
3301.20 -112.43   87.59 -351.77   15.16   -9.42   -5.15
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2  i5->h2  i6->h2
   0.86    2.81  -10.25  -34.33  -18.24   13.73   13.53
   b->o   h1->o   h2->o
   0.97   -0.97    0.00
>
> pred <- round(fin.nn$fit)
> conf <- table(df$HSI, pred)                    # confusion matrix
> conf
 pred
    0   1
0 647   1
1   2  30
> sum(diag(conf)) / length(df$HSI)               # accuracy
[1] 0.9955882
```

Programme 7.4.7: Improved `nnet()` using `ANNet()` function from Programme `Prog:Modified ANN in R` with the HSI dataset in **R**.

Similarly in Python, the improved ANN function `ANNet()` can be built by:

```
# Try nnet(x,y) K times and output the best trial
# X is the matrix of input variable
# y is the dependent value
from sklearn.neural_network import MLPRegressor, MLPClassifier
```

```
5
6  def ANNet(X, y, size, linout=True, max_iter=1e4, trial=5):
7      if linout:
8          model = MLPRegressor(hidden_layer_sizes=size, max_iter=max_iter,
9                               solver="lbfgs", activation="logistic")
10     else:
11         model = MLPClassifier(hidden_layer_sizes=size, max_iter=max_iter,
12                               solver="lbfgs", activation="logistic")
13
14     model.fit(X, y)
15     score = model.score(X, y)   # score in the first trial
16     Best_ANN = model
17
18     for i in range(trial):
19         if linout:
20             model = MLPRegressor(hidden_layer_sizes=size, max_iter=max_iter,
21                                  solver="lbfgs", activation="logistic")
22         else:
23             model = MLPClassifier(hidden_layer_sizes=size, max_iter=max_iter,
24                                   solver="lbfgs", activation="logistic")
25         model.fit(X, y)
26         if model.score(X, y) > score:   # check if improved
27             score = model.score(X, y)   # save the best score
28             Best_ANN = model            # save the best model
29     return Best_ANN
```

Programme 7.4.8: Improved ANN in Python.

Here the `model.score()` function returns the mean accuracy of the feature matrix $\boldsymbol{X}$ and the labels $\boldsymbol{y}$, which is fundamentally different than the `ANN$value` object for loss in Programme 7.4.6 in **R**. We maximize the former one but we minimize the latter one. Finally, we use the `ANNet()` function defined in Programme 7.4.8 to fit the HSI financial ratio dataset:

```
1  import pandas as pd
2  import numpy as np
3  np.random.seed(4012)
4
5  df = pd.read_csv("fin-ratio.csv")
6  X = df.drop(columns="HSI")
7  y = df["HSI"]
8
9  model = ANNet(X, y, size=2, linout=True, max_iter=1000, trial=10)
10 y_pred = model.predict(X)
11 y_pred = np.round(y_pred)
12
13 from sklearn.metrics import classification_report
14 print(classification_report(y, y_pred))
15 print(pd.crosstab(y, y_pred))                # confusion matrix
16 print(model.coefs_)                          # get weights W^(1), W^(2)
17 print(model.intercepts_)                     # get bias b^(1), b^(2)
```

```
1                precision    recall  f1-score   support
2
3             0       1.00      1.00      1.00       648
4             1       0.91      1.00      0.96        32
```

```
5
6       accuracy                           1.00        680
7      macro avg       0.96      1.00      0.98        680
8   weighted avg       1.00      1.00      1.00        680
9
10  col_0  -0.0   1.0
11  HSI
12  0        645     3
13  1          0    32
14  [array([[  7.6322765 ,  -1.09354803],
15          [-11.86577696, -10.94272935],
16          [ 24.3276239 ,   7.82883666],
17          [  0.41621982,   6.72390623],
18          [  5.23689022,   7.94792356],
19          [ -1.0633425 ,   0.5782938 ]]), array([[ 0.93980627],
20          [-0.9362688 ]])]
21  [array([-233.67806882, -241.72430557]), array([-0.00055586])]
```

Programme 7.4.9: Improved ANN using `ANNet()` from Programme 7.4.8 with the HSI dataset in Python.

The accuracy rate is $(645 + 32)/680 = 99.56\%$.

7.4.1 ANN with Logistic Output

In **R**, we have seen the `nnet()` function with the linear output option `linout=T` (`TRUE`), and the output is a linear function of the values delivered by the neurons in the hidden layer and so it is a real vector. When using this for classification problems, like those for iris flowers or HSI financial ratio dataset, we have to round the output to its nearest integer (can be larger or smaller than the corresponding original real number). Nevertheless, we may use the default option `linout=F` (`FALSE`), and then K different logistic functions, corresponding to K labels, are used to connect the hidden neurons with the output. This can be interpreted as the probability of an observation with a feature $\boldsymbol{x}$ belonging to a particular label group.

Again, let us look at the Iris flower dataset. Note that the iris flower label has $K = 3$ categories, so that the fitted values in the output have 3 columns. Suppose that the fitted values are respectively $z_j^{(2)}$, for $j = 1, 2, 3$. Then the probability of the feature variable $\boldsymbol{x}$ belonging to the the j-th group is:

$$p_j = a_j^{(2)} = \text{logit}(z_j^{(2)}) = \frac{\exp(z_j^{(2)})}{\sum_{k=1}^{K} \exp(z_k^{(2)})} \qquad \text{where} \quad K = 3,\ j = 1, 2, 3\,,$$

where $\text{logit}(\cdot)$ is also called the softmax function.

For this iris flower example, the output has $K = 3$ levels, the diagram for this 4-2-3 ANN with logistic output is further depicted in Figure 7.4.2.

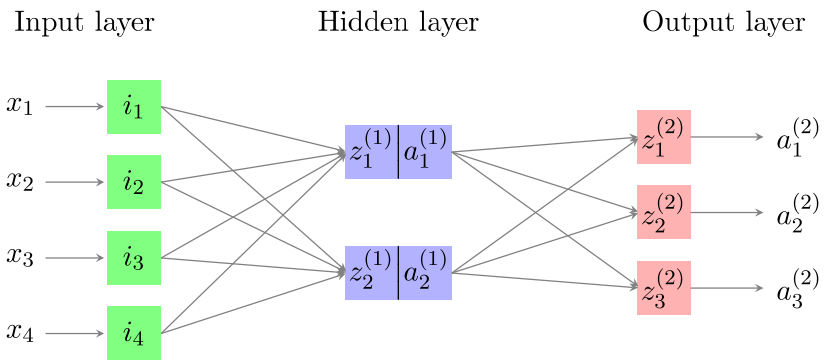


Figure 7.4.2: A 4-2-3 ANN for Iris flower dataset.

Let us look at how to use this logistic output option in `nnet()` in Programme 7.4.10 with the Iris flower dataset.

```
> set.seed(4012)
> df <- iris
> classes <- factor(df$Species)
> levels(classes) <- 1:length(levels(classes))
> df$Species <- classes
> df <- apply(df, 2, as.numeric)
> X <- df[,-ncol(df)]                          # define input X
> y <- as.factor(df[,ncol(df)])                # output as factor
>
> iris.nn <- ANNet(X, y, size=2, linout=F, maxit=200, try=10)
> iris.nn$value                                # best value after 10 attempts
[1] 4.922277
> summary(iris.nn)                             # display weights
a 4-2-3 network with 19 weights
options were - softmax modelling
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1
 527.23   28.84   50.56 -130.48 -131.23
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2
  31.44   28.30  -93.87   30.63   47.26
  b->o1  h1->o1  h2->o1
 167.92  119.62 -244.02
  b->o2  h1->o2  h2->o2
-112.29  166.13  148.61
  b->o3  h1->o3  h2->o3
 -55.63 -285.67   95.86
>
> # find the column number of the max. fitted values,
> # See further description below
> pred <- max.col(iris.nn$fit)
> table(y, pred)                               # confusion matrix
   pred
```

```
y     1  2  3
  1  50  0  0
  2   0 49  1
  3   0  0 50
```

Programme 7.4.10: Using logistic functions in `ANNET()` from Programme 7.4.6 with `lineout=F` for Iris flower dataset in **R**.

Note that for logistic output, the prediction of the i-th observation is

$$a_j^{(2)} = \text{logit}(z_j^{(1)}) = \mathbb{P}(\text{y}_i = j)\,.$$

Normally speaking, we predict that the observation $\boldsymbol{x}$ belongs to group j if p_j is the maximum among the others. Since p_j is maximum if and only if $a_j^{(2)}$ is maximum and there are 150 observations in total, we assign the label to `pred` according to the maximum column index in each row (observation) of `iris.nn$fit` using the `max.col()` function. By using this method, we have only 1 error case out of 150 cases and the error rate is 0.67%, which is so low!

Similarly in Python, the `MLPClassifier()` (Multiple Layer Perceptron Classifier) function in the `sklearn.neural_network` is used for the logistic output of 4-2-3 ANN.

```
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np
np.random.seed(4012)

iris = load_iris()
X, y = iris.data, iris.target

model = ANNet(X, y, size=2, linout=False, max_iter=1000, trial=10)
y_pred = model.predict(X)
y_pred = np.round(y_pred)

from sklearn.metrics import classification_report
print(classification_report(y, y_pred))
print(pd.crosstab(y, y_pred))                # confusion matrix
print(model.coefs_)                          # get weights W(1),W(2)
print(model.intercepts_)                     # get bias b(1),b(2)
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      0.98      0.99        50
           2       0.98      1.00      0.99        50

    accuracy                           0.99       150
   macro avg       0.99      0.99      0.99       150
weighted avg       0.99      0.99      0.99       150

```

```
col_0   0   1   2
row_0
0      50   0   0
1       0  49   1
2       0   0  50
[array([[ -2.41208932, -19.85688561],
        [ -5.64902351, -20.31519222],
        [  8.45452575,  40.52004774],
        [  3.32476384,  34.40989985]]), array([[-19.62601053,  13.81189717,
5.32736104],
        [-11.87679237, -15.74239264,  28.40626703]])]
[array([ -2.0627694 , -72.78777118]), array([ 16.16841476,   7.84703132,
-23.80370557])]
```

Programme 7.4.11: Using logistic functions in `ANNET()` from Programme 7.4.8 with `lineout=False` for Iris flower dataset in Python.

The error rate is 0.67%, which is the same as in **R**.

Next, we illustrate its application with a 6-2-1 ANN for HSI financial ratio dataset in Programme 7.4.12. Note that the output variable is now binary, *i.e.* $K = 2$, and so the fitted values shown in the output can only be just one column, the probability of belonging to the group 1, as the sum of two outputs is always equal to 1.

```
> set.seed(4012)
> df <- read.csv("fin-ratio.csv")
> X <- df[,-ncol(df)]                    # define X
> y <- as.factor(df$HSI)                 # y as factor
>
> fin.nn <- ANNet(X, y, size=2, linout=F, maxit=200, try=10)
> fin.nn$value                           # best value
[1] 4.532677
> summary(fin.nn)
a 6-2-1 network with 17 weights
options were - entropy fitting
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1  i5->h1  i6->h1
   6.62   22.25   -6.02   -2.63   -2.02    2.64  -10.99
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2  i5->h2  i6->h2
4081.60 -191.34  190.01 -430.50   -4.79  -71.28   35.74
   b->o   h1->o   h2->o
   3.50  -54.86 -286.21
>
> pred <- (fin.nn$fit > 1/2)*1           # create label
> table(y, pred)                         # confusion matrix
   pred
```

```
y      0    1
  0 647    1
  1    0   32
```

Programme 7.4.12: Using logistic functions in `ANNET()` from Programme 7.4.6 with `lineout=F` for HSI dataset in **R**.

Here, although there are two classes, we only see one output neuron when using the logistic output with the argumement `linout=F`; meanwhile in Programme 7.4.10, the Iris flower dataset has three classes such that there are respectively three output neurons. `nnet()` transforms the two class classification into a binary classification, where the output value close to zero is predicted as the first class, and the output value close to one is predicted as the second class. In this binary classification, we only have one output neuron such that logistic function is used to give the predictive probability instead of the softmax function. We assign the label to `pred` according to the probability given in `fin.nn$fit`. Note that there is only 1 error case out of a total of 680 cases and the error rate is only 0.14%!

Similarly in Python,

```
import pandas as pd
import numpy as np
np.random.seed(4012)

df = pd.read_csv("fin-ratio.csv")
X = df.drop(columns="HSI")
y = df["HSI"]

model = ANNet(X, y, size=2, linout=False, max_iter=1000, trial=10)
y_pred = model.predict(X)
y_pred = np.round(y_pred)

from sklearn.metrics import classification_report
print(classification_report(y, y_pred))
print(pd.crosstab(y, y_pred))                # confusion matrix
print(model.coefs_)                          # get weights W(1),W(2)
print(model.intercepts_)                     # get bias b(1),b(2)
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       648
           1       0.94      0.94      0.94        32

    accuracy                           0.99       680
   macro avg       0.97      0.97      0.97       680
weighted avg       0.99      0.99      0.99       680

col_0    0   1
    HSI
0      646   2
```

```
1         2  30
[array([[-0.31501266, -0.16796956],
        [-1.43109197,  0.19882254],
        [ 8.50144384, -0.89176694],
        [-0.8658395 ,  0.02534074],
        [ 0.36385037, -0.02124511],
        [-0.03802064,  0.01619787]]), array([[  5.78551629],
        [-27.11697328]])]
[array([1.95013903, 8.43582161]), array([8.03068956])]
```

Programme 7.4.13: Using logistic functions in `ANNET()` from Programme 7.4.8 with `lineout=False` for HSI dataset in **R**.

We assign the label to `y_pred` according to the rounded values returned in the `model.predict()` function. Note that there are 4 error cases out of a total of 680 cases and the error rate is 0.588%!

7.4.2 Training and testing datasets

To fully understand how to make predictions on a testing dataset using a trained ANN, let us consider the Iris flower dataset again, and randomly choose 120 observations as training data, while the remaining 30 observations serve as testing data; again, we apply the same type of 4-2-1 ANN in Programme 7.4.14 in **R**.

```
> set.seed(4012)
> df <- iris
> classes <- factor(df$Species)
> levels(classes) <- 1:length(levels(classes))
> df$Species <- classes
> df <- apply(df, 2, as.numeric)
>
> id <- sample(1:nrow(df), size=round(0.7*nrow(df)))    # training:testing = 7:3
> df_train <- df[id,]
> X_train <- df_train[,-ncol(df)]
> y_train <- df_train[,ncol(df)]
> df_test <- df[-id,]
>
> # Using linear output option
> iris.nn <- ANNet(X_train, y_train, size=2, linout=T, maxit=200, try=10)
> iris.nn$value                                  # best value after 10 attempts
[1] 1.01439
> summary(iris.nn)
a 4-2-1 network with 13 weights
options were - linear output units
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1
  5.00   0.95   7.02 -10.65 -13.18
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2
110.11  -0.16  27.18 -10.61 -78.88
```

```
  b->o  h1->o  h2->o
  3.01  -1.02  -0.98
```

Programme 7.4.14: 4-2-1 ANN using `ANNet()` from Programme 7.4.6 with the training set of iris dataset in **R**

There is an array `wts` in `iris.nn` that contains all the weights in this ANN. We first create two matrices $\boldsymbol{W}^{(1)}$ (`W1`) and $\boldsymbol{W}^{(2)}$ (`W2`) containing the weights for the two respective layer transitions `i->h` and `h->o` as follows, see Programme 7.4.15. Here $\boldsymbol{W}^{(1)} = \left(\omega_{ji}^{(1)}\right)_{ji}$, where $i = 1, 2$, $j = 1, \cdots, 5$ such that $\omega_{1i}^{(1)}$ corresponds to the constant term (bias) for the linear combination leading from input to hidden layer node i, $\omega_{ji}^{(1)}$ for $j = 2, \cdots, 5$ is the coefficient for the input $\boldsymbol{x}_i$. Similarly, $\boldsymbol{W}^{(2)} = \left(\omega_{kj}^{(2)}\right)_{kj}$, where $j = 1, 2, k = 1$ such that $\omega_{1j}^{(2)}$ represents the constant term (bias), and $\omega_{kj}^{(2)}$ for $j = 1, 2$ stands for the coefficient of $a_j^{(1)}$ for the linear combination in the output layer. Actually, these weights are shown in Programme 7.4.14.

```
> # bias and weights matrix from i to h
> h1 <- matrix(iris.nn$wts[1:10], nrow=2, byrow=T)
> b1 <- h1[,1]
> W1 <- h1[,-1]
> b1
[1]   5.001692 110.108163
> W1
           [,1]       [,2]       [,3]       [,4]
[1,]  0.9487816  7.017607 -10.65388 -13.17935
[2,] -0.1556770 27.180194 -10.60718 -78.88091
>
> # bias and weights matrix from h to o
> h2 <- matrix(iris.nn$wts[11:13], nrow=1, byrow=T)
> b2 <- h2[,1]
> W2 <- h2[,-1]
> b2
[1] 3.005308
> W2
[1] -1.0212755 -0.9845398
```

Programme 7.4.15: ANN weights output for iris in **R**

Let us apply these weights to the test set `df_test`. First we need to define the logistic function in Programme 7.4.16:

```
> logistic <- function(x) {1/(1+exp(-x))}        # logistic function
> X_test <- df_test[,-ncol(df_test)]
> dim(X_test)
[1] 45  5
>
> # the output a^(1), t(·) is to transpose the matrix "X_test"
> a1 <- logistic(W1%*%t(X_test) + b1)
```

```
8 >
9 > a2 <- t(W2%*%a1 + b2)                          # compute fitted values
10 > pr <- round(a2)                                  # round to nearest integer
11 > y_test <- df_test[,ncol(df_test)]
12 > table(y_test, pr)                                # confusion matrix
13       pr
14 y_test  1  2  3
15      1 16  0  0
16      2  0 12  0
17      3  0  2 15
```

Programme 7.4.16: Predicting the testing data of iris dataset with the model in Programme 7.4.15 in **R**.

Here in the code line 2, we first create a feature matrix `X_test` from `df_test`, and then compute the output values of $\boldsymbol{a}^{(1)}$ as `a1`. Finally, we multiply `W2` to `a1` to obtain the final fitted values of $\boldsymbol{a}^{(2)}$.

Actually, as a comparison, the built-in function `predict()` will perform the same calculations as above and give the same prediction.

```
1 > pr <- predict(iris.nn, newdata=df_test)
2 > table(y_test, round(pr))                          # confusion matrix
3
4 y_test  1  2  3
5      1 16  0  0
6      2  0 12  0
7      3  0  2 15
```

Programme 7.4.17: Using `predict()` function as a comparison to Programme 7.4.15 in **R**.

Although we use linear output option in this illustration, similar commands can be used for logistic output option, and we leave the explanation as an exercise for readers.

```
1 > set.seed(4012)
2 > df <- iris
3 > classes <- factor(df$Species)
4 > levels(classes) <- 1:length(levels(classes))
5 > df$Species <- classes
6 > df <- apply(df, 2, as.numeric)
7 >
8 > id <- sample(1:nrow(df), size=round(0.7*nrow(df)))
9 > df_train <- df[id,]
10 > X_train <- df_train[,-ncol(df)]
11 > y_train <- as.factor(df_train[,ncol(df)])
12 > df_test <- df[-id,]
13 >
14 > iris.nn <- ANNet(X_train, y_train, size=2, linout=F, maxit=200, try=10)
15 > iris.nn$value
```

```
[1] 4.511579
> summary(iris.nn)
a 4-2-3 network with 19 weights
options were - softmax modelling
 b->h1  i1->h1  i2->h1  i3->h1  i4->h1
 -9.87  -18.55  -46.14   68.72   35.87
 b->h2  i1->h2  i2->h2  i3->h2  i4->h2
314.04  128.83  121.38 -234.43 -188.16
 b->o1  h1->o1  h2->o1
 43.67  -99.39   35.03
 b->o2  h1->o2  h2->o2
-40.74   66.97   38.66
 b->o3  h1->o3  h2->o3
 -2.72   32.45  -74.23
>
> # weights matrix from i to h
> h1 <- matrix(iris.nn$wts[1:10], nrow=2, byrow=T)
> b1 <- h1[,1]
> W1 <- h1[,-1]
> b1
[1]  -9.871155 314.044375
> W1
          [,1]      [,2]       [,3]       [,4]
[1,] -18.5492 -46.14499   68.72062   35.86595
[2,] 128.8335 121.37892 -234.43489 -188.15836
>
> # weights matrix from h to o
> h2 <- matrix(iris.nn$wts[11:length(iris.nn$wts)], nrow=3, byrow=T)
> b2 <- h2[,1]
> W2 <- h2[,-1]
> b2
[1]  43.670231 -40.738923  -2.722843
> W2
           [,1]      [,2]
[1,] -99.39455  35.03203
[2,]  66.96623  38.65856
[3,]  32.44675 -74.23262
>
> logistic <- function(x) {1/(1+exp(-x))}          # logistic function
> softmax <- function(x) {t(exp(x))/colSums(exp(x))} # softmax function
> X_test <- df_test[,-ncol(df_test)]
>
> a1 <- logistic(W1%*%t(X_test) + b1)              # the output a^{(1)}
> a2 <- softmax(W2%*%a1 + b2)        # compute fitted values
```

```
>
> y_pred <- max.col(a2)        # round to nearest integer
> y_test <- df_test[,ncol(df_test)]
> table(y_test, y_pred)
       y_pred
y_test  1  2  3
     1 16  0  0
     2  0 12  0
     3  0  0 17
>
> # find the column number of the max. fitted values
> prob <- predict(iris.nn, newdata=df_test)
> pred <- max.col(prob)
> table(y_test, pred)                              # confusion matrix
       pred
y_test  1  2  3
     1 16  0  0
     2  0 12  0
     3  0  0 17
```

Programme 7.4.18: Logistic ANN using only training dataset of iris dataset in Programme 7.4.10 in **R**.

Finally, we can try a dozen more attempts for the training model, so that the best possible ANN model is shown in Programme 7.4.18.

Similarly in Python,

```
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
np.random.seed(4012)

X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=4012)

model = ANNet(X_train, y_train, size=2, linout=True, max_iter=1000, trial=10)

W1, W2 = model.coefs_                          # get weights W(1),W(2)
b1, b2 = model.intercepts_                     # get bias b(1),b(2)
W1, W2 = W1.T, W2.T
print(W1)
print(W2)
print(b1)
print(b2)

def logistic(x):
    return 1/(1 + np.exp(-x))
```

```
23
24 a1 = logistic(W1 @ X_test.T + b1.reshape(-1, 1))
25 a2 = W2 @ a1 + b2.reshape(-1, 1)
26 pr = np.round(a2[0])
27 print(pd.crosstab(y_test, pr))                    # confusion matrix
28
29 y_pred = model.predict(X_test)
30 y_pred = np.round(y_pred)
31
32 from sklearn.metrics import classification_report
33 print(classification_report(y_test, y_pred))
34 print(pd.crosstab(y_test, y_pred))                # confusion matrix
```

```
1  [[  2.43988048  -9.14827525   6.75336869   0.9658007 ]
2  [-16.97401103  -5.37363846  23.64004407  19.82841178]]
3  [[0.99343003 0.98580338]]
4  [ -5.88675064 -26.73782679]
5  [-0.00031438]
6  col_0  -0.0   1.0   2.0
7  row_0
8  0         14     0     0
9  1          0    15     2
10 2          0     0    14
11                precision    recall  f1-score   support
12
13             0       1.00      1.00      1.00        14
14             1       1.00      0.88      0.94        17
15             2       0.88      1.00      0.93        14
16
17     accuracy                           0.96        45
18    macro avg       0.96      0.96      0.96        45
19 weighted avg       0.96      0.96      0.96        45
20
21 col_0  -0.0   1.0   2.0
22 row_0
23 0         14     0     0
24 1          0    15     2
25 2          0     0    14
```

Programme 7.4.19: Linear 4-2-1 ANN using `ANNet()` function from Programme 7.4.8 with the training set of iris dataset in Python

Here the biases $\boldsymbol{b}^{(1)}$ and $\boldsymbol{b}^{(2)}$ are respectively vector and scalar. In order to meet the dimension of $\boldsymbol{Z}^{(1)} = (\boldsymbol{z}_1^{(1)}, \cdots, \boldsymbol{z}_N^{(1)})$ and $\boldsymbol{Z}^{(2)} = (\boldsymbol{z}_1^{(2)}, \cdots, \boldsymbol{z}_N^{(2)})$, we reshape the biases $\boldsymbol{b}^{(1)}$ and $\boldsymbol{b}^{(2)}$ N into a column vector by `.reshape(-1, 1)`. The accuracy rate is $(14 + 15 + 14)/45 = 95.56\%$.

Next, consider the performance of ANN for HSI financial ratio dataset by randomly select 544 records as a training dataset. Programme 7.4.20 is the illustration.

```
> set.seed(4012)
> df <- read.csv("fin-ratio.csv")
> id <- sample(1:nrow(df), size=round(0.7*nrow(df)))    # training:testing = 7:3
> df_train <- df[id,]
> X_train <- df_train[,-ncol(df)]
> y_train <- as.factor(df_train$HSI)
> df_test <- df[-id,]
> X_test <- df_test[,-ncol(df)]
> y_test <- as.factor(df_test$HSI)
> fin.nn <- ANNet(X_train, y_train, size=2, linout=F, maxit=200, try=20)
> fin.nn$value                          # best value after 20 attempts
[1] 4.238618
> summary(fin.nn)
a 6-2-1 network with 17 weights
options were - entropy fitting
a 6-2-1 network with 17 weights
options were - entropy fitting
  b->h1  i1->h1  i2->h1  i3->h1  i4->h1  i5->h1  i6->h1
   0.48   -2.79  -29.92   -2.41   -2.03    5.30    0.31
  b->h2  i1->h2  i2->h2  i3->h2  i4->h2  i5->h2  i6->h2
3915.94  -40.63   88.14 -416.02    0.36  -77.66   26.08
   b->o   h1->o   h2->o
   3.22  -31.28 -761.74
>
> pred <- (fin.nn$fit > 1/2)*1
> # confusion matrix for training dataset
> table(y_train, pred)
        pred
y_train   0   1
      0 450   1
      1   0  25
>
> h1 <- matrix(fin.nn$wts[1:14], nrow=2, byrow=T)
> b1 <- h1[,1]
> W1 <- h1[,-1]
> b1
[1]    0.4836542 3915.9363465
> W1
           [,1]      [,2]         [,3]       [,4]      [,5]       [,6]
[1,]  -2.78610 -29.92035   -2.412327 -2.0312696   5.30143  0.3120672
[2,] -40.63106  88.14350 -416.019941  0.3632691 -77.66332 26.0834504
>
```

```
> # weights matrix from h to o
> h2 <- matrix(fin.nn$wts[15:length(fin.nn$wts)], nrow=1, byrow=T)
> b2 <- h2[,1]
> W2 <- h2[,-1]
> b2
[1] 3.218877
> W2
[1]  -31.27854 -761.74068
>
> logistic <- function(x) {1/(1+exp(-x))}          # logistic function
> X_test <- df_test[,-ncol(df_test)]
>
> a1 <- logistic(W1%*%t(X_test) + b1)              # the output a^{(1)}
> a2 <- logistic(W2%*%a1 + b2)       # logistc used instead of softmax
>
> y_pred <- (a2 > 1/2)*1            # round to nearest integer
> y_test <- df_test[,ncol(df_test)]
> table(y_test, y_pred)
       y_pred
y_test   0   1
     0 193   4
     1   0   7
>
> pr <- predict(fin.nn, df_test)
> pred <- (pr > 1/2)*1
> # classification table for testing data df_test
> table(df_test$HSI, pred)                                  # confusion matrix
   pred
      0   1
  0 193   4
  1   0   7
>
> write.csv(df_train, "fin-ratio_train.csv")
> write.csv(df_test, "fin-ratio_test.csv")
```

Programme 7.4.20: Logistic ANN using only training dataset of HSI dataset in Programme 7.4.12 in **R**.

As aforementioned in Programme 7.4.12, logistic function is adopted instead of the softmax function for binary output neuron. We save the training dataset to `fin-ratio_train.csv` and the testing dataset to `fin-ratio_test.csv` for Python as a comparison.

Similarly in Python, we first import the training and testing dataset saved in Programme 7.4.20.

```
import pandas as pd
import numpy as np
np.random.seed(560)
```

```

df_train = pd.read_csv("fin-ratio_train.csv", index_col=0)
df_test = pd.read_csv("fin-ratio_test.csv", index_col=0)
X_train, X_test = df_train.drop(columns="HSI"), df_test.drop(columns="HSI")
y_train, y_test = df_train["HSI"], df_test["HSI"]

model = ANNet(X_train, y_train, size=2, linout=False, max_iter=1000, trial=20)

print(model.coefs_)                 # get weights W(1),W(2)
print(model.intercepts_)            # get bias b(1),b(2)

# training dataset
y_pred = model.predict(X_train)
y_pred = np.round(y_pred)

from sklearn.metrics import classification_report
print(classification_report(y_train, y_pred))
print(pd.crosstab(y_train, y_pred))# confusion matrix

# testing dataset
y_pred = model.predict(X_test)
y_pred = np.round(y_pred)

print(classification_report(y_test, y_pred))
print(pd.crosstab(y_test, y_pred))          # confusion matrix
```

```
[array([[ 3.64303551e+00,  4.79101091e-01],
        [-1.76092566e+01,  4.63201386e+00],
        [-3.47047873e+00, -1.13760666e+01],
        [-1.94337259e+00, -5.21283548e-01],
        [-5.17446129e+00, -2.53589301e+00],
        [-3.27727343e-03,  2.33087822e-01]]), array([[-31.96316572],
        [-68.57502685]])]
[array([  1.83924971, 106.9534205 ]), array([3.38653779])]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       451
           1       0.96      1.00      0.98        25

    accuracy                           1.00       476
   macro avg       0.98      1.00      0.99       476
weighted avg       1.00      1.00      1.00       476

col_0    0   1
HSI
0      450   1
1        0  25
              precision    recall  f1-score   support
```

```
23
24              0       1.00      0.97      0.99       197
25              1       0.58      1.00      0.74         7
26
27     accuracy                           0.98       204
28    macro avg       0.79      0.99      0.86       204
29 weighted avg       0.99      0.98      0.98       204
30
31 col_0     0  1
32 HSI
33 0       192  5
34 1         0  7
```

Programme 7.4.21: Logistic ANN using only training dataset of HSI dataset in Programme 7.4.13 in Python.

Finally, it is almost the case that the coefficients of the weights and biases computed in **R** and Python should be different. However, we can still study the generated system in a qualitative manner, so as to reconcile one by **R** to that by Python. The following discussion is one representative illustration. To facilitate a better comparison with the outputs from Programme 7.4.20 in **R** with that for Python, a chosen random seed of `4012` with the argument `set.seed(4012)` is used in **R** in Programme 7.4.20 and the random seed `560` with the argument `np.random.seed(560)` is used in Python in Programme 7.4.21. The 6-2-1 ANN model coefficients of **R** and Python are summarized into the Table 7.4.1 below:

Hidden Layer 1						**Output Layer**		
Neuron 1	**R**	Python	Neuron 2	**R**	Python	Neurons	**R**	Python
$b_1^{(1)}$	0.48	1.8392	$b_2^{(1)}$	3915.94	106.9534	$b_1^{(2)}$	3.22	3.3865
$\omega_{11}^{(1)}$	-2.79	3.6430	$\omega_{12}^{(1)}$	-40.63	0.4791	$\omega_{11}^{(2)}$	-31.28	-31.9632
$\omega_{21}^{(1)}$	-29.92	-17.6093	$\omega_{22}^{(1)}$	88.14	4.6320	$\omega_{12}^{(2)}$	-761.74	-68.5750
$\omega_{31}^{(1)}$	-2.41	-3.4705	$\omega_{32}^{(1)}$	-416.02	-11.3761			
$\omega_{41}^{(1)}$	-2.03	-1.9434	$\omega_{42}^{(1)}$	0.36	-0.5213			
$\omega_{51}^{(1)}$	5.3	-5.1745	$\omega_{52}^{(1)}$	-77.66	-2.5359			
$\omega_{61}^{(1)}$	0.31	-0.003277	$\omega_{62}^{(1)}$	26.08	2.3309			

Table 7.4.1: 6-2-1 ANN model coefficients of **R** in Programme 7.4.20 and Python in Programme 7.4.21.

As illustrated in Table 7.4.1, all coefficients for $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}$ and those for biases for $\boldsymbol{b}^{(1)}$ and $\boldsymbol{b}^{(2)}$ are much larger as computed in **R** generated by the `nnet()` function than those generated by `MLPClassifier()` in Python. Although the confusion matrices for both training dataset and the testing dataset in Python and **R** are similar, it seems that the `nnet()` model built in **R** suffers more from overfitting than the `MLPClassifier()` model built in Python. Moreover, for the output layer, both bias $\boldsymbol{b}^{(2)}$ in **R** with $\boldsymbol{b}^{(1)} = (3.22)$ and Python with $\boldsymbol{b}^{(1)} = (3.38653779)$, respectively, are close in value; meanwhile the respective weights $\boldsymbol{W}^{(2)}$ in **R** with $\boldsymbol{W}^{(2)} = (-31.28, -761.74)$ and Python with $\boldsymbol{W}^{(2)} = (-31.96316572, -68.57502685)$ share

the same property of having heavier second weight component. Moreover, in the first neuron of the hidden layer, both **R** and Python have their maximum value in magnitude occuring at the second weight $\omega_{21}^{(1)}$ (-29.92 in **R** versus -17.61 in Python) and the least value in magnitude can be found at the last weight $\omega_{61}^{(1)}$ (0.31 in **R** versus 0.003277 in Python); meanwhile for the second neuron of the hidden layer, the largest value in magnitude occurs at the third weight $\omega_{32}^{(1)}$ (-416.02 in **R** versus -11.376 in Python), the second largest value in magnitude is at the second weight $\omega_{22}^{(1)}$ (88.14 in **R** versus 4.632 in Python), and the third largest value in magnitude appears in the fifth weight $\omega_{52}^{(1)}$ (-77.66 in **R** versus -2.5359 in Python). Therefore, we believe that the most neurons built by **R** and those generated by Python are functioning in an equivalent manner.

7.5 Universal Approximation Theorem

As we have seen in the previous discussions, the input to each neuron is a weighted average of the output of those in the preceding layer. This weighted average is further fed into a non-linear activation function for so as to introduce non-linearity to the data. One common example of activation functions is a *sigmoid function*, which is an S-shaped monotonically increasing (not necessarily differentiable) function defined for all real numbers, which ranges usually in $(0, 1)$ or $(-1, 1)$; the logistic function and the hyperbolic tangent function are typical examples of sigmoid functions; also see Figure 7.2.3.

In practice, we often have to deal with feature variables and the labels, where the relations from the first batch to the latter are often described by very complicated functions. Like building a LEGO model, an ANN models complex functions by combining very simple building blocks (*e.g.* the sigmoid functions) in an effective way, so that there are interactions among neurons from one layer to another; indeed, with a certain number of hidden layers should be able to approximate many quite arbitrary functions from the inputs to the outputs, and this is ensured by the *Universal Approximation Theorem*.

Theorem 7.5.1. Universal Approximation Theorem: Consider a feed-forward network with a single hidden layer containing a finite number d_1 of neurons with a fixed choice of sigmoid activation function, which is bounded and non-constant on $\mathbb{R}$, and for an arbitrary $\varepsilon > 0$, then for any continuous function f on a compact subset of $\mathbb{R}$, then there is a $d_1 \in \mathbb{Z}^+$ such that the maximum possible absolute difference of the network generated and f is less than ε.

As a motivation, we first sketch the main idea of its proof. Consider a simple example where both the input x and output y are in $\mathbb{R}$, so that the relation between x and y is described by a function $y = f(x)$ as depicted in Figure 7.5.1. We divide the function into a number of partitions where each part is represented by a simple step function, see Figure 7.5.2, one bar column corresponds to one simple function. The combination of these simple functions then approximates the function f, and the approximation tends to be more accurate as the number of partitions M increases.

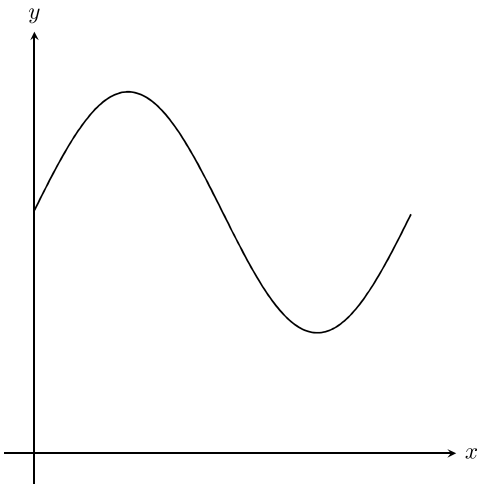


Figure 7.5.1: The function $y = f(x)$.

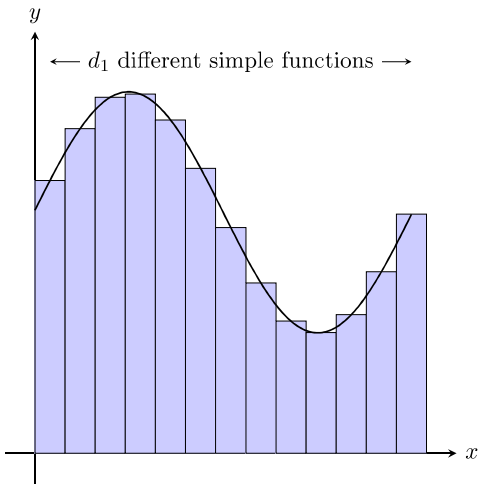


Figure 7.5.2: Partitioning the function f.

Figure 7.5.3

To see why using activation function suffices, we first consider two rectilinear sigmoid functions $h_1, h_2 : \mathbb{R} \to \mathbb{R}$ such that for a small $\delta > 0$, h_1 stays at 0 for all $x \leq -\delta$ and then it rises linearly but sharply to 1 at $x = -\delta/2$ and remains constant thereafter; similarly, h_2 stays at 0 for all $x \leq \delta/2$, and it reaches 1 linearly at δ and remains constant thereafter. The function $h = h_1 - h_2$ then approximately gives us a rectangular tower, the simple function as desired.

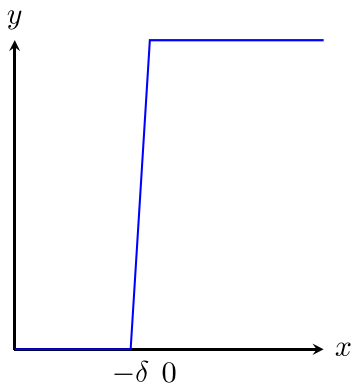


Figure 7.5.4: $h_1(x)$.

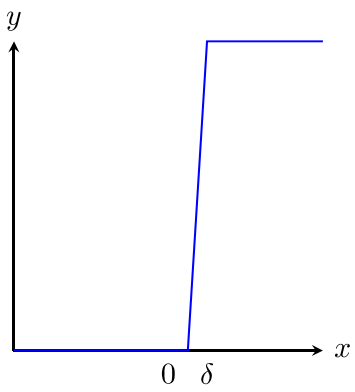


Figure 7.5.5: $h_2(x)$.

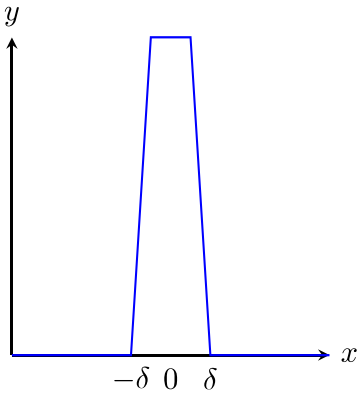


Figure 7.5.6: $h(x)$.

Figure 7.5.7: Construction of rectangles.

Hence, with an input x for two rectilienar sigmoid activation functions h_1 and h_2, if we apply the weights $+1$ and -1 to the respective output, we can obtain the desired rectangle-shaped function. By combining a multiple number of similar component functions, see Figure 7.5.8, we can approximate a complicated function f, and this illustrates the *representational power* of ANNs and even DNNs.

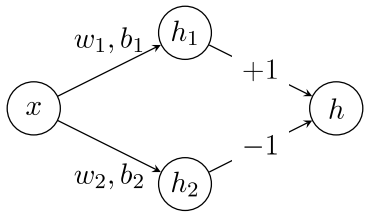


Figure 7.5.8: A component function for constructing a step-wise approximation for f in Figure 7.5.2.

7.6 Training an ANN using Backpropagation

As seen from the **R** or Python outputs in Section 7.4, training the neural network is to find the appropriate matrices of weights $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$ such that the error function $\mathcal{E} = (\boldsymbol{y} - \boldsymbol{a}^{(2)})^\top(\boldsymbol{y} - \boldsymbol{a}^{(2)})/2$ is minimized by using Back-propagation algorithm [4][2]. Let us illustrate this through a simple 2-2-1 ANN example with a logistic output as depicted in Figure 7.6.1.

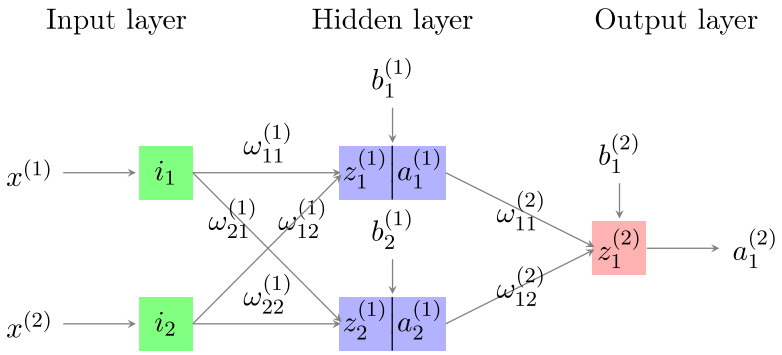


Figure 7.6.1: Illustration of a 2-2-1 ANN.

Suppose that we have the dataset $\mathcal{S} = \{(\boldsymbol{x}_n, y_n)\}_{n=1}^4$, each observation $\boldsymbol{x}_n$ consists of 2 feature variables, and the feature matrix is $\boldsymbol{X} := (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4)^\top \in \mathbb{R}^{N\times D}$ and further define $\boldsymbol{x}_n = (x_n^{(1)}, x_n^{(2)})$ for $n = 1, 2, 3, 4$, where

$$\boldsymbol{X} := (\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4)^\top = \begin{pmatrix} 0.4 & 0.7 \\ 0.8 & 0.9 \\ 1.3 & 1.8 \\ -1.3 & -0.9 \end{pmatrix} \quad \text{and} \quad \boldsymbol{y} := \begin{pmatrix} y_1 & y_2 & y_3 & y_4 \end{pmatrix}^\top = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}.$$

And suppose that the initial weights $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$ of the ANN model is randomly set as:

$$\boldsymbol{W}^{(1,0)} = \begin{pmatrix} \omega_{11}^{(1,0)} & \omega_{12}^{(1,0)} \\ \omega_{21}^{(1,0)} & \omega_{22}^{(1,0)} \end{pmatrix} = \begin{pmatrix} -0.2 & 0.1 \\ 0.2 & 0.9 \end{pmatrix} \quad \text{and} \quad \boldsymbol{W}^{(2,0)} = \begin{pmatrix} \omega_{11}^{(2,0)} & \omega_{12}^{(2,0)} \end{pmatrix} = \begin{pmatrix} -0.5 & 0.1 \end{pmatrix},$$

[2]Geoffry Hinton is the great-great-grandson of the well-known logician and mathematician George Boole. In an interview, Hinton said that "*David E. Rumelhart came up with the basic idea of backpropagation, so it's his invention*". In collaboration with his PhD students Alex Krizhevsky and Ilya Sutskever, they invented the AlexNet for the ImageNet Large Scale Visual Recognition Challenge on September 30, 2012. The AlexNet was a breakthrough in image recognition, it achieved a top-5 (Identifying 5 different objects in a photo, if the learner is able to recognize any one of them, the identification is said to be successful.) error of 15.3%, which is 10.8% lower than that of the runner up.

where the superscript $(\ell, 0)$ means that the layer ℓ and the iteration $t = 0$, and the bias $\boldsymbol{b}^{(1,0)}$ and $b_1^{(2,0)}$:

$$\boldsymbol{b}^{(1,0)} = \begin{pmatrix} b_1^{(1,0)} \\ b_2^{(1,0)} \end{pmatrix} = \begin{pmatrix} 0.1 \\ 0.4 \end{pmatrix} \qquad \text{and} \qquad b_1^{(2,0)} = 0.2\,.$$

First, we compute the output from the ANN using the initial weights as in Programme 7.6.1.

```
> logistic <- function(x) {1/(1+exp(-x))}
>
> X <- matrix(c(0.4,0.8,1.3,-1.3,0.7,0.9,1.8,-0.9), ncol=2)
> y <- c(0,0,1,0)
>
> # transpose to fit the input and output format of ANN
> X <- t(X)
>
> # hidden layer weights W^(1) and bias b^(1)
> W1 <- matrix(c(-0.2,0.2,0.1,0.9), nrow=2)
> b1 <- c(0.1,0.4)
> # output layer weights W^(2) and bias b^(2)
> W2 <- matrix(c(-0.5,0.1), nrow=1)
> b2 <- 0.2
>
> a1 <- logistic(W1%*%X + b1)                    # compute a^(1)
> a2 <- logistic(W2%*%a1 + b2)                   # output value
> E <- y - a2                                        # output error
> SSE <- sum(E^2)                                    # SSE
> E                                                  # y - a^(2)
           [,1]       [,2]     [,3]       [,4]
[1,] -0.5034926 -0.5064967 0.490446 -0.4875784
> SSE / 2                                            # error function E
[1] 0.4941568
```

Programme 7.6.1: Forward-propagation of 2-2-1 ANN in **R**.

In Programme, we update the output layer weights `W2` and hidden layer weights `W1` by working backward, and this is a two-step illustration of Backpropagation algorithm, its mathematical explanation is put in Subsection 7.6.1.

```
> eta <- 0.5                                       # learning rate η
> N <- length(y)
> # δ^(2) = (y - a^(2)) ⊙ a^(2) ⊙ (1 - a^(2)), see (7.6.4)
> delta2 <- E * a2 * (1 - a2)
> # ΔW^(2) = η δ^(2) (a^(1))^T, see (7.6.5)
> Delta_W2 <- eta * delta2%*%t(a1) / N
> # Δb^(2) = η δ^(2), see (7.6.6)
> Delta_b2 <- eta * delta2 / N
> new_W2 <- W2 + Delta_W2
> new_b2 <- b2 + rowSums(Delta_b2)
```

```
>
> # δ(1) = {(W(2))⊤δ(2)} ⊙ a(1) ⊙ (1 − a(1)), see (7.6.7)
> delta1 <- (t(W2)%*%delta2) * a1 * (1 - a1)     # hidden layer δ(1)
> # ΔW(1) = ηδ(1)x⊤, see (7.6.8)
> Delta_W1 <- eta * delta1%*%X / N
> # Δb(1) = ηδ(1), see (7.6.9)
> Delta_b1 <- eta * delta1 / N
> new_W1 <- W1 + Delta_W1
> new_b1 <- b1 + rowSums(Delta_b1)
>
> new_a1 <- logistic(new_W1%*%X + new_b1)
> new_a2 <- logistic(new_W2%*%new_a1 + new_b2)
> E <- y - new_a2
>
> # new sum of square error
> SSE <- sum(E^2)
> SSE / 2
[1] 0.4820338
```

Programme 7.6.2: Backward-propagation of 2-2-1 ANN in **R**.

Here the `rowSums()` function sums over all the gradients with respect to the bias for the batch Gradient Descent method. The revised new weights `new_W1` and `new_W2` are computed, and now the SSE decreases from 0.988 to 0.833. We can update the `W1` by `new_W1` and `W2` by `new_W2` and iterate until SSE is smaller than a preassigned threshold or the maximum number of iteration set in advance was exceeded. Besides, we can scale up the learning rate, let say `eta <- 1.1*eta`, at the next iteration if the next SSE reduces or scale down the learning rate, let say `eta <- 0.5*eta`, if the next SSE raises up.

Similarly in Python, we have

```
import numpy as np
def logistic(x):
    return 1/(1 + np.exp(-x))

X = np.array([[0.4,0.7], [0.8, 0.9], [1.3, 1.8], [-1.3, -0.9]])
y = np.array([0,0,1,0])

# transpose to fit the input format of ANN
X = X.T

# hidden layer weights W(1) and bias b(1)
W1 = np.array([[-0.2,0.1], [0.2,0.9]])
b1 = np.array([0.1, 0.4])
# output layer weights W(2) and bias b(2)
W2 = np.array([[-0.5,0.1]])
b2 = np.array([0.2])

a1 = logistic(W1 @ X + b1.reshape(-1, 1))           # compute a(1)
```

```
a2 = logistic(W2 @ a1 + b2.reshape(-1, 1))          # output value
E = y - a2                                          # output error
SSE = np.sum(E**2)                                  # SSE
print(E)                                            # y - a^(2)
print(SSE / 2)
```

```
[[-0.50349257 -0.50649671  0.49044597 -0.48757836]]
0.4941567943022961
```

Programme 7.6.3: Forward-propagation of 2-2-1 ANN in Python.

The two-step Backpropagation algorithm in Python is shown in Programme 7.6.4 below:

```
eta = 0.5                                           # learning rate η
N = len(y)
# δ^(2) = (y - a^(2)) ⊙ a^(2) ⊙ (1 - a^(2)), see (7.6.4)
delta2 = E*a2*(1 - a2)
# ΔW^(2) = η δ^(2) (a^(1))^T, see (7.6.5)
Delta_W2 = eta * delta2 @ a1.T / N
# Δb^(2) = η δ^(2), see (7.6.6)
Delta_b2 = eta * delta2 / N
new_W2 = W2 + Delta_W2
new_b2 = b2 + np.sum(Delta_b2, axis=-1)

# δ^(1) = {(W^(2))^T δ^(2)} ⊙ a^(1) ⊙ (1 - a^(1)), see (7.6.7)
delta1 = (W2.T @ delta2)*a1*(1 - a1)
# ΔW^(1) = η δ^(1) x^T, see (7.6.8)
Delta_W1 = eta * delta1 @ X.T / N
# Δb^(1) = η δ^(1), see (7.6.9)
Delta_b1 = eta * delta1 / N
new_W1 = W1 + Delta_W1
new_b1 = b1 + np.sum(Delta_b1, axis=-1)

new_a1 = logistic(new_W1 @ X + new_b1.reshape(-1, 1))
new_a2 = logistic(new_W2 @ new_a1 + new_b2.reshape(-1, 1))
E = y - new_a2
# new sum of square error
SSE = np.sum(E**2)
print(E)
print(SSE / 2)
```

```
[[-0.49038343 -0.49332007  0.50394286 -0.47567705]]
0.48203382672659223
```

Programme 7.6.4: Backward-propagation of 2-2-1 ANN in Python.

Here the `sum()` function in the `numpy` library sums over all the gradients with respect to the bias for the batch Gradient Descent method.

7.6.1 Mathematical Principle behind Backpropagation algorithm

Let us consider a d_0-d_1-d_2 ANN with the logistic activation function $\boldsymbol{f}_1$ and a logistic output with activation function $\boldsymbol{f}_2$. Denote the linear combinations and neuron output of the Input to Hidden layer and the Hidden

layer to Output respectively by:

$$z_j^{(1)} = b_j^{(1)} + \sum_{i=1}^{d_0} \omega_{ji}^{(1)} x^{(i)}\,, \quad a_j^{(1)} = f_1(z_j^{(1)})\,, \quad j = 1, \cdots, d_1\,, \tag{7.6.1}$$

and

$$z_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{d_1} \omega_{kj}^{(2)} a_j^{(1)}\,, \quad a_k^{(2)} = f_2(z_k^{(2)})\,, \quad k = 1, \cdots, d_2\,. \tag{7.6.2}$$

The objective error function needs to minimize is

$$\frac{1}{2N} \sum_{n=1}^{N} (\boldsymbol{y}_n - \boldsymbol{a}_n^{(2)})^\top (\boldsymbol{y}_n - \boldsymbol{a}_n^{(2)})\,.$$

Yet, for the sake of discussion, we first consider one sigle summand and also omit the subscript n:

$$\mathcal{E} = \frac{1}{2} (\boldsymbol{y} - \boldsymbol{a}^{(2)})^\top (\boldsymbol{y} - \boldsymbol{a}^{(2)}) = \frac{1}{2} \sum_{k=1}^{d_2} (y_k - a_k^{(2)})^2\,; \tag{7.6.3}$$

recall that as the activation function $f(x)$ is logistic, we have a very useful relation: $f'(x) = f(x)[1 - f(x)]$, for all $x \in \mathbb{R}$. If batch Gradient Descent method is used, we should take an average of all the delta change terms Δ over all the N observations to obtain the final Δ, *i.e.* $\Delta = \sum_{n=1}^{N} (\Delta)_n / N$. These concepts of forward and backward propagation will be discussed further in Subsection 8.1.1, together with all variants of Gradient Descent methods.

The Backpropagation algorithm is a Gradient Descent method with variable-step length for minimizing the error function $\mathcal{E}$, and its procedure can be described as follows. We first note that:

1. By the Law of total derivative, we see that by regarding $\boldsymbol{W}^{(1)}$ as a long vector formed by pasting the consecutive column vectors, for $i = 1, 2$,
$$\Delta\mathcal{E} = (\nabla_{\boldsymbol{W}^{(1)}} \mathcal{E})^\top \Delta \boldsymbol{W}^{(1)} + (\nabla_{\boldsymbol{W}^{(2)}} \mathcal{E})^\top \Delta \boldsymbol{W}^{(2)}\,;$$
2. According to Gradient Descent method, the direction of $\left(\left(\Delta \boldsymbol{W}^{(1)}\right)^\top, \left(\Delta \boldsymbol{W}^{(2)}\right)^\top \right)^\top$ that can maximize $\Delta\mathcal{E}$ is along the vector $\left((\nabla_{\boldsymbol{W}^{(1)}} \mathcal{E})^\top, (\nabla_{\boldsymbol{W}^{(2)}} \mathcal{E})^\top \right)^\top$, that is the reason why the method is called steepest descent.

The general algorithm for updating $\boldsymbol{W}^{(\ell)}$ is

$$\boldsymbol{W}^{(\ell,t+1)} = \boldsymbol{W}^{(\ell,t)} + \Delta \boldsymbol{W}^{(\ell,t)}, \quad \text{such that} \quad \Delta \boldsymbol{W}^{(\ell,t+1)} = -\eta \left(\nabla_{\boldsymbol{W}^{(\ell)}} \mathcal{E} \right) \Big|_{\boldsymbol{W} = \boldsymbol{W}^{(t)}}\,, \qquad \ell = 1, 2\,,$$

where $\boldsymbol{W}^{(t)} = \left(\left(\boldsymbol{W}^{(1,t)}\right)^\top, \left(\boldsymbol{W}^{(2,t)}\right)^\top \right)^\top$ and η is the hyperparameter learning rate which is kept at a low level, so that no exaggerated step size would be taken. Now, $\Delta \boldsymbol{W}^{(2)} = \eta \boldsymbol{\delta}^{(2)} \left(\boldsymbol{a}^{(1)}\right)^\top$, where

$$\boldsymbol{\delta}^{(2)} = (\boldsymbol{y} - \boldsymbol{a}^{(2)}) \odot \boldsymbol{a}^{(2)} \odot (\boldsymbol{1} - \boldsymbol{a}^{(2)})\,; \tag{7.6.4}$$

to see this, we first compute the gradient of $\mathcal{E}$ with respect to the weights $\boldsymbol{W}^{(2)}$ from $\boldsymbol{W}^{(2)} = \boldsymbol{W}^{(2,t)}$, (7.6.2), and (7.6.3) (from hidden layer to output): For $j = 1, \cdots, d_1$ and $k = 1, \cdots, d_2$,

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial \omega_{kj}^{(2)}} &= \frac{\partial \mathcal{E}}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial \omega_{kj}^{(2)}} = \frac{\partial \mathcal{E}}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial \omega_{kj}^{(2)}} \\ &= -(y_k - a_k^{(2)}) \cdot f_2(z_k^{(2)})(1 - f_2(z_k^{(2)})) \cdot a_j^{(1)} \\ &= -\delta_k^{(2)} \cdot a_j^{(1)}\,. \end{aligned}$$

Similarly the bias $\boldsymbol{b}^{(2)}$ is

$$\frac{\partial \mathcal{E}}{\partial b_k^{(2)}} = \frac{\partial \mathcal{E}}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial b_k^{(2)}} = -\delta_k^{(2)}$$

In other words, we have

$$\nabla_{\boldsymbol{W}^{(2)}} \mathcal{E} = -\boldsymbol{\delta}^{(2)} \left(\boldsymbol{a}^{(1)}\right)^\top \qquad \Rightarrow \qquad \Delta \boldsymbol{W}^{(2)} = -\eta \nabla_{\boldsymbol{W}^{(2)}} \mathcal{E} = \eta \boldsymbol{\delta}^{(2)} \left(\boldsymbol{a}^{(1)}\right)^\top , \tag{7.6.5}$$

where $\boldsymbol{a}^{(1)} = (a_1^{(1)}, \cdots, a_{d_1}^{(1)})^\top \in \mathbb{R}^{d_1}$, and

$$\nabla_{\boldsymbol{b}^{(2)}} \mathcal{E} = -\boldsymbol{\delta}^{(2)} \qquad \Rightarrow \qquad \Delta \boldsymbol{b}^{(2)} = -\eta \nabla_{\boldsymbol{b}^{(2)}} \mathcal{E} = \eta \boldsymbol{\delta}^{(2)} , \tag{7.6.6}$$

while for our example, we only have $\boldsymbol{b}^{(2)} = (b_1^{(2)})$ and $\boldsymbol{\delta}^{(2)} = (\delta_1^{(2)})$ such that $\Delta b_1^{(2)} = \eta \delta_1^{(2)}$. Next, we also have $\Delta \boldsymbol{W}^{(1)} = \eta \boldsymbol{\delta}^{(1)} \boldsymbol{x}^\top$, where $\boldsymbol{\delta}^{(1)}$ is defined as 7.6.7 as below; to see this, finally, we first compute the sensitivity of $\mathcal{E}$ with respect to $z_j^{(1)}$, $2\boldsymbol{\delta}^{(1)}$, using the chain rule for differentiation:

$$\delta_j^{(1)} := \frac{\partial \mathcal{E}}{\partial z_j^{(1)}} = \sum_{k=1}^{d_2} \frac{\partial \mathcal{E}}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial z_j^{(1)}} = \sum_{k=1}^{d_2} \delta_k^{(2)} \cdot \omega_{kj}^{(2)} \cdot f_1(z_j^{(1)})(1 - f_1(z_j^{(1)}))$$

or in matrix notation,

$$\boldsymbol{\delta}^{(1)} = \begin{pmatrix} \delta_1^{(1)} & \cdots & \delta_{d_1}^{(1)} \end{pmatrix}^\top := \left\{ (\boldsymbol{W}^{(2)})^\top \boldsymbol{\delta}^{(2)} \right\} \odot \boldsymbol{a}^{(1)} \odot (\boldsymbol{1} - \boldsymbol{a}^{(1)}) , \tag{7.6.7}$$

and so with which the gradient of $\mathcal{E}$ wil respect to $\omega_{ji}^{(1)}$ from $\boldsymbol{W}^{(1)}$ (from input to hidden layer) is given by:

$$\frac{\partial \mathcal{E}}{\partial \omega_{ji}^{(1)}} = \frac{\partial \mathcal{E}}{\partial z_j^{(1)}} \frac{\partial z_j^{(1)}}{\partial \omega_{ji}^{(1)}} = -\delta_j^{(1)} \cdot x^{(i)} .$$

In other words, we have

$$\nabla_{\boldsymbol{W}^{(1)}} \mathcal{E} = -\boldsymbol{\delta}^{(1)} \boldsymbol{x}^\top \qquad \Rightarrow \qquad \Delta \boldsymbol{W}^{(1)} = -\eta \nabla_{\boldsymbol{W}^{(1)}} \mathcal{E} = \eta \boldsymbol{\delta}^{(1)} \boldsymbol{x}^\top ; \tag{7.6.8}$$

similarly, with respect to the bias $\boldsymbol{b}^{(1)}$, the partial derivative of $\mathcal{E}$ is

$$\frac{\partial \mathcal{E}}{\partial b_j^{(1)}} = \frac{\partial \mathcal{E}}{\partial z_j^{(1)}} \frac{\partial z_j^{(2)}}{\partial b_j^{(1)}} = -\delta_j^{(1)} .$$

Hence, we further have

$$\nabla_{\boldsymbol{b}^{(1)}} \mathcal{E} = -\boldsymbol{\delta}^{(1)} \qquad \Rightarrow \qquad \Delta \boldsymbol{b}^{(1)} = \eta \boldsymbol{\delta}^{(1)} . \tag{7.6.9}$$

Finally, Programme 7.6.1 and Programme 7.6.3 use the batch gradient descent, in which we have $\Delta = \sum_{n=1}^{N} (\Delta)_n / N$.

BIBLIOGRAPHY

[1] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6), 386.

[2] McCulloch, W. S., and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4), 115-133.

[3] Minsky, M., and Papert, S. (1969). Perceptron: an introduction to computational geometry. Cambridge tiass., HIT.

[4] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. nature, 323(6088), 533-536.