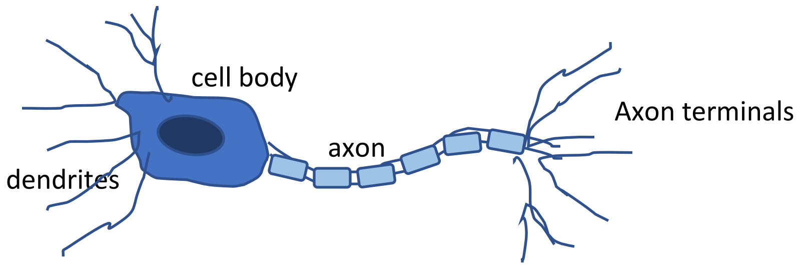
7 ARTIFICIAL NEURAL NETWORK I



Neurons are nerve cells in the brain. They use electrical impulses and chemical signals to transmit information to other nerve cells, and between the brain and the rest of the nervous system. Neurons are composed of dendrites (signal inputs), cell body (processing), and axon with terminals (outputs). Neurons can connect with other neurons to form a chain of complex input and output responses. Signals pass from one neuron to another via synapses connecting axon terminals with dendrites of the next neuron. Refer to the diagram above.

Artificial intelligence (AI) fundamentally is about harnessing computer programming to mimic human behavior such as choice and recognition using large data inputs. Machine learning (ML) may be viewed as a specific AI domain to perform specific tasks such as prediction and classification. One major area of ML in this context is neural network (NN) that builds an architecture or structure of input-processing-output nodes mimicking the structure of human brain neurons. The processing of input information and the eventual output of prediction and classification is contained in the algorithm of the ML. Deep AI in such areas refers to more complex algorithms.

In this chapter, we study a basic NN called Multi-Layer Perceptron (MLP). This is sometimes referred to as Artificial Neural Network (ANN). In the next chapter we will study some other more complex NNs.

Understanding Neural Networks mainly includes two aspects. One is the structure of neural networks, and the other is the training and learning of neural networks. It is just like how our brain structure is composed, and how we learn and recognize different things based on this composition.

A single layer perceptron may be characterized as follows in the simple architecture with inputs, the computation process node in the middle, and output(s).

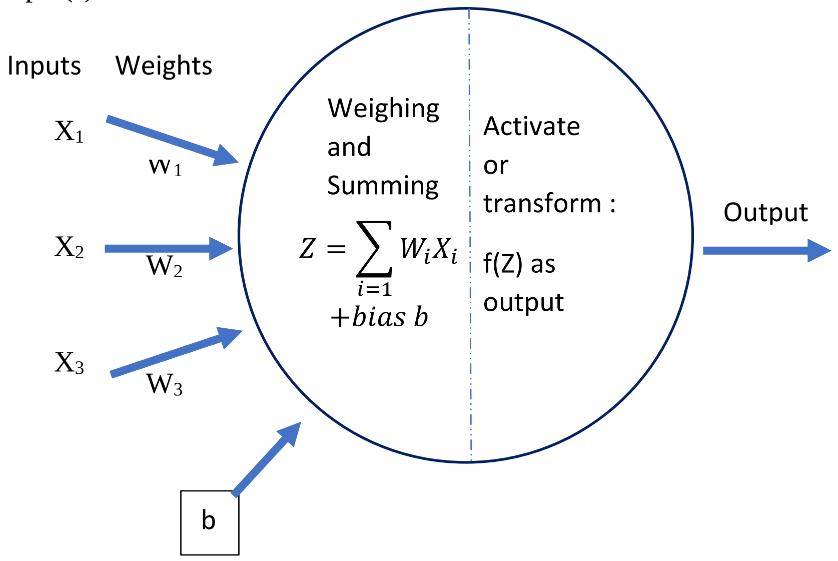


Figure 7.1

This embryonic version of today's NN essentially takes inputs or features of a case or instance, {X_1, X_2. X_3}, aggregates it with corresponding weights {W_1, W_2, W_3}, adds a bias term b, then computes the output

$$f\left(\sum_{i=1}^{3} W_i X_i + b\right)$$

where the bias (or adjustment of mean) can be treated like an input of 1 with weight b. The weights W_j's in a NN are initiated typically randomly as small positive or negative numbers, e.g., between -1 and +1. f(.) is an activation or transformation function.

An example of an activation function is $f(Z) = 1$ if $Z \geq \theta$, and 0 otherwise, as seen in Figure 7.2. This could be an effective function in classifying cases with labels or targets that can be represented as binary 1 or 0.

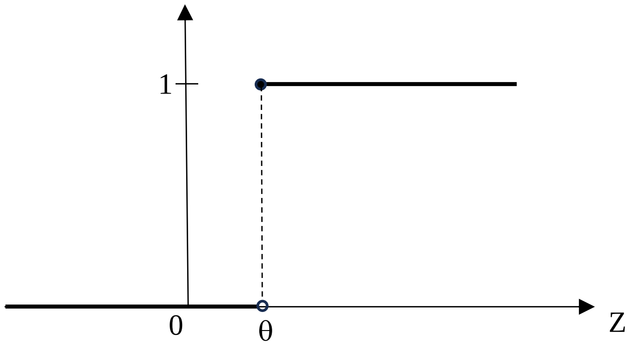


Figure 7.2

In a 2-dimensional problem with only two features X_1 and X_2, if cases with label 1 and cases with label 0 are separated by dotted line $w_1x_1+w_2x_2+b = \theta$, then activation function $f(w_1x_1+w_2x_2+b) = 1$ or 0 depending on $w_1x_1+w_2x_2+b \geq \theta$ or $< \theta$ clearly serves the purpose of predicting the labels based on inputs x_1 and x_2. See in Figure 7.3 such a separation of cases.

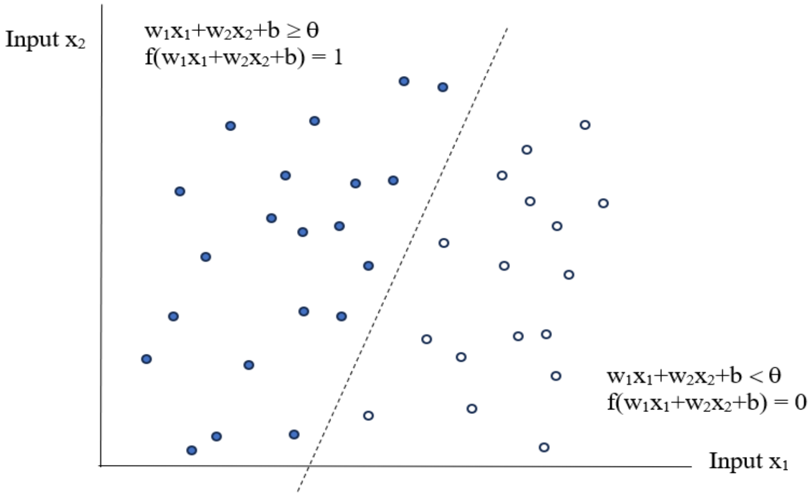


Figure 7.3

In network language, the inputs are like nodes and the weights are edges.[1] (Sometimes the latter are referred to as the synapses.) The computation node may be called a neuron.

[1] In mathematics and computer science, networks are structures that are represented by nodes, edges, and other abstract connections – they form a part of graph theory.

Suppose we use two layers of computations as follows. We call such NN multi-layer perceptron (MLP) when there is more than 1 layer. Some authors prefer to call the following a single layer perceptron (not including the output layer as the second layer).

The same inputs or features are now fed to not one, but three different nodes or neurons $\#1^{[1]}$, $\#2^{[1]}$, $\#3^{[1]}$ in the first layer. Notationally, superscript [j] denotes the j^th computational layer. The first layer neurons each has output just like in a single layer perceptron and these outputs become inputs (with no change of values) to the next (output) layer of neuron. In the latter it is neuron $\#1^{[2]}$. Neuron $\#1^{[2]}$ then produces one output.

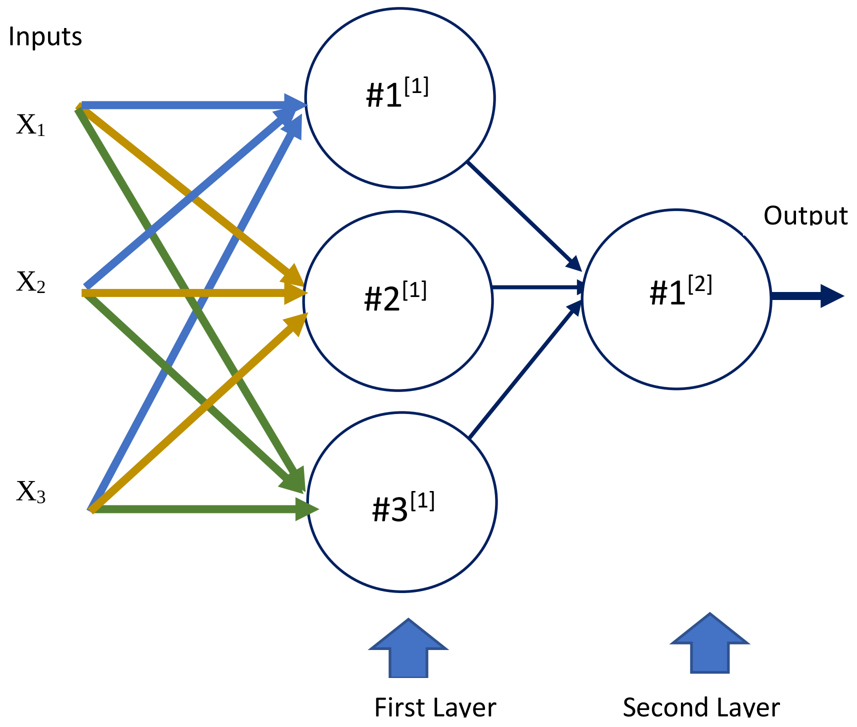


Figure 7.4

The above is a Many-to-One (many inputs toward one output) ANN. For a different problem, the structure could also be a Many-to-Many ANN (see below) where there are now two nodes or neurons in the second (output) layer and then two outputs accordingly from neurons #1[2] and #2[2].

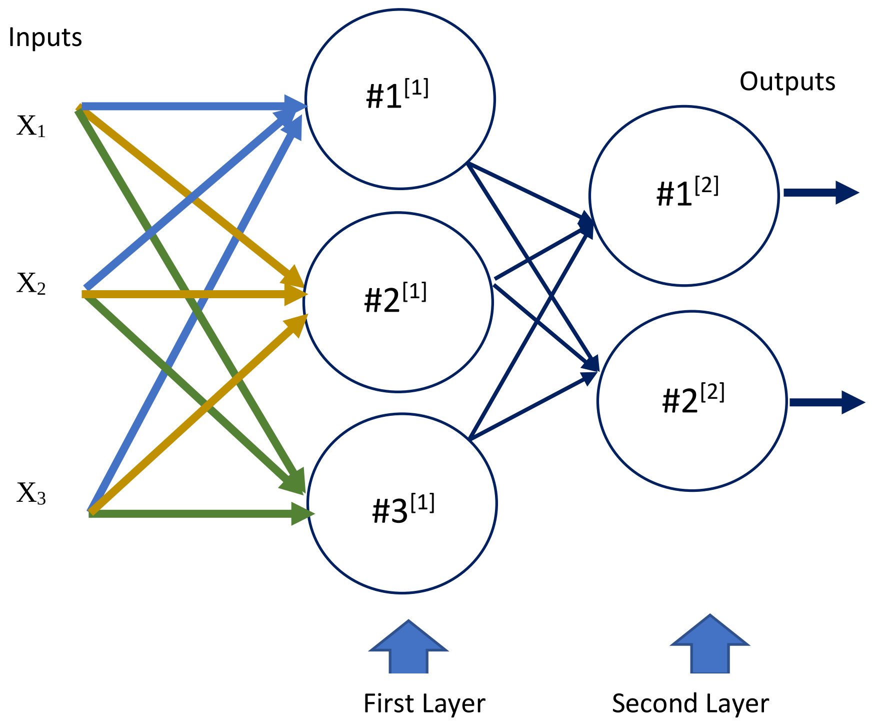
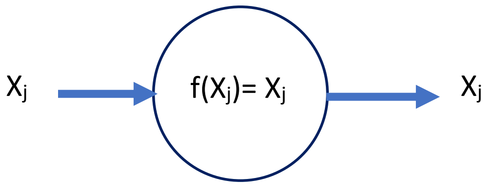


Figure 7.5

Note that we have not yet shown the weights and the bias in the diagram. In the following diagram, if we consider an input X_j producing identical output X_i (with a trivial identity activation function), then we can also treat an input as a neuron since we see that the last layer as outputs in Figure 7.5 has inputs as neurons in the first layer.



Then all the inputs can be represented as input nodes or neurons – now drawn similarly as circles. Moreover, we envelope neurons together that belong to the same forward step.

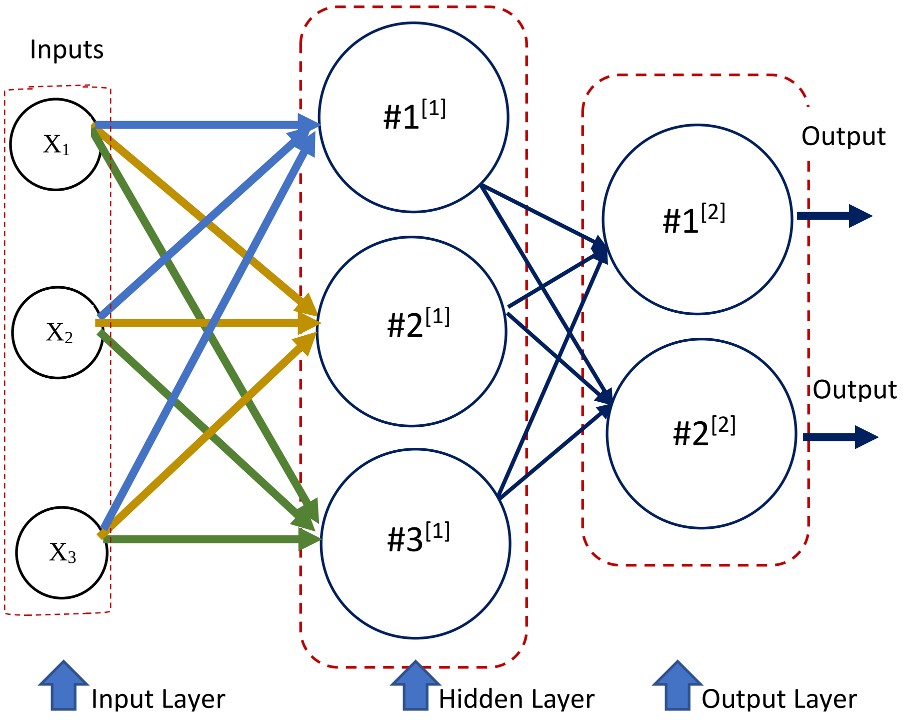


Figure 7.6

Thus, there is an input layer of input nodes or neurons, an output layer of outputs or neurons, and a middle layer called “hidden layer” as the computed outputs from the nodes in this layer are not explicitly observed, unlike those in the input and output layer nodes. For the same prediction or classification problem, an ANN can be made wider (larger number of input nodes or neurons in the same layer) and/or deeper (higher number of hidden layers). These affect the size (total number of neurons in the NN). A deeper NN can increasingly learn more nonlinear patterns. In this standard structure, each neuron will connect with all neurons in the right-side layers via the forward pass process denoted by the pointed arrows.

A rule of thumb is that the number of nodes in the hidden layer could be about 2/3 of those in the input layer and the output layer combined, and less than twice the number in the input layer. These node width and depth could be hyperparameters to be tuned.

Pruning removes nodes where input weights are close to zero – pruning can occur during training of model. Normalizing features also avoids some weights being close to 0 as out-scale feature sizes are standardized.

Suppose an ANN has two hidden layers, an input layer of similarly 3 nodes (besides bias), and one output node. It can be represented diagrammatically as follows.

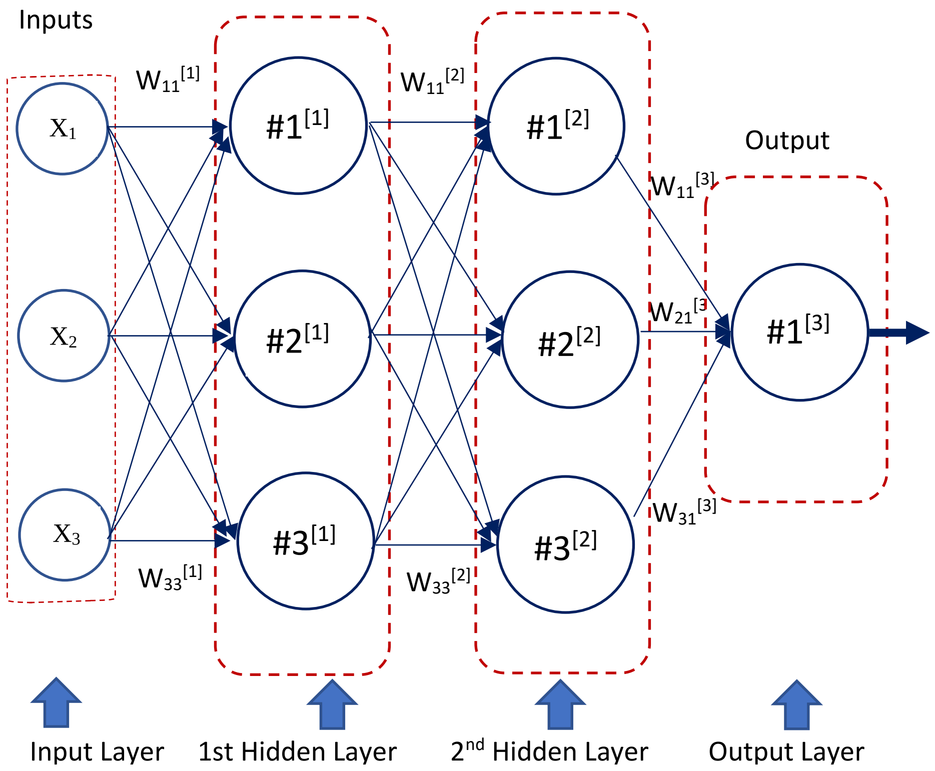


Figure 7.7

The edges or synapses connecting different neurons in different layers continue to represent weights. We depict some weights in the diagram above (not all – so as not to clutter up the diagram). The weights of input X_1 on neurons $\#1^{[1]}$, $\#2^{[1],}$ $\#3^{[1]}$ are respectively $W_{11}^{[1]}$, $W_{12}^{[1]}$, and $W_{13}^{[1]}$. The weights of input X_2 on neurons $\#1^{[1]}$, $\#2^{[1],}$ $\#3^{[1]}$ are respectively $W_{21}^{[1]}$, $W_{22}^{[1]}$, and $W_{23}^{[1]}$. The weights of input X_3 on neurons $\#1^{[1]}$, $\#2^{[1],}$ $\#3^{[1]}$ are respectively $W_{31}^{[1]}$, $W_{32}^{[1]}$, and $W_{33}^{[1]}$.

The weights of output from neuron $\#1^{[1]}$ on neurons $\#1^{[2]}$, $\#2^{[2],}$ $\#3^{[2]}$ are respectively $W_{11}^{[2]}$, $W_{12}^{[2]}$, and $W_{13}^{[2]}$. The weights of output from neuron $\#2^{[1]}$ on neurons $\#1^{[2]}$, $\#2^{[2],}$ $\#3^{[2]}$ are respectively $W_{21}^{[2]}$, $W_{22}^{[2]}$, and $W_{23}^{[2]}$. The weights of output from neuron $\#3^{[1]}$ on neurons $\#1^{[2]}$, $\#2^{[2],}$ $\#3^{[2]}$ are respectively $W_{31}^{[2]}$, $W_{32}^{[2]}$, and $W_{33}^{[2]}$.

The weights of outputs from neurons $\#1^{[2]}$, $\#2^{[2],}$ $\#3^{[2]}$ on neuron $\#1^{[3]}$ are respectively $W_{11}^{[3]}$, $W_{21}^{[3]}$, and $W_{31}^{[3]}$. The weights are connections from one neuron to another in the next forward layer. There is no connection between neurons in the same layer. In general, the weight from n^{th} hidden layer $\#j^{[n]}$ neuron to the next layer $\#k^{[n+1]}$ neuron is $w_{jk}^{[n+1]}$.

The purpose of the ANN is to perform prediction or classification on a set of data containing cases or instances with labels or targets and are each associated with a set of features or characteristics. Examples are predictions of likely default (target status) given cases of borrowers with different background features/characteristics, of treatment success (target status) given patients (subjects) with different medical background features/conditions, of correctly identifying criminals (target status) given subjects' profile and history, of click-through rate (CTR) in a digital marketing campaign with different campaign resources, of next stock price or corporate earnings given latest corporate financial and other information, of identifying an object whether it belongs to a particular classification given features/characteristics of the object, and so on.

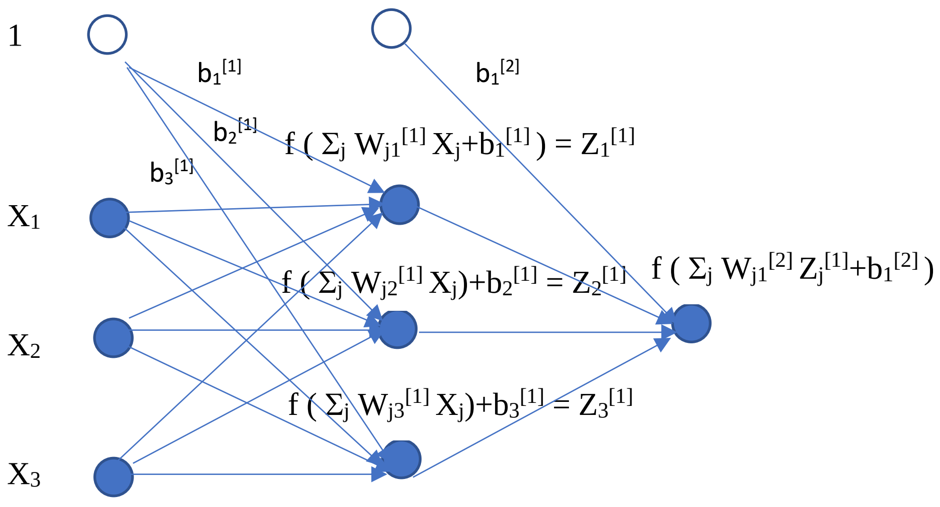
To prepare a model for prediction with new data, the model must be trained using many known cases/subjects or instances with their known target statuses and with associated features/characteristics or conditions in the training data set. Hyperparameters in the model could be fine-tuned or optimally selected using a validation data set. Once the model is calibrated with optimal hyperparameters, the weights could be optimally trained again using a re-grouped training data set (discussed in earlier chapters). The ANN can then perform the necessary computations based on these optimal weights (including optimal biases), fine-tuned hyperparameters, and the fresh inputs to find the predicted output(s) or target value(s) in the test data set.

The ANN computes the output(s) in the output layer in a forward pass process called Forward Propagation – this is discussed as follows. At the end of the pass, the errors of the fittings of all training data points are measured by a loss function to be minimized. The reduction in fitting errors requires the revision of the weights and biases – this uses a process called Backward Propagation which will also be discussed later. One forward propagation together with one backward propagation of the entire training data set is called an epoch. Sometimes the training data set (whole batch) is broken down into (mini-) batches that are smaller subsets in a random partitioning. If the size of each mini batch is fixed at n, then the number of mini batches in an epoch is total training sample size divided by the mini batch size. The forward and backward propagations are then done one mini-batch after another until the whole training data set has gone through one epoch. The number of mini batches implies the number of iterations in updating or revising the weights and biases.

Many epochs can be performed, which means a new epoch will require a new re-sampled training set (some may use the same sample). Each forward propagation on a new mini batch of same epoch or next epoch uses the updated weights and biases from the previous backward propagation. The total number of iterations is equal to the number of epochs multiplied by the number of mini batches in each epoch. Sometimes it may not be necessary to increase the number of epochs or mini batches per epoch when the fitting/prediction error in the training data set is sufficiently small.

7.1 Forward Propagation

We use a single hidden layer MLP to illustrate the forward propagation computation of a single output. The bias inputs are depicted as originating from the unfilled circles – we use a more general structure whereby the bias can be different for each of hidden and output layer neuron – shown as $b_j^{[n]}$ – corresponding to the #$j^{[n]}$ neuron. (This is equivalent to using the same bias of value 1 for all neurons in the next forward layer and then putting different weights $b_j^{[n]}$ on them.)



1
X_1
X_2
X_3
$b_1^{[1]}$
$b_2^{[1]}$
$b_3^{[1]}$
$b_1^{[2]}$
$f\ (\ \Sigma_j\ W_{j1}^{[1]}\ X_j+b_1^{[1]}\ ) = Z_1^{[1]}$
$f\ (\ \Sigma_j\ W_{j2}^{[1]}\ X_j)+b_2^{[1]} = Z_2^{[1]}$
$f\ (\ \Sigma_j\ W_{j3}^{[1]}\ X_j)+b_3^{[1]} = Z_3^{[1]}$
$f\ (\ \Sigma_j\ W_{j1}^{[2]}\ Z_j^{[1]}+b_1^{[2]}\ )$

Figure 7.8

Given the various weights, biases, and inputs, the final output is

$$f\ (\ \Sigma_j\ W_{j1}^{[2]}\ Z_j^{[1]}+b_1^{[2]}\ ) = \hat{Y}$$

for a particular case or training sample point with associated features (X_1, X_2, X_3) where superscript [1] to output Z_j denotes output is at hidden layer 1. In a standard ANN structure, the activation function f(.) remains the same for every layer.

Activation Function f (.)

The weighted inputs and biases are passed through activation function f(Z). The latter is typically a nonlinear function with an S-curve such as the Sigmoid function (logistic function) $f(Z) = 1/(1 + e^{-Z})$, the Hyperbolic Tangent function (Tanh) $f\ (Z) = (e^Z - e^{-Z})/\ (e^Z + e^{-Z})$, or the Rectifier Linear Unit function[2] (ReLU) that is $f(Z) = \max\ (Z,\ 0)$. The nonlinear functions are depicted graphically as follows. In the standard ANN, the activation function remains

[2] An alternative version, the Leaky ReLu is sometimes used. Unlike ReLu, Leaky ReLu has a small positive slope instead of zero slope for negative Z values. This slope is pre-fixed before training. Using Leaky ReLu generally appears to help in reaching convergence faster than the standard ReLu.

the same for each neuron in the same layer and remains the same also for different layers.

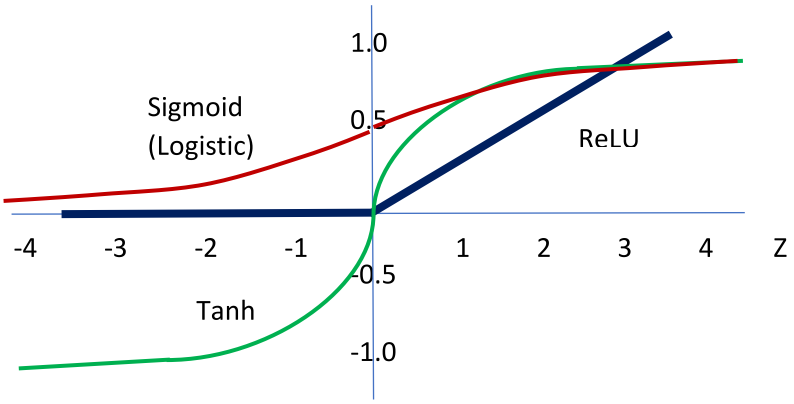


Figure 7.9

Suppose we have N number of data points (sample points or cases) in our training data set. Each case/subject/instance k has an actual output Y_k and associated features ($X_{1,k}$, $X_{2,k}$, $X_{3,k}$). The notation now includes subscript k to indicate association with one case or instance.

The forward propagation computations after an optimal number of revisions of weights and biases end up with a target prediction of

$$\hat{Y}_k = f\ (\ \Sigma_j\ W_{j1}^{[2]}\ Z_{j,k}^{[1]} + b_1^{[2]}\ )$$

where $Z_{m,k}^{[1]} = f\ (\ \Sigma_j\ W_{jm}^{[1]}\ X_{j,k} + b_m^{[1]}\ )$ denotes output node m of the hidden layer in the kth case. Note there is only one output in this NN, but we add subscripts k to denote the kth case or instance in the training set data. Also, the weights W_{jm} are constant for whichever k. As the final output or target prediction $\hat{Y}_k$ is computed in a forward pass manner, this ANN is also called a feedforward NN due to this simple architecture. We consider NNs with more complicated architecture in a later chapter – those are sometimes referred to as deep learning NNs.

Error or Loss Functions

The training fitting/prediction error is an error or loss function based on the training data set of all training sample points and measures 'deviations' of NN outputs (target predictions) of all cases with the actual outputs of the associated cases. The idea in training the model is to find optimal weights and biases so that the training set error or loss function is minimized or reduced toward zero. A commonly used error/loss function for continuous variable targets is the average least squares of deviations (or mean square error, MSE):

$$L\left(Y,\hat{Y}\right) \equiv \tfrac{1}{N}\textstyle\sum_{k=1}^{N} L\left(Y_k,\hat{Y}_k\right) = \tfrac{1}{N}\textstyle\sum_{k=1}^{N}\left(Y_k - \hat{Y}_k\right)^2. \qquad (7.1)$$

where $Y = (Y_1, Y_2, Y_3, \dots, Y_N)$ and $\hat{Y} = (\hat{Y}_1, \hat{Y}_2, \hat{Y}_3, \dots, \hat{Y}_N)$. This is also called L^2 loss function and is common for target variables that are continuous variables and not categorical or binary numbers. Sometimes its square root, the root mean square error (RMSE) is used. An alternative is the mean absolute loss or mean absolute error (MAE) or L^1 loss function

$$L\left(Y,\hat{Y}\right) = \tfrac{1}{N}\textstyle\sum_{k=1}^{N}\left|Y_k - \hat{Y}_k\right|.$$

In scenarios where there are outliers, MSE will usually be much larger than MAE because the difference between estimated target values and actual target values will be amplified after squaring. MAE is thus more robust in the presence of some outliers, i.e., producing more stable results. However, MSE has better convergence properties when it comes to finding optimal weights.

The Huber loss function $L_\delta \left(Y,\hat{Y}\right)$ is a combination of MSE and MAE, including a hyperparameter δ: $\frac{1}{N}\sum_{k=1}^{N} L_\delta\left(Y_k,\hat{Y}_k\right)$ where

$$L_\delta\left(Y_k,\hat{Y}_k\right) = \begin{cases} \dfrac{1}{2}(Y_k - \hat{Y}_k)^2, & \text{if } \left|Y_k - \hat{Y}_k\right| < \delta \\ \delta\left|Y_k - \hat{Y}_k\right| - \dfrac{1}{2}\delta^2, & \text{otherwise.} \end{cases}$$

When $\left|Y_k - \hat{Y}_k\right| < \delta$, L_δ becomes MSE. When $\left|Y_k - \hat{Y}_k\right| \geq \delta$, it functions like MAE. Huber loss function overcomes the shortcoming of MSE when the deviation becomes larger perhaps due to an outlier.

When the actual target or output is categorical, different types of error/loss functions may be more practicable. In a binary classification where the target is either 1 or 0, e.g., it rains or it does not in weather prediction, or it is a bird or it is not based on picture pixels, or the firm has earnings increase or not, then error function could be 1 minus the accuracy where accuracy is the fraction of correct prediction. However, this function may not be easily differentiable which makes it harder to use for finding optimal weights.

Typically, a binary classification prediction using features would provide a probability estimate $\hat{P}$ of the target status whether it is 1. Otherwise, 1-$\hat{P}$ would be estimate of the probability the target status is 0. A useful error/loss function for the binary classification is binary cross-entropy that employs estimate $\hat{P}_k$for each case k: $L(Y,\hat{P}) = \frac{1}{N}\sum_{k=1}^{N} L(Y_k,\hat{P}_k)$ where

$$L(Y_k,\hat{P}_k) = -[\, Y_k \, ln \, \hat{P}_k + (1-Y_k) \ln (1-\hat{P}_k) \,].$$

This is also the log loss function for binary classification. $\hat{P}_k$ is the predicted probability that Y_k = 1, and $1-\hat{P}_k$ is the predicted probability that Y_k = 0. Note that here when Y_k = 1, the loss function is reduced when $\hat{p}_k \in (0,1)$ is higher. When Y_k = 0, the loss function is reduced when $\hat{p}_k \in (0,1)$ is lower. This loss function therefore captures the idea of ensuring the estimated or predicted probability of (Y_k = 1), $\hat{P}_k$, is high when Y_k = 1, and that the predicted probability of (Y_k = 1), $\hat{P}_k$, is low when Y_k = 0. The log loss function above is positive. The smaller is the log loss function or the smaller the binary cross-entropy measure toward zero, the less is the uncertainty and hence better prediction.

Recall that we may deem $\hat{Y}_k = 1$ or 0 if $\hat{P}_k$ > 0.5 or else ≤ 0.5. Note that unlike the MSE error/loss function where the predictor $\hat{Y}_k$ enters intuitively in the loss function $L(Y_k,\hat{Y}_k)$, the loss function in the categorical output cases is $L(Y_k,\hat{P}_k)$ where the predicted classification is not exactly the predicted output $\hat{P}_k$ but a function of it, $\hat{Y}_k(\hat{P}_k)$.

When there is more than one output, e.g., Q outputs for Q number of categories, then the (categorical) cross-entropy loss function for each case k is

$$L(Y_{k,c},\hat{P}_{k,c}) = -\sum_{c=1}^{Q} Y_{k,c} \, ln \, \hat{P}_{k,c}$$

where $Y_{k,c}$ is 1 (otherwise 0) if case Y_k belongs to class c, and $\hat{P}_{k,c}$ is the predicted probability of case k belonging to class c. We may in general use the notation of error/loss function as $L(Y_k,\hat{Z}_k)$ where $\hat{Z}_k \equiv \hat{Y}_k$ in the continuous variable case and $\hat{Z}_k \equiv \hat{P}_k$ in the categorical case.

7.2 Backward Propagation

In each epoch or training of the entire training data set from inputs to final output(s) in the output layer, the predicted outputs are compared with the actual (target) outputs and the loss function $L(Y, \hat{Z})$ is computed. Thus, the epoch involves passing the entire training set of points forward, then backward with revision of weights and biases before final passing forward to obtain the output and loss function. The second epoch will involve a resampled training data set, continued updating with a second backward propagation and then forward pass to obtain the updated loss function. This may continue for many epochs where necessary to obtain satisfactory convergence in the loss function.

When the loss function or binary cross entropy measure is too high, the model needs further improvement – we need to train the NN further, by updating/revising the weights and biases and run more epochs and so on, and hopefully end with an acceptable loss function outcome.

The Backward Propagation is the mechanism of training the NN toward an acceptable loss before proceeding for application to the validation and then the test data set. While forward propagation performs computations by moving forward in the process from inputs to the final output, backward propagation involves backwards pass that performs computations by moving backward from the loss function at the final output to adjusting the weights and biases.

As in Figure 7.8, the weights and biases feeding into each layer forward are initially obtained by random draws from usually a small range (-1,+1). In the single hidden layer MLP, the weights are $\{W_{11}^{[1]}, W_{21}^{[1]}, W_{31}^{[1]}, b_1^{[1]}\}$ (including the bias) for neuron #$1^{[1]}$ with output $Z_{1,k}^{[1]}$ for case or instance k. The weights are $\{W_{12}^{[1]}, W_{22}^{[1]}, W_{32}^{[1]}, b_2^{[1]}\}$ (including the bias) for neuron #$2^{[1]}$ with output $Z_{2,k}^{[1]}$. , and $\{W_{13}^{[1]}, W_{23}^{[1]}, W_{33}^{[1]}, b_3^{[1]}\}$ (including the bias) for neuron #$3^{[1]}$ with output $Z_{3,k}^{[1]}$. In the output layer, the weights are $\{W_{11}^{[2]}, W_{21}^{[2]}, W_{31}^{[2]}, b_1^{[2]}\}$ (including the bias) for neuron #$1^{[2]}$ with output $Z_{1,k}^{[2]} \equiv f(\Sigma_j W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]}) = \hat{Y}_k$. Hence in this structure there are altogether 16 parameters of weights and biases to update/revise: $\{W_{11}^{[1]}, W_{21}^{[1]}, W_{31}^{[1]}, b_1^{[1]}, W_{12}^{[1]}, W_{22}^{[1]}, W_{32}^{[1]}, b_2^{[1]}, W_{13}^{[1]}, W_{23}^{[1]}, W_{33}^{[1]}, b_3^{[1]}, W_{11}^{[2]}, W_{21}^{[2]}, W_{31}^{[2]}, b_1^{[2]}\}$.

To update $W_{11}^{[1]}$ toward minimizing the error/loss objective function, a standard approach is to use the gradient descent method/algorithm or its variants. The gradient descent method requires the computation of the partial slope of the objective function with respect to each of the weights and biases.

The estimation uses the chain rule. In the single layer MLP, the diagram below shows how

$$\frac{\partial L(Y,\hat{Z})}{\partial W_{11}^{[1]}} = \frac{\partial \sum_{k=1}^{N} \frac{1}{N} L(Y_k,\hat{Z}_k)}{\partial W_{11}^{[1]}}$$

is computed. For each case k,

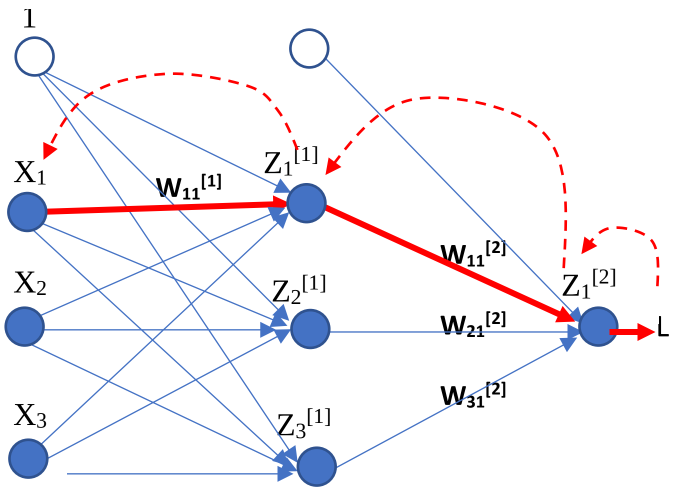


Figure 7.10

$$\frac{\partial L(Y_k,\widehat{Z}_k)}{\partial W_{m1}^{[1]}} = \left(\frac{\partial L(Y_k,\widehat{Z}_k)}{\partial Z_{1,k}^{[2]}} \times \frac{\partial Z_{1,k}^{[2]}}{\partial \Sigma \mathrm{j}\ (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})}\right)$$
$$\times \left(\frac{\partial \Sigma \mathrm{j}\ (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})}{\partial Z_{1,k}^{[1]}} \times \frac{\partial Z_{1,k}^{[1]}}{\partial \Sigma \mathrm{j}\ (W_{j1}^{[1]} X_j + b_1^{[1]})}\right)$$
$$\times \frac{\partial \Sigma \mathrm{j}\ (W_{j1}^{[1]} X_j + b_1^{[1]})}{\partial W_{m1}^{[1]}} \qquad (7.2)$$

for m=1,2,3. In the case m=1, Figure 7.10 shows the backwards pass along the path of the forward pass process starting at X_1 with edge or synapse weight $W_{11}^{[1]}$ (see bold line). The partial derivatives components follow the dotted lines. Similarly,

$$\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial W_{m2}^{[1]}}=\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial Z_{1,k}^{[2]}}\times\frac{\partial Z_{1,k}^{[2]}}{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}\times\frac{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}{\partial Z_{2,k}^{[1]}}\times\frac{\partial Z_{2,k}^{[1]}}{\partial \Sigma \text{j } (W_{j2}^{[1]}X_j+b_2^{[1]})}\times\frac{\partial \Sigma \text{j } (W_{j2}^{[1]}X_j+b_2^{[1]})}{\partial W_{m2}^{[1]}}$$

for m = 1,2,3.

$$\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial W_{m3}^{[1]}}=\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial Z_{1,k}^{[2]}}\times\frac{\partial Z_{1,k}^{[2]}}{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}\times\frac{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}{\partial Z_{3,k}^{[1]}}\times\frac{\partial Z_{3,k}^{[1]}}{\partial \Sigma \text{j } (W_{j3}^{[1]}X_j+b_3^{[1]})}\times\frac{\partial \Sigma \text{j } (W_{j3}^{[1]}X_j+b_3^{[1]})}{\partial W_{m3}^{[1]}}$$

for m = 1,2,3.

And

$$\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial b_1^{[1]}}=\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial Z_{1,k}^{[2]}}\times\frac{\partial Z_{1,k}^{[2]}}{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}\times\frac{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}{\partial Z_{1,k}^{[1]}}\times\frac{\partial Z_{1,k}^{[1]}}{\partial \Sigma \text{j } (W_{j1}^{[1]}X_j+b_1^{[1]})}\times\frac{\partial \Sigma \text{j } (W_{j1}^{[1]}X_j+b_1^{[1]})}{\partial b_1^{[1]}}$$

$$\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial b_2^{[1]}}=\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial Z_{1,k}^{[2]}}\times\frac{\partial Z_{1,k}^{[2]}}{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}\times\frac{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}{\partial Z_{2,k}^{[1]}}\times\frac{\partial Z_{2,k}^{[1]}}{\partial \Sigma \text{j } (W_{j2}^{[1]}X_j+b_2^{[1]})}\times\frac{\partial \Sigma \text{j } (W_{j2}^{[1]}X_j+b_2^{[1]})}{\partial b_2^{[1]}}$$

$$\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial b_3^{[1]}}=\frac{\partial L\left(Y_k,\widehat{Z}_k\right)}{\partial Z_{1,k}^{[2]}}\times\frac{\partial Z_{1,k}^{[2]}}{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}\times\frac{\partial \Sigma \text{j } (W_{j1}^{[2]}Z_{j,k}^{[1]}+b_1^{[2]})}{\partial Z_{3,k}^{[1]}}\times\frac{\partial Z_{3,k}^{[1]}}{\partial \Sigma \text{j } (W_{j3}^{[1]}X_j+b_3^{[1]})}\times\frac{\partial \Sigma \text{j } (W_{j3}^{[1]}X_j+b_3^{[1]})}{\partial b_3^{[1]}}$$

and so on.

For example, if $L(Y, \hat{Y}) = \frac{1}{N}\sum_{k=1}^{N}\left(Y_k - \hat{Y}_k\right)^2$, then for each k:

$$\frac{\partial L(Y_k, \hat{Y}_k)}{\partial Z_{1,k}^{[2]}} = -2\left(Y_k - \hat{Y}_k\right).$$

Note that if activation function f(X) = 1/(1 + e^{-X}), df(X)/dX = e^{-X}/(1 + e^{-X})^2 = (1-f(X)) f(X). Note that $Z_{1,k}^{[2]} = \text{f}(\ \Sigma\text{j}\ (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})$, $Z_{1,k}^{[2]} \equiv \hat{Y}_k$. Then,

$$\frac{\partial Z_{1,k}^{[2]}}{\partial\ \Sigma\text{j}\ (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})} = (1-\hat{Y}_k)\ \hat{Y}_k$$

$$\frac{\partial \Sigma\text{j}\ (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})}{\partial Z_{1,k}^{[1]}} = W_{11}^{[2]}$$

$$\frac{\partial Z_{1,k}^{[1]}}{\partial \Sigma\text{j}\ (W_{j1}^{[1]} X_j + b_1^{[1]})} = (1-Z_{1,k}^{[1]})\ Z_{1,k}^{[1]}$$

and $\frac{\partial \Sigma\text{j}\ (W_{j1}^{[1]} X_j + b_1^{[1]})}{\partial W_{11}^{[1]}} = X_1$.

Hence,

$$\frac{\partial L(Y_k, \hat{Y}_k)}{\partial W_{11}^{[1]}} = -2\left(Y_k - \hat{Y}_k\right) \times (1 - \hat{Y}_k)\ \hat{Y}_k \times W_{11}^{[2]} \times (1 - Z_{1,k}^{[1]})\ Z_{1,k}^{[1]} \times\ X_1$$

and

$$\frac{\partial L(Y, \hat{Y})}{\partial W_{11}^{[1]}} = -\frac{1}{N}\sum_{k=1}^{N} 2\left(Y_k - \hat{Y}_k\right) \times (1 - \hat{Y}_k)\ \hat{Y}_k \times W_{11}^{[2]} \times (1 - Z_{1,k}^{[1]})\ Z_{1,k}^{[1]} \times\ X_1\ .$$

For a shorter forward pass from $Z_1^{[1]}$ to L, the partial derivative of

$$\frac{\partial L\left(Y_k,\widehat{Y}_k\right)}{\partial W_{11}^{[2]}} = \left(\frac{\partial L\left(Y_k,\widehat{Y}_k\right)}{\partial Z_{1,k}^{[2]}} \times \frac{\partial Z_{1,k}^{[2]}}{\partial \Sigma \mathrm{j}\ (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})}\right) \times \frac{\partial \Sigma \mathrm{j}\ (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})}{\partial W_{11}^{[2]}} \qquad (7.3)$$

which is just part of (7.2). For longer forward pass involving a deeper NN such as the following with 3 hidden layers and two final outputs, the partial derivative of error/loss function (possibly weighted linear combination of the loss functions L_1 and L_2 of the two final outputs) with respect to $W_{11}^{[1]}$ (the first edge or weight from input X_1) is the sum of each of $3^2 \times 2$ backward derivative chain computations – one such backward pass is shown in bold below.

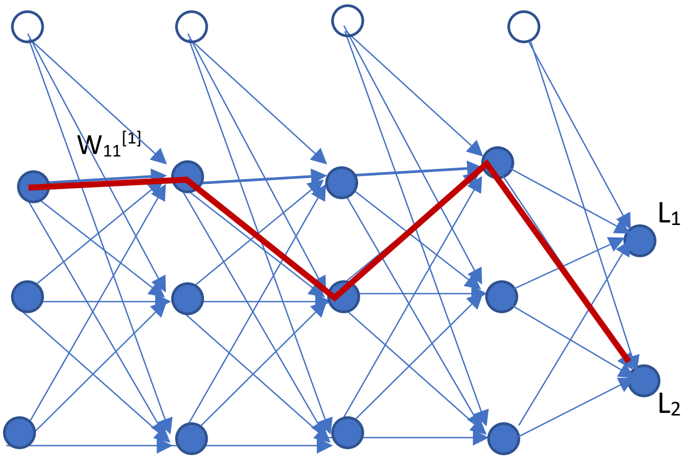


Figure 7.11

The purpose of finding all derivatives of loss function with respect to all the weights and biases in the ANN is as follows. Suppose for a particular weight (or bias) W, ∂ L/∂ W > (<) 0, then we could try to attain a lower loss by decreasing (increasing) the weight (or bias). To do so, we could find a revised weight (or bias) by adjusting it lower (higher) from the previous one. Hence the adjustment for each iteration (over mini batches and over epochs) or updating of all weights at the same time is as follows.

$$W_{ij}^{*[r]} = W_{ij}^{[r]} - \alpha \frac{\partial L}{\partial W_{ij}^{[r]}}$$

where updated weight is $W^{*}{}_{ij}^{[r]}$, and $\alpha > 0$ is a learning rate that determines how fast is the descent toward the minimum. α is a hyperparameter in this algorithm. These adjustments toward a smaller loss function form the gradient descent approach in the loss function minimization.

In Eqs. (7.1) and (7.2), we see that one backward propagation computation (and the initial forward pass) in one epoch (across all data points N in the training set) involves at least N times the number of computations of the partial derivatives for one case/instance k such as in (7.2). The computing memory resource may be overstretched with the large number of sample point derivatives. With E epochs, there would only be E number of iterations or updates on the parameters and this may not be adequate if the NN is highly nonlinear.

A variant of the (batch) gradient descent, is to perform the backpropagation using not the entire training sample of size N in one epoch, but smaller (mini) batches of size $n < N$. This is called the 'mini-batch' gradient descent approach.

In this way, the computational memory burden for each iteration involving only n sample points and correspondingly fewer partial derivatives is lesser. The number of mini batches in an epoch is then N/n. There are then N/n number of iterations in each epoch, i.e., all the data need to pass through before one epoch is counted. If E number of epochs is run, then in total there will be $E \times N/n > E$ number of iterations in updating the parameters in the training. This will provide more accuracy if the NN is highly nonlinear.

Another popular variant is the stochastic gradient descent (SGD) that randomly (by default random generator in the algorithm) sequentially computes the gradients and update for each sample point within the batch – it is like the mini-batch with a batch size of 1. The updating of the stochastic gradient descent partial derivatives for each sample point can cause the convergence to the minimum error/loss function to be more volatile than when using the (full) batch or mini-batch training data set. However, there are far more iterations. Different platforms such as TensorFlow may, however, have related SGD algorithms that are built a bit differently with momentum factor to adjust the learning rate.

Besides batch size strategy in achieving computationally efficient convergence, another related algorithm is in adjusting the learning rate with each parameter/weights update/iteration instead of keeping it constant throughout the training. One popular algorithm is Adam (adaptive moment estimation). The algorithm computes the exponentially decaying (moving) averages of past gradients and past squared gradients, viz.

$$M_t = \beta_1 M_{t-1} + (1-\beta_1)G_t$$

and

$$V_t = \beta_2 V_{t-1} + (1-\beta_2)G_t^2$$

where t is each iteration step forward, G_t and G_t^2 are the computed gradient and squared gradient at t for a particular weight updating, i.e., $G_t = \partial L/\partial W_{ij,t}^{[r]}$. M_0 and V_0 are typically initialized at 0. M_t and V_t are moving averages of G_t and G_t^2 respectively.

Now, $M_1 = (1-\beta_1)G_1$, $M_2 = \beta_1(1-\beta_1)G_1 + (1-\beta_1)G_2$, $M_3 = \beta_1^2(1-\beta_1)G_1 + \beta_1(1-\beta_1)G_2 + (1-\beta_1)G_3$.

In general, $M_t = (1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\, G_i$. Taking unconditional expectation,

$E(M_t) = (1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\, E(G_i)$.

Assuming $E(G_i)$ for every i is approximately constant, then

$E(M_t) \approx E(G_t)\,(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i} = E(G_t)\,(1-\beta_1^t)$.

The approximate mean of G_t, $E(G_t) \approx E(M_t)\,/\,(1-\beta_1^t)$. It is said that the M_t process supposedly converging over long-term to some exponentially (decreasing) weighted averages of past and current G_t is biased away from $E(G_t)$ and M_t needs to be adjusted to $\widehat{M}_t = M_t/(1-\beta_1^t)$ so that $E(\widehat{M}_t) = E(M_t)/(1-\beta_1^t) \approx E(G_t)$. Now $\widehat{M}_t$ is (approximately) unbiased so that its long-term weighted averaging of past and current G_t should lead to $E(G_t)$.

In a similar analysis,
$V_1 = (1-\beta_2)G_1^2$, $V_2 = \beta_2(1-\beta_2)G_1^2 + (1-\beta_2)G_2^2$, $V_3 = \beta_2^2(1-\beta_2)G_1^2 + \beta_2(1-\beta_2)G_2^2 + (1-\beta_2)G_3^2$,…, $V_t = (1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\, G_i^2$. Taking unconditional expectation,

$E(V_t) = (1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\; E(G_i^2)$.

Assuming $E(G_i^2)$ for every i is approximately constant, then

$E(V_t) \approx E(G_i^2)\,(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i} = E(G_i^2)\,(1-\beta_2^t)$.

The approximate "uncentered variance" (second moment) of G_t, $E(G_i^2) \approx E(V_t)\,/\,(1-\beta_2^t)$. It is said that the V_t process supposedly converging over long-term to some exponentially (decreasing) weighted averages of past and current G_t^2 is biased away from $E(G_t^2)$ and V_t needs to be adjusted to $\hat{V}_t = V_t/(1-\beta_2^t)$ so that $E(\hat{V}_t) = E(V_t)/(1-\beta_2^t) \approx E(G_t^2)$. Now $\hat{V}_t$ is (approximately) unbiased so that its long-term weighted averaging of past and current G_t^2 should lead to $E(G_t^2)$.

The Adam update rule for gradient descent in the weight adjustment during backward propagation is:

$$W_{ij,t}^{[r]} = W_{ij,t-1}^{[r]} - \theta \frac{\widehat{M}_t}{\sqrt{\widehat{V}_t} + \varepsilon}$$

where given a step size θ (hyperparameter to be tuned – it is typically a small positive number), adjustment or descent is faster with higher $\widehat{M}_t$ and lower $\widehat{V}_t$ (more persistent gradient in one direction with less variability). In most implementations, default values for β_1 and β_2 are 0.9 and 0.999. ε is included to prevent G_t (hence also G_t^2) disappearing to 0 when division otherwise becomes not possible in the denominator. Typical value for ε is 10^{-8}. A comparison with the gradient descent shows the learning rate in Adam may be construed as changing according to $\alpha_t = \theta\, \widehat{M}_t / \left[G_t\left(\sqrt{\widehat{V}_t} + \epsilon\right)\right]$. Hence, using the Adam "optimizer" or the updating or learning rule involves more computations of these hyperparameter-related parameters in the NN.

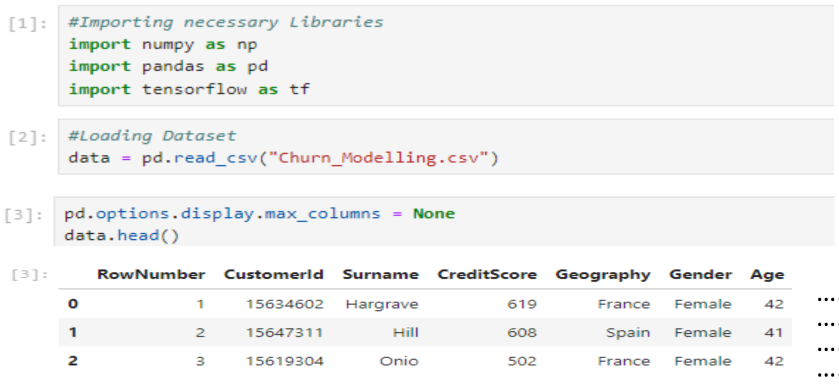
Any algorithm can also set a limit time to the iterations or stop the iterations when the loss function is close to minimum and not reducing anymore with iterations.

7.3 Worked Example – Data

The data set -- "Churn_Modelling.csv" – is found in public resource: https://www.kaggle.com/datasets/. The data set contains a bank's 10,000 customers' details, viz., nationality, gender, credit score (higher score means more credit-worthy), age, tenure (years with the bank), USD bank balance in the period of the data, the number of the bank products used by the customer, whether the customer has the bank credit card, is active customer, and estimated USD salary of the customer. These serve as features of each customer that could explain the target variable – which is whether the customer exited or left the bank. In the latter the customer had closed his/her bank account during the data period and the classification of exit is "1". Otherwise, the customer continues with the bank and the classification of exit is "0". For ease of interpretation, assume the exit prediction is for a year forward.

The ability to predict if an existing customer is likely to exit the bank (or analogously to exit as customer of a store or as employee of a company) is important as customer exit or attrition or loss of company clients could be highly damaging to the bottom line or even reputation of a company. If a customer is predicted to be likely to exit, or those customers likely to exit possess certain types or characteristics, then the bank or company can step up certain strategies on those characteristics to try to retain such customers and promote customer loyalty.

The following code lines [1] to [3] in demonstration file Chapter7-1.ipynb show the data set. Note that TensorFlow (incorporating Keras) is the key platform and API/algorithms used for the building the MLP neural networks here.



```
[1]: #Importing necessary Libraries
     import numpy as np
     import pandas as pd
     import tensorflow as tf
```

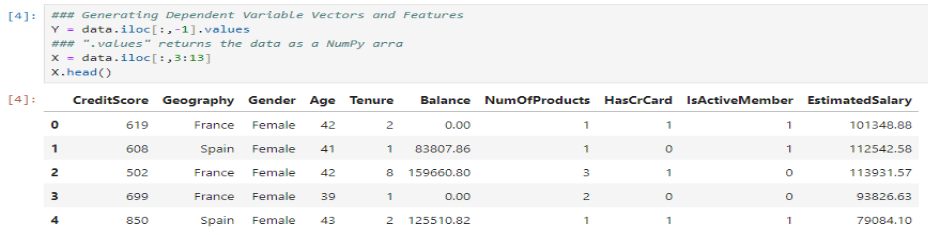
```
[2]: #Loading Dataset
     data = pd.read_csv("Churn_Modelling.csv")
```

```
[3]: pd.options.display.max_columns = None
     data.head()
```

[3]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	
0	1	15634602	Hargrave	619	France	Female	42	....
1	2	15647311	Hill	608	Spain	Female	41	....
2	3	15619304	Onio	502	France	Female	42	....
								....

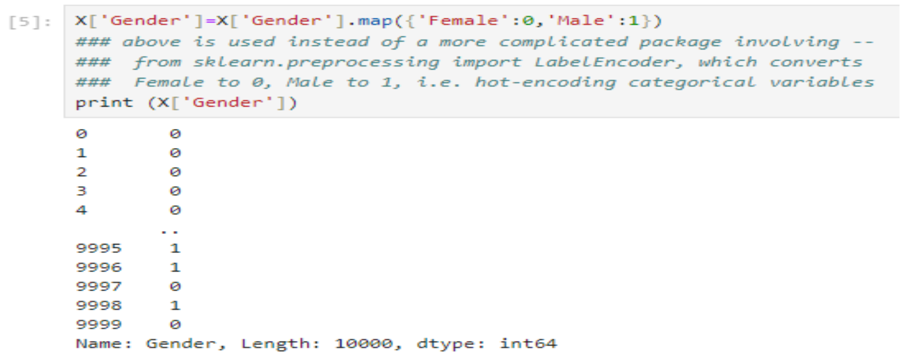
In code lines [4] to [6], the Gender variable and the Geography variable are transformed to dummy or 1, 0 variables.



```
[4]: ### Generating Dependent Variable Vectors and Features
     Y = data.iloc[:,-1].values
     ### ".values" returns the data as a NumPy arra
     X = data.iloc[:,3:13]
     X.head()
```

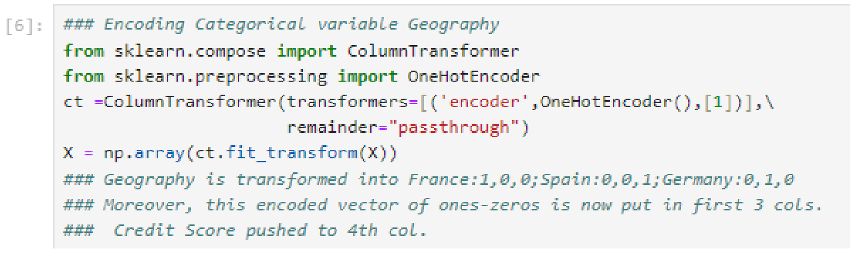
[4]:

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
0	619	France	Female	42	2	0.00	1	1	1	101348.88
1	608	Spain	Female	41	1	83807.86	1	0	1	112542.58
2	502	France	Female	42	8	159660.80	3	1	0	113931.57
3	699	France	Female	39	1	0.00	2	0	0	93826.63
4	850	Spain	Female	43	2	125510.82	1	1	1	79084.10



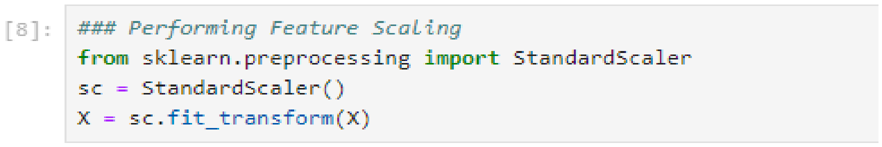
```
[5]: X['Gender']=X['Gender'].map({'Female':0,'Male':1})
     ### above is used instead of a more complicated package involving --
     ###  from sklearn.preprocessing import LabelEncoder, which converts
     ###  Female to 0, Male to 1, i.e. hot-encoding categorical variables
     print (X['Gender'])

     0       0
     1       0
     2       0
     3       0
     4       0
            ..
     9995    1
     9996    1
     9997    0
     9998    1
     9999    0
     Name: Gender, Length: 10000, dtype: int64
```



```
[6]: ### Encoding Categorical variable Geography
     from sklearn.compose import ColumnTransformer
     from sklearn.preprocessing import OneHotEncoder
     ct =ColumnTransformer(transformers=[('encoder',OneHotEncoder(),[1])],\
                           remainder="passthrough")
     X = np.array(ct.fit_transform(X))
     ### Geography is transformed into France:1,0,0;Spain:0,0,1;Germany:0,1,0
     ### Moreover, this encoded vector of ones-zeros is now put in first 3 cols.
     ###  Credit Score pushed to 4th col.
```

Code line [8] shows the scaling of the feature data in the whole sample (both training and test set combined) which is useful if the features are data with different orders of magnitudes.

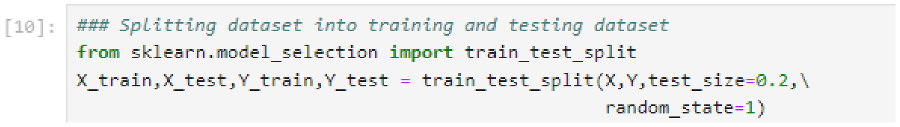


```
[8]: ### Performing Feature Scaling
     from sklearn.preprocessing import StandardScaler
     sc = StandardScaler()
     X = sc.fit_transform(X)
```

In cross-sectional feature data, we can do same mean and standard deviation transforms of each feature before shuffling into train and test set, since each random shuffling should be invariant to the scaling.

Each feature in the training set is scaled to mean 0 and variance 1. In sklearn.preprocessing.StandardScaler(), centering and scaling operations happen independently on each feature. The fit method is calculating the mean and variance of each of the features present in the data. The transform method is transforming all the features (and target variable) using the respective features' (target's) means and variances. Note that normalizing the features may avoid some weights being close to 0 as out-scale feature sizes are standardized.

Code line [10] shows split of the dataset into training (80%) and test (20%) data sets.

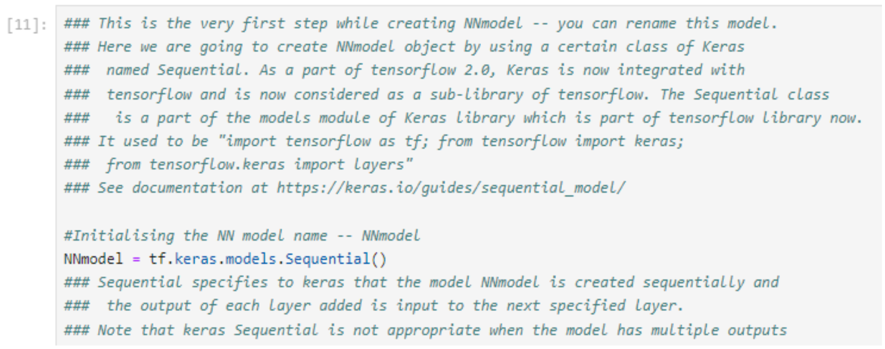


```
[10]: ### Splitting dataset into training and testing dataset
      from sklearn.model_selection import train_test_split
      X_train,X_test,Y_train,Y_test = train_test_split(X,Y,test_size=0.2,\
                                                        random_state=1)
```

Note that it is assumed a validation set would have been already used to find appropriate hyperparameters, including the type of hyperparameter computation such as "Adam". Then the training and validation data sets are re-grouped into this current training data set for training and then testing.

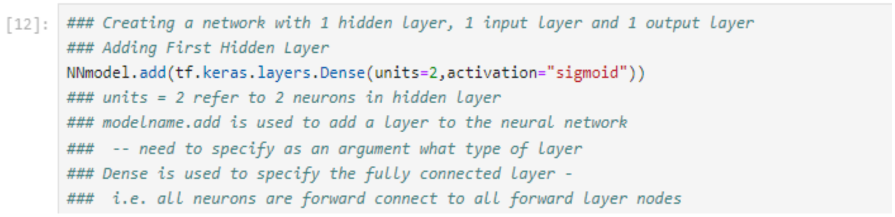
In this Chapter7-1.ipynb example, the depth and width of the NN are kept small to show comparative improvements in later examples in this chapter. Besides the learning rate, the other hyperparameters are the number of hidden layers (depth) and the number of neurons in each layer (width).

Code lines [11], [12] use the tf.keras.models.Sequential() API to build the MLP neural network model.



```
[11]: ### This is the very first step while creating NNmodel -- you can rename this model.
      ### Here we are going to create NNmodel object by using a certain class of Keras
      ###  named Sequential. As a part of tensorflow 2.0, Keras is now integrated with
      ###  tensorflow and is now considered as a sub-library of tensorflow. The Sequential class
      ###   is a part of the models module of Keras library which is part of tensorflow library now.
      ### It used to be "import tensorflow as tf; from tensorflow import keras;
      ###  from tensorflow.keras import layers"
      ### See documentation at https://keras.io/guides/sequential_model/

      #Initialising the NN model name -- NNmodel
      NNmodel = tf.keras.models.Sequential()
      ### Sequential specifies to keras that the model NNmodel is created sequentially and
      ###  the output of each layer added is input to the next specified layer.
      ### Note that keras Sequential is not appropriate when the model has multiple outputs
```



```
[12]: ### Creating a network with 1 hidden layer, 1 input layer and 1 output layer
      ### Adding First Hidden Layer
      NNmodel.add(tf.keras.layers.Dense(units=2,activation="sigmoid"))
      ### units = 2 refer to 2 neurons in hidden layer
      ### modelname.add is used to add a layer to the neural network
      ###  -- need to specify as an argument what type of layer
      ### Dense is used to specify the fully connected layer -
      ###  i.e. all neurons are forward connect to all forward layer nodes
```

In the above, first hidden layer is created using the Dense class which is part of the 'layers' module. A Dense class denotes a dense layer in the neural network. A dense layer is also called a 'fully connected' layer. This means that every neuron in one layer is connected to every neuron in the next layer. This class accepts 2 inputs -- (1) units: number of neurons that will be present in the respective layer (2) activation: specifying which activation function to be used. This example uses first input as 2. We start with a small width and will increase this to show how the NN will become more effective. In general, there is no correct answer which is the exactly right number of neurons in the layer – it will be by trial and error or some mechanical iterations till optimal accuracy or predictive effectiveness is attained. It would not be too small to be ineffective and also not be too large to be computationally too expensive. For the second input, we try the sigmoid or logistic function as an activation function for hidden layers. We can also try "relu" [rectified linear unit].

Then we create the output layer as follows.

```
[14]:  ### now we create the output layer
       #Adding Output Layer
       NNmodel.add(tf.keras.layers.Dense(units=1,activation="sigmoid"))
       ### Only 1 output neuron
```

For a binary classification problem as above, actual case output is 1 or 0. Hence we require only one neuron to output layer - output could be estimated probability of case's actual output = 1. For multiclass classification problem, if the output contains m categories, then we need to create m – 1 different neurons for the different categories. In the binary output case, the suitable activation function is the sigmoid function. For multiclass classification problem, the activation function is typically 'softmax'. The 'softmax' function predicts a multinomial probability distribution. Next we need to compile the NN model using Keras.

```
[16]:  ### After creating the layers - require compiling the NNmodel. Compiling allows the
       ###  computer to run and understand the program. It adds other elements or link other
       ###  libraries, and does optimization, such that after compiling the results are readily
       ###  computed e.g. in a binary executable program as an output.
       ### Compiling NNmodel
       NNmodel.compile(optimizer="adam",loss="binary_crossentropy",metrics=['accuracy'])
       ### Note optimizer here is a more sophisticated version of the Mean Square loss
```

The Compile method above accepts as inputs – (1) optimizer: specifies which optimizer to be used in order to perform gradient descent, (2) error/loss function, e.g., 'binary_cross-entropy' here. (For multiclass classification, it should be categorical_crossentropy), and (3) metrics: the performance metrics to use in order to compute performance. 'accuracy' is one such performance metric.

Next, we check the dimension of X_train.

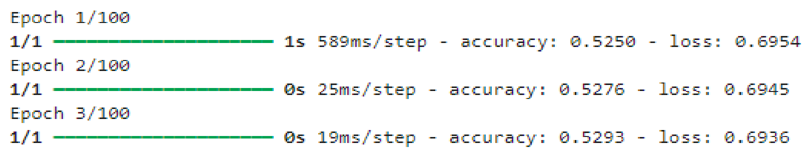
```
[18]:  X_train.shape

[18]:  (8000, 12)
```

In code line [19], we run the training of the NN on the training data set. This uses the '.fit' in TensorFlow-Keras.

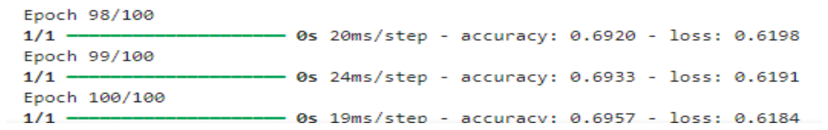
```
[19]:  ### Last step in creation of NNmodel NNmodel is trained on the training set here
       ###  with Tensor-Keras .fit based on Compiled NNmodel
       history=NNmodel.fit(X_train,Y_train,batch_size=8000,epochs = 100)
       ### Note that tf.keras.models.Sequential() by default uses glorot initializer -
       ### - drawing intial weights from a uniform distribution -- see other possibilities
       ### in https://keras.io/api/layers/initializers/
       ### Or you could try own customized wts inputs using for layer in model.layers:
       ###  init_layer_weight = []
       ###  layer.set_weights(init_layer_weight)
```

We show the output as follows. Since we specify in [19] batch size of 8000 on a 8000 sample point training set, the entire training set is run in one iteration to update the parameters. This constitutes one epoch. Then the next epoch is run. The outputs show the updated accuracy and loss each epoch 1/100, 2/100,…, till the 100/100 epoch. Within each epoch, there is only one batch, so it completes 1/1 in each epoch. (If there were two mini-batches, then it would show completion of 2/2 before moving to the next epoch.)



```
Epoch 1/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 589ms/step - accuracy: 0.5250 - loss: 0.6954
Epoch 2/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.5276 - loss: 0.6945
Epoch 3/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.5293 - loss: 0.6936
```

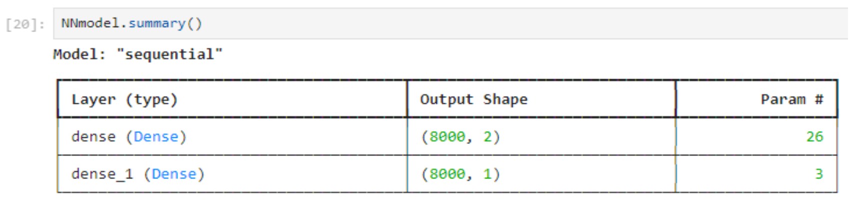
…………………………………………………………………………………………



```
Epoch 98/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.6920 - loss: 0.6198
Epoch 99/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.6933 - loss: 0.6191
Epoch 100/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6957 - loss: 0.6184
```

It is noted that after 100 epochs or iterations, the loss is 0.6184 and accuracy (% of correct predictions of exit or retention) is 0.6957 (69.57%). There is one more computation after this when the updated weights after the 100th epoch is used in the .evaluate computation of the prediction – see below.

Code line [20] provides a summary of the architecture/structure of this NN.



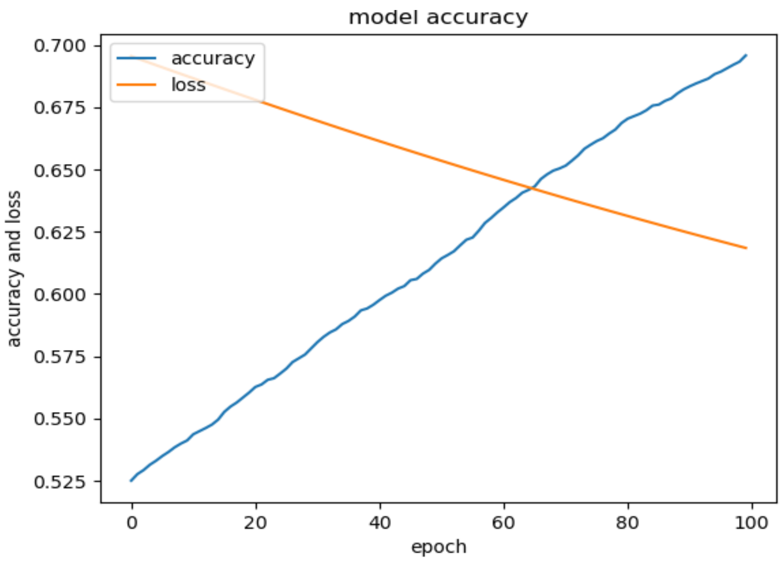
```
[20]: NNmodel.summary()
```

```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━┓
┃ Layer (type)                   ┃ Output Shape          ┃     Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━┩
│ dense (Dense)                  │ (8000, 2)             │          26 │
├────────────────────────────────┼───────────────────────┼─────────────┤
│ dense_1 (Dense)                │ (8000, 1)             │           3 │
└────────────────────────────────┴───────────────────────┴─────────────┘
```

As there are 12 features as inputs, and one bias as input to each of the two neurons in the first hidden layer, there are (12+1) × 2 = 26 weight parameters in the first dense (fully connected) hidden layer. In the 2 outputs of the first hidden layer, adding the next bias, there are 2+1 = 3 weight parameters in the next dense layer which is also the output layer.

The outputs of losses and accuracies in code line [19] can be captured in the 'history' output of the NNmodel. They are plotted as shown in [21].

```
[21]: import matplotlib.pyplot as plt
      print(history.history['accuracy'])
      print(history.history['loss'])
      plt.plot(history.history['accuracy'])
      plt.plot(history.history['loss'])
      plt.title('model accuracy')
      plt.ylabel('accuracy and loss')
      plt.xlabel('epoch')
      plt.legend(['accuracy', 'loss'], loc='upper left')
      plt.show()
```



The plot shows that as the number of training iterations (epochs) increases, the loss decreases, and the accuracy increases to about 70%.

We can obtain the final iteration weights and biases of the trained NN and use these to compute any predicted output (which is a probability of exit) given the features' values. These are shown in code lines [22] to [26].

```
[22]: ### Below are the weights in the final iteration
      first_layer_weights = NNmodel.layers[0].get_weights()[0]
      first_layer_biases  = NNmodel.layers[0].get_weights()[1]
      second_layer_weights = NNmodel.layers[1].get_weights()[0]
      second_layer_biases  = NNmodel.layers[1].get_weights()[1]
```

```
[23]: print(first_layer_weights)
      first_layer_weights.shape
      [[ 0.38304186 -0.4201266 ]
       [ 0.6633316  -0.09484755]
       [ 0.536904    0.48574576]
       [-0.19971466 -0.09429155]
       [ 0.22192973  0.60840285]
       [-0.22986005 -0.02993411]
       [ 0.37355074 -0.10736771]
       [ 0.39336446 -0.6348443 ]
       [ 0.19186138 -0.45866606]
       [ 0.5351657   0.3326683 ]
       [-0.25606754  0.5614511 ]
       [-0.36787778  0.01808825]]
[23]: (12, 2)
```

Code line [23] shows the trained weights for the (first) hidden layer neurons. There are 2 units (width of 2) or 2 neurons in the hidden as indicated in code line [12]. The first column shows weight on each input feature to the first or top neuron. The second column shows weight on each input feature to the second or bottom neuron. Code line [24] shows the two biases' weights for the two neurons in the (first) hidden layer.

```
[24]: print(first_layer_biases)
      first_layer_biases.shape
      ### (2,) here basically means 2 elements in a 1-dim array.
      ### .T has no effect on 1d array

      [-0.09503298  0.09826887]
[24]: (2,)
```

Code lines [25] and [26] show the weights in the (second) or output layer.

```
[25]: print(second_layer_weights)
      second_layer_weights.shape

      [[ 0.83238465]
       [-1.1146983 ]]
[25]: (2, 1)
```

```
[26]: print(second_layer_biases)
      second_layer_biases.shape

      [-0.09636633]
[26]: (1,)
```

Now we can use the trained weights and biases to predict on a new case with features (1, 0, 0, 500, 1, 40, 3, 60000, 2, 1, 1, 100000). These are standardized as in [8]. See code line [27]. (Note sometimes we may not need to standardize dummy variables.)

```
[27]: ### Now use trained weights and biases to predict based on a new case
      tr=sc.transform([[1, 0, 0, 500, 1, 40, 3, 60000, 2, 1, 1, 100000]])
      print(tr)  ### tr.shape is (1,12)

      [[ 0.99720391 -0.57873591 -0.57380915 -1.55748773  0.91241915  0.10281024
        -0.69598177 -0.26422114  0.80773656  0.64609167  0.97024255 -0.00156918]]
```

Based on the NN architecture, the output is then computed in code line [28] using '.predict' as 0.3890. This is verified using code lines [29] to [33] showing the forward pass computations with the activation function as the sigmoid or logistic function.

```
[28]: ### Example
      ### Predicting result for Single Observation
      print(NNmodel.predict(tr))
      ### note in each recompute -- this no. will change slightly because of
      ###  the random initiation of the weights

      1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
      [[0.38903028]]

[29]: ### now we compute the predicted prob of 1, manually
      tr.dot(first_layer_weights)  ### gives a 1 x 2 matrix

[29]: array([[0.06886018, 0.68774673]])

[30]: Flayerneurons_sum=tr.dot(first_layer_weights) + first_layer_biases
      print(Flayerneurons_sum)  ### 1 x 2 matrix

      [[-0.0261728  0.7860156]]

[31]: Flayerneurons_act=1/(1+np.exp(-Flayerneurons_sum))
      print(Flayerneurons_act)  ### 1 x 2 matrix: output of neurons in hidden layer

      [[0.49345717 0.68697516]]

[32]: Slayerneurons_sum=Flayerneurons_act.dot (second_layer_weights)+second_layer_biases
      print(Slayerneurons_sum)

      [[-0.4513902]]

[33]: predprob=1/(1+np.exp(-Slayerneurons_sum))
      print(predprob)
      ### Note this is the same output as NNmodel.predict(tr). This manual computation
      ###  of the forward pass should have output same as in NNmodel.predict(tr)

      [[0.38903029]]
```

In code line [34], we use '.evaluate' (based on the trained NN in [19]) to predict outputs for X_train and then compare with the actual outputs in Y_train. This returns a loss of 0.6178 and accuracy of 0.6974. Thus '.evaluate' produces loss and accuracy while '.predict' in [28] produces the probability of case output being "1".

```
[34]: ### Now we use the trained NNmodel to predict output in X_train sample
      NNmodel.evaluate(X_train,Y_train)
      ### evaluates the loss and accuracy as specified in the Compiler
```

```
[34]: [0.6178003549575806, 0.6973749995231628]
```

The above is based on the final weights after the 100 epochs and should be consistent with '.predict'. Code lines [35] to [44] check the predicted outputs and compare with the actual outputs and returns an accuracy of 0.697375 in [44] which is the same as that in [34].

```
[35]: ### Now use the trained NNmodel to predict output in X_train sample
      ###   - computing manually via .predict
      TE=NNmodel.predict(X_train)  ### note X_train has 8000 data points
```

```
[36]: TE.shape
```

```
[36]: (8000, 1)
```

TE is a 8000 × 1 vector.

```
[37]: h=(TE > 0.5).astype(int) ### Convert TE>0.5 == true ==> 1, False to 0
```

```
[38]: print(h)
      h.shape
```

```
      [[0]
       [1]
       [0]
       ...
       [0]
       [0]
       [0]]
[38]: (8000, 1)
```

```
[39]: ### replace all predicted elements in numpy array of value 0 with value -1
      h[h==0]=-1
```

```
[41]: ### replace all actual elements in numpy array of value 0 with value -1
      Y_train1=Y_train
      Y_train1[Y_train1==0]=-1
```

```
[44]: J=np.multiply(Y_train1.T,h.T)  ### element by element multiplication
      c=np.count_nonzero(J > 0)
      print(c,c/8000)

      5579 0.697375
```

In code line [45], we finally use the trained NN to perform prediction of exits in the test sample (of 2000 points).

```
[45]: ### Now use the trained NNmodel to predict output in X_test sample
      NNmodel.evaluate(X_test,Y_test)
      ### evaluates the loss and accuracy as specified in the Compiler

[45]: [0.6181555986404419, 0.6990000009536743]
```

The accuracy is verified in code lines [46], [47] as 69.90%.

```
[46]: ### Now use the trained NNmodel to predict output in X_test sample -
      ###  - computing manually via .predict
      TE1=NNmodel.predict(X_test)  ### note X_test has 2000 data points

      63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 511us/step

[47]: h1=(TE1 > 0.5).astype(int) ### Convert TE1>0.5 == true ==> 1, False to 0
      h1[h1==0]=-1
      Y_test1=Y_test
      Y_test1[Y_test1==0]=-1
      J1=np.multiply(Y_test1.T,h1.T)  ### element by element multiplication
      c1=np.count_nonzero(J1 > 0)
      print(c1,c1/2000)

      1398 0.699
```

Note the '.evaluate' here used the trained weights in [19] '.fit'. It should be noted that as the weights and biases feeding into each layer forward are initially obtained by random draws from usually a small range (-1,+1), a MLP will produce slightly different predictive results for each new run unless we fix the random seed.[3] Re-sampling with each new epoch can also create a difference. Different results may be more obvious in NN with less depth and less width.

The TE1 in code line [46] of keras provides the vector of predicted probabilities in the context of binary cross entropy loss in predicting the binary

[3] This can be done by putting in the first code line of the notebook the following. import json \ import numpy as np \ import tensorflow as tf \ import keras \ from keras import layers \ from keras import initializers\ keras.utils.set_random_seed(5) ## set a seed number 5 or something else \ tf.config.experimental.enable_op_determinism()

class of “1” or “0”. The vector also translates into a vector of predicted 1’s or 0’s depending on whether the predicted probability (of “1”) is > 0.5 or not. These vectors can be used to compute the confusion matrix, the classification table, and the ROC AUC using sklearn.metrics.

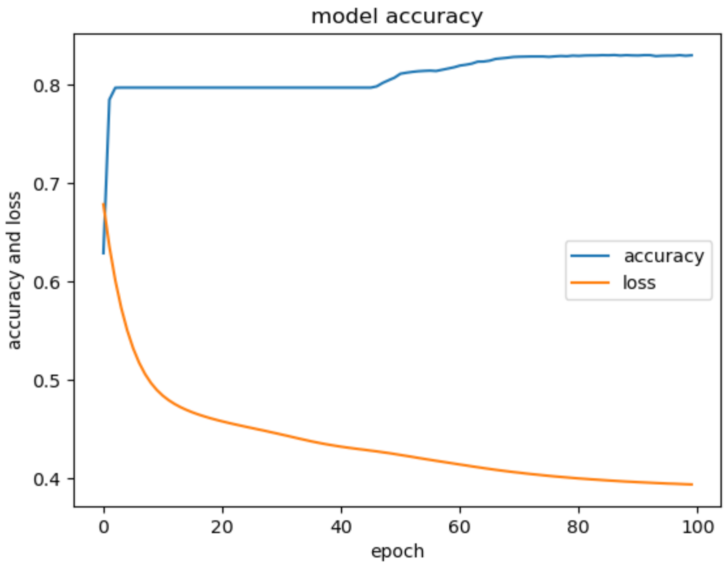
7.4 Improving the NN and Understanding the Results

Although the accuracy of the prediction of bank customer exit in the single hidden layer (2 neurons) MLP shown in the last section for the test data set is 69.90%, which is reasonably good, it can perhaps be improved. Also, sometimes a single batch may not yield robust or stable results. A telltale sign of the possibility of improvement is indicated in [21] where the training performance graph shows the loss is still decreasing and accuracy still climbing at the end of the training iterations.

We increase the number of backward passes and iterations, hence parameter revisions, by using “mini-batches” of batch size = 100 (instead of the entire batch size of 8000), and still using 100 epochs. See demonstration file Chapter7-2.ipynb.

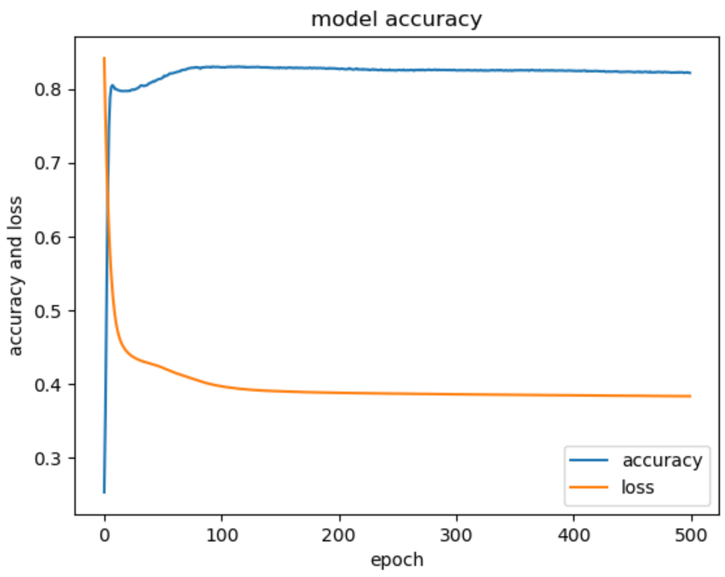
```
[18]: ### Last step in creation of NNmodel NNmodel is trained on the training set
      ###   here with Tensor-Keras .fit based on Compiler Fitting NNmodel
      history=NNmodel.fit(X_train,Y_train,batch_size=100,epochs = 100)
```

This yields the following loss and accuracy graph.



It is seen that after 1 epoch or 80 iterations (each epoch has 80 mini-batch iterations; 80 = 8000/100), the accuracy had already climbed to over 77%. The loss and accuracy stabilize at higher epochs, ending with training accuracy of about 82.99% (see output to code line [33], [43]). The test accuracy is 83.10% (see output to code line [46]). Both accuracies are better than those of 69.74% and 69.90% in the previous case of batch size = 8000 and 100 epochs. Hence higher number of iterations is useful to improve accuracy.

We also document the case of keeping batch size = 100 but increasing the number of epochs to 500. This created a total of 80 × 500 = 40,000 iterations. See demonstration file Chapter7-3.ipynb. This yields the following loss and accuracy graph.

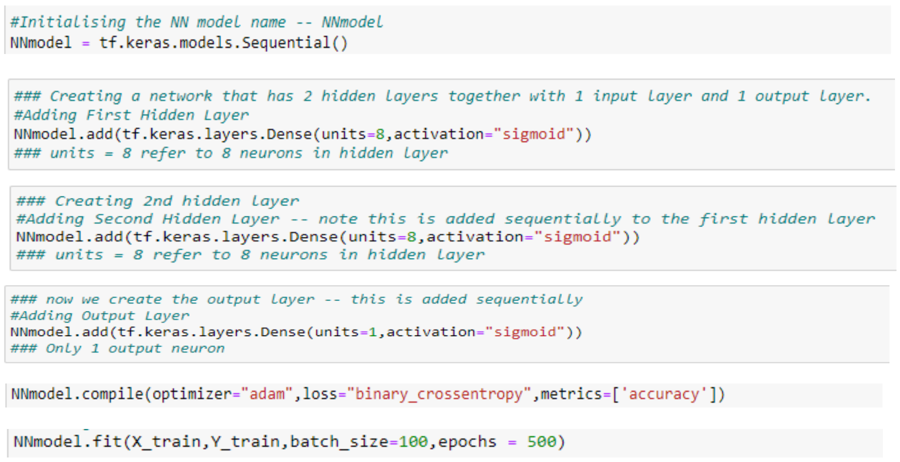


The loss and accuracy stabilize after about 6 epochs, ending with training accuracy of 82.25%. The test accuracy is 82.60%. Both accuracies are about similar to the case of batch size = 100 and 100 epochs. The initial few epochs saw a sharp hike in accuracy from a starting low of about 24%. The latter is possible as the initiating weights are random draws and could start off with first several iterates of large error/loss. Overall, it is seen that having 100 epochs in this case and a batch size of 100 may already clock in a sufficient number of iterations to adequately train the NN.

We employ the same batch size = 100 and 500 epochs, but attempt 3 different models (1), (2), and (3). In (1) (see demonstration file Chapter7-

4.ipynb), we adjust the optimizer to stochastic gradient descent (SGD) instead of Adam. In (2) (see demonstration file Chapter7-5.ipynb), we add a second hidden layer with now 4 neurons in both the first and second hidden layers. We create an even deeper NN in (3) (see demonstration file Chapter7-6.ipynb) where the number of neurons in the first and the second hidden layers are increased from 4 to 8. In (3) there is therefore a total of (12+1) × 8 = 104 neurons in the first hidden layer, (8+1) × 8 = 72 neurons in the second hidden layer, and 8 + 1 = 9 neurons in the output layer.

For (3), the code lines for building the NN are shown as follows. Note how the second hidden layer is built in the Keras-Sequential algorithm.



```
#Initialising the NN model name -- NNmodel
NNmodel = tf.keras.models.Sequential()
```

```
### Creating a network that has 2 hidden layers together with 1 input layer and 1 output layer.
#Adding First Hidden Layer
NNmodel.add(tf.keras.layers.Dense(units=8,activation="sigmoid"))
### units = 8 refer to 8 neurons in hidden layer
```

```
### Creating 2nd hidden layer
#Adding Second Hidden Layer -- note this is added sequentially to the first hidden layer
NNmodel.add(tf.keras.layers.Dense(units=8,activation="sigmoid"))
### units = 8 refer to 8 neurons in hidden layer
```

```
### now we create the output layer -- this is added sequentially
#Adding Output Layer
NNmodel.add(tf.keras.layers.Dense(units=1,activation="sigmoid"))
### Only 1 output neuron
```

```
NNmodel.compile(optimizer="adam",loss="binary_crossentropy",metrics=['accuracy'])
```

```
NNmodel.fit(X_train,Y_train,batch_size=100,epochs = 500)
```

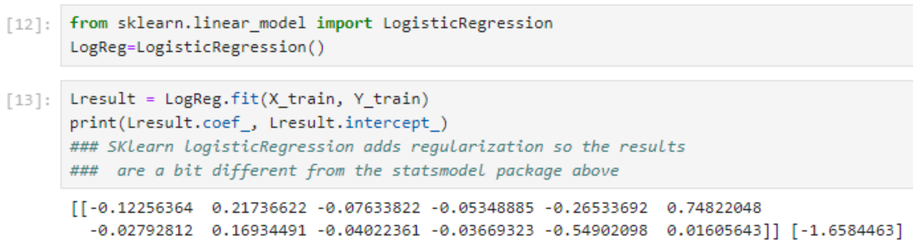
The minimized (after the specified number of iterations) loss value and the accuracy of the predictions for the different models of (1), (2), (3) are reported below. Note the results may change a little for any new run since the initialization of the parameters is random and each epoch may see a re-sampling.

Metrics:	(1)	(2)	(3)
Training Set			
Loss	0.3951	0.3355	0.3229
Accuracy (%)	83.00%	86.00%	86.46%
Test Set			
Loss	0.3879	0.3331	0.3280
Accuracy (%)	83.10%	86.30%	86.75%

In (1), it is seen that the SGD optimizer performs about the same as Adam with the same number of iterations in batch size = 100, epochs = 500, one hidden layer and 2 neurons in the layer. The former, (1), has slightly larger loss but slightly higher accuracy. Model (2) with 2 hidden layers of 4 neurons each improves accuracy by about 3% compared to the case of only 1 hidden layer. The deeper NN with 8 neurons in each hidden layer in (3) improves further by about 0.5% on both training and test accuracies from (2).

During the training, pruning or reducing the total number of neurons or nodes could occur in order to improve the prediction performances. tensorflow_model_optimization module in tensorflow can be used to prune away weights that are close to zero. The weights from one layer to another can be viewed as matrices. When the weights are too small and close to zero, they can be seen as sparse matrices. The module can prune based on indicated final level of sparsity to be attained where sparsity is the percentage number of weights that is set to zero. In general, pruned NN can perform better.

We also tap on the Scikit-learn (Sklearn) python library to use the LogisticRegression codes to perform classification test on the training data set. Logistic regression performance outcomes can be used for baseline comparison. See demonstration file Chapter7-7.ipynb. The code lines are shown as follows.

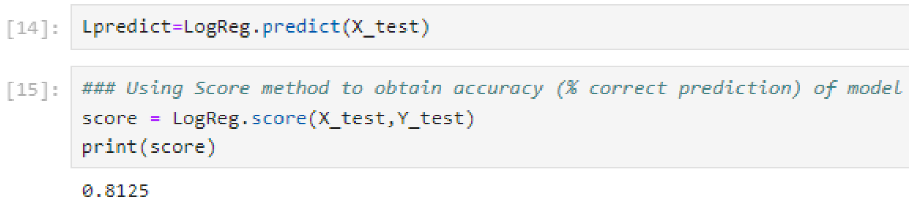


```
[12]: from sklearn.linear_model import LogisticRegression
      LogReg=LogisticRegression()

[13]: Lresult = LogReg.fit(X_train, Y_train)
      print(Lresult.coef_, Lresult.intercept_)
      ### SKlearn LogisticRegression adds regularization so the results
      ###  are a bit different from the statsmodel package above

      [[-0.12256364  0.21736622 -0.07633822 -0.05348885 -0.26533692  0.74822048
        -0.02792812  0.16934491 -0.04022361 -0.03669323 -0.54902098  0.01605643]] [-1.6584463]
```

Code lines [14], [15] show the prediction accuracy of the logistic regression method.



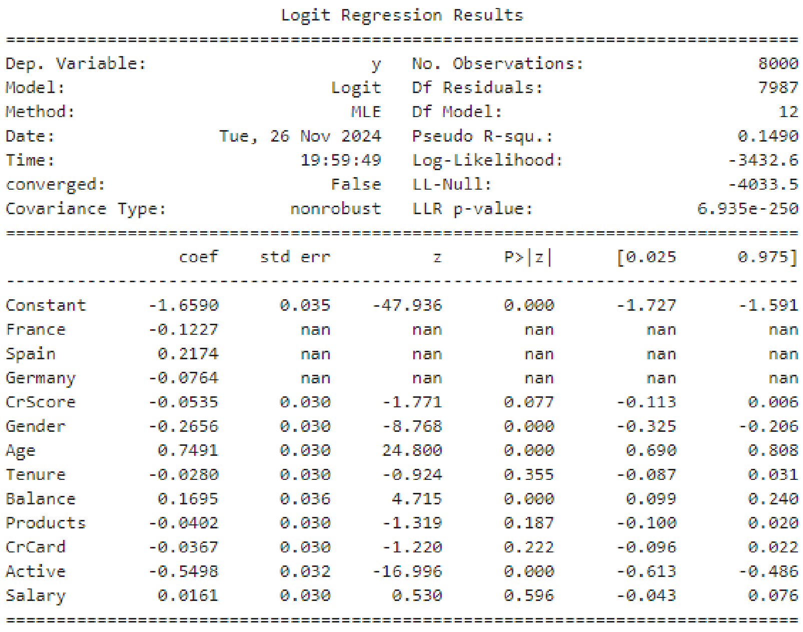
```
[14]: Lpredict=LogReg.predict(X_test)

[15]: ### Using Score method to obtain accuracy (% correct prediction) of model
      score = LogReg.score(X_test,Y_test)
      print(score)

      0.8125
```

The prediction accuracy using the estimated logit regression coefficients (where estimated probability > 0.5 is designated as predicting 1, otherwise 0) shows 81.25%. This is better than the first case in the last section with 1 hidden

layer and epoch = 100. It is just slight below case (1) with 1 hidden neural network layer but batch size = 100 and epoch = 500. The better trained and deeper NNs in (2) and (3) appear to perform better than logistic regression prediction.
We also perform a logit regression of the exit variable (1 or 0) on the feature variables using package ‘statsmodel’ in python. See code line [19]. The regression outputs are summarized as follows. Note that sklearn Logistic Regression adds regularization, so the results are a little bit different from statsmodel package results shown here.

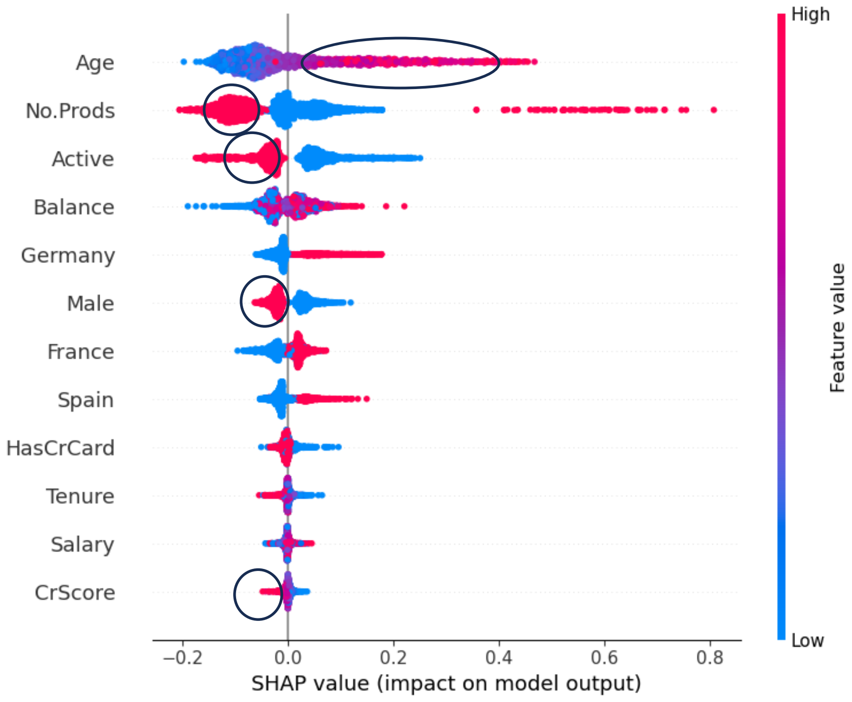


```
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 8000
Model:                          Logit   Df Residuals:                     7987
Method:                           MLE   Df Model:                           12
Date:                Tue, 26 Nov 2024   Pseudo R-squ.:                  0.1490
Time:                        19:59:49   Log-Likelihood:                -3432.6
converged:                      False   LL-Null:                       -4033.5
Covariance Type:            nonrobust   LLR p-value:                6.935e-250
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Constant      -1.6590      0.035    -47.936      0.000      -1.727      -1.591
France        -0.1227        nan        nan        nan         nan         nan
Spain          0.2174        nan        nan        nan         nan         nan
Germany       -0.0764        nan        nan        nan         nan         nan
CrScore       -0.0535      0.030     -1.771      0.077      -0.113       0.006
Gender        -0.2656      0.030     -8.768      0.000      -0.325      -0.206
Age            0.7491      0.030     24.800      0.000       0.690       0.808
Tenure        -0.0280      0.030     -0.924      0.355      -0.087       0.031
Balance        0.1695      0.036      4.715      0.000       0.099       0.240
Products      -0.0402      0.030     -1.319      0.187      -0.100       0.020
CrCard        -0.0367      0.030     -1.220      0.222      -0.096       0.022
Active        -0.5498      0.032    -16.996      0.000      -0.613      -0.486
Salary         0.0161      0.030      0.530      0.596      -0.043       0.076
==============================================================================
```

The estimated coefficients and their statistical significance provide some observations to help understand how the features influence the prediction of ‘exit’ in the NN. Recall that if a customer had closed his/her bank account during the data period, then the classification of exit is “1”. Hence the predicted probability of Class “1” being higher means that it would be more likely that the customer closed his/her bank account. Predicted probability of Class “1” being lower means that the customer is likely to continue his/her bank account.

It is seen that (at 10% 2-tailed significance level or a p-value of < 0.10), higher credit score (CrScore), Gender being Male, lower Age, being an active customer with the bank are features that predicted a lower probability of exit. A larger bank balance, however, predicted a higher probability of exit.

In Chapter7-8.ipynb, model (3) with 8 neurons in the first and the second hidden layer, and an output layer, the Shap values are obtained as follows.



It is seen that instances/individuals that have feature values that are circled such as using a large number of the bank products ("No.Prods"), being active, being Male, and having a high credit score ("CrScore") have negative Shap values for the features. These features directly contributed to lower predicted probability of exit. Instances/individuals that have higher ages show the age feature producing positive Shap values, i.e., higher age contributed to higher predicted probability of exit from the bank. The Shap value results are consistent with the coefficient estimates from the logistic regression. Moreover, for each instance/individual, we can find how much does each feature contributed to the predicted probability of exit, whether increasing it or decreasing it.

7.5 Concluding Thoughts

One major application of ML is the use of neural network (NN) that processes input information and produces output(s) of prediction and classification. The Multi-Layer Perceptron (MLP) or Artificial Neural Network (ANN) is seen to be able to predict close to 87% accuracy in the bank customer exit data set when the NN is adequately structured with sufficiently deep number of hidden layers and adequate training iterations. When the training accuracies are high, it is more likely that the testing accuracies are also high. The trained NN can also help to provide understanding of how the various features affect the final output such as the predicted probability in the case of classification.

7.6 Other References

Diederik P. Kingma and Jimmy Lei Ba. Adam : A method for stochastic optimization. 2014. arXiv:1412.6980v9

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141

https://www.v7labs.com/blog/neural-network-architectures-guide

https://www.andreaperlato.com/aipost/adaptive-momentum/

https://towardsdatascience.com/how-not-to-use-machine-learning-for-time-series-forecasting-avoiding-the-pitfalls-19f9d7adf424

https://www.analyticsvidhya.com/blog/2021/10/machine-learning-for-stock-market-prediction-with-step-by-step-implementation/

https://machinelearningmastery.com/build-multi-layer-perceptron-neural-network-models-keras/

https://machinelearningmastery.com/prediction-intervals-for-deep-learning-neural-networks/