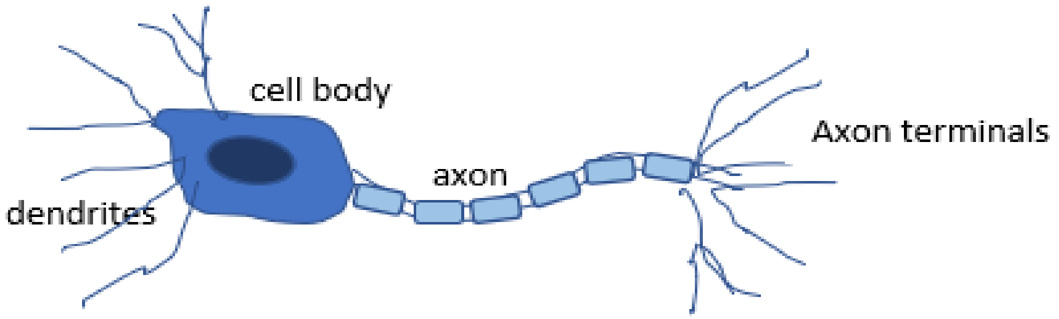
QF634 APPLIED QUANTITATIVE RESEARCH METHODS
LECTURE 7

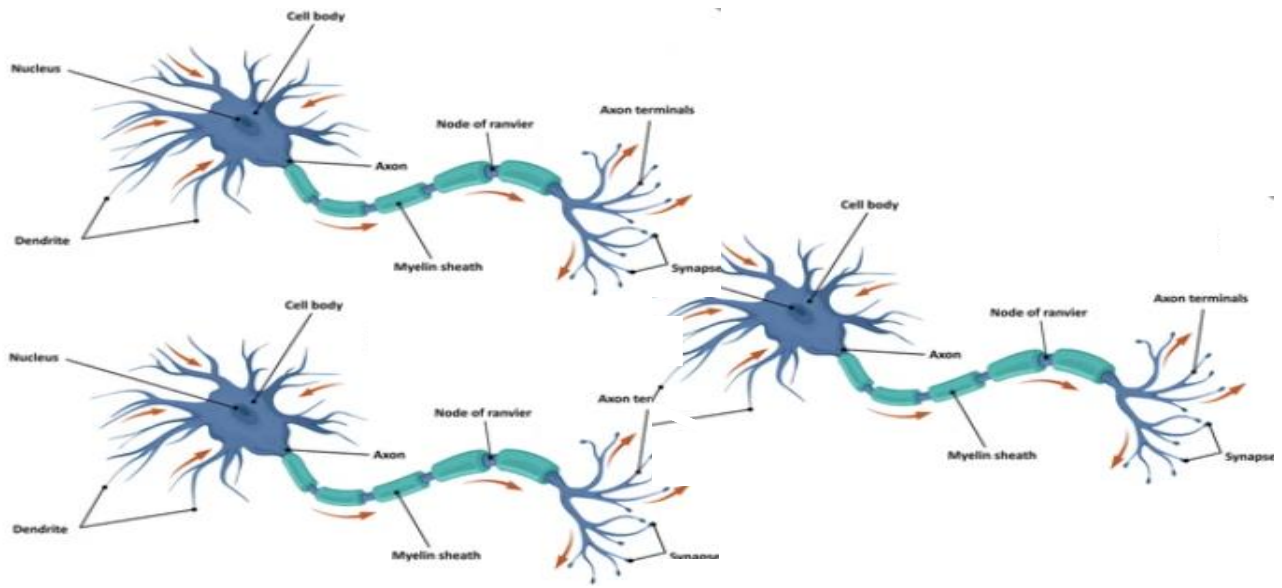
Lecturer: Prof Emeritus Lim Kian Guan

ARTIFICIAL INTELLIGENCE

Artificial Neural Network I

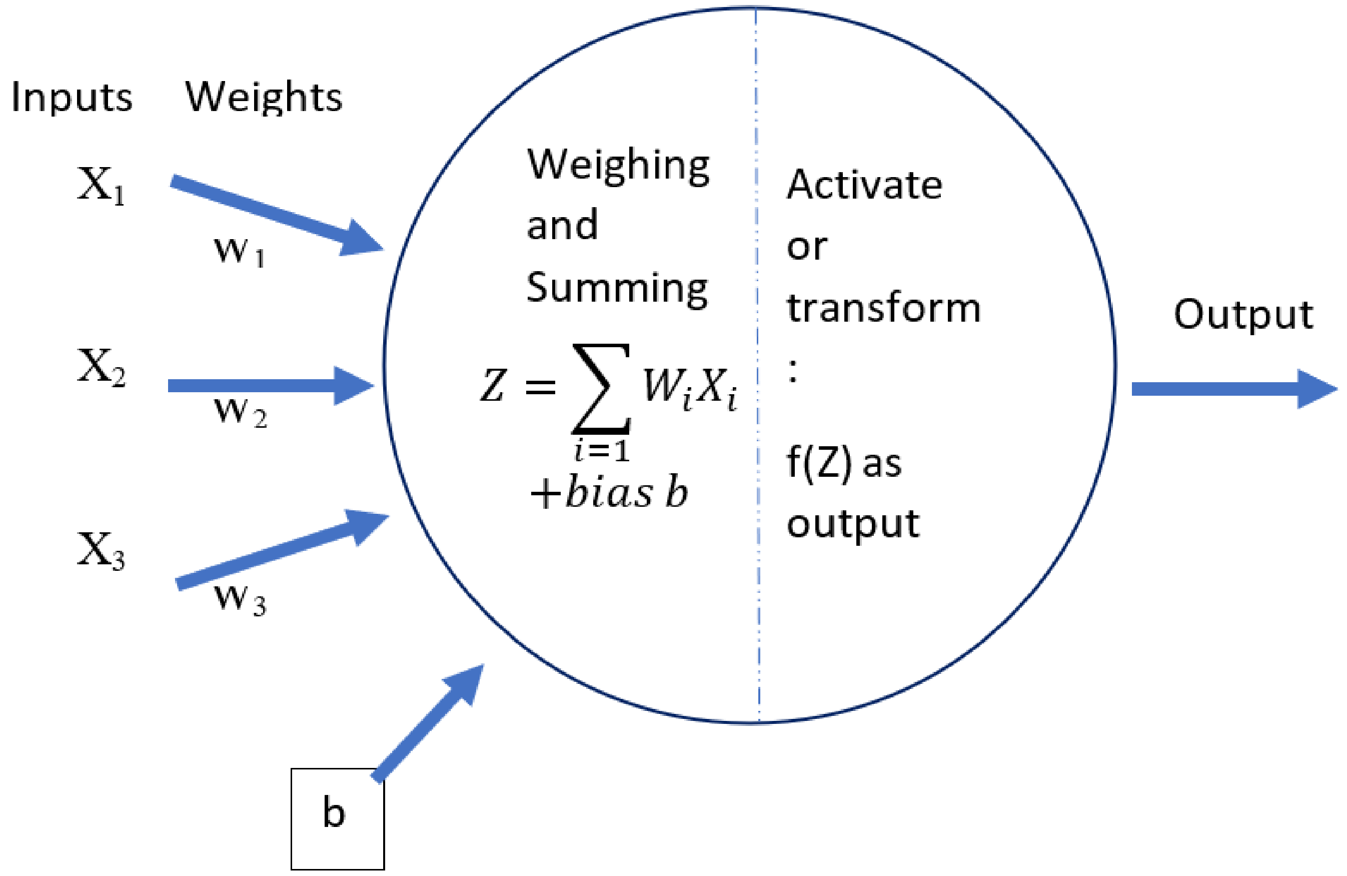
Neurons – Nerve Cells in the Brain





- Neurons are information messengers. They transmit information between different areas of the brain and the rest of the nervous system.
- Neurons communicate with each other by sending neurotransmitters, across a tiny spaces – synapse -- between the axons and dendrites of adjacent neurons.
- Neurons can connect with other neurons to form a chain of complex input and output responses.

Single Layer Perceptron



- Takes inputs or features of a subject or case, $\{X_1, X_2. X_3\}$
- Aggregates it with corresponding weights $\{W_1, W_2, W_3\}$, adds a bias term b

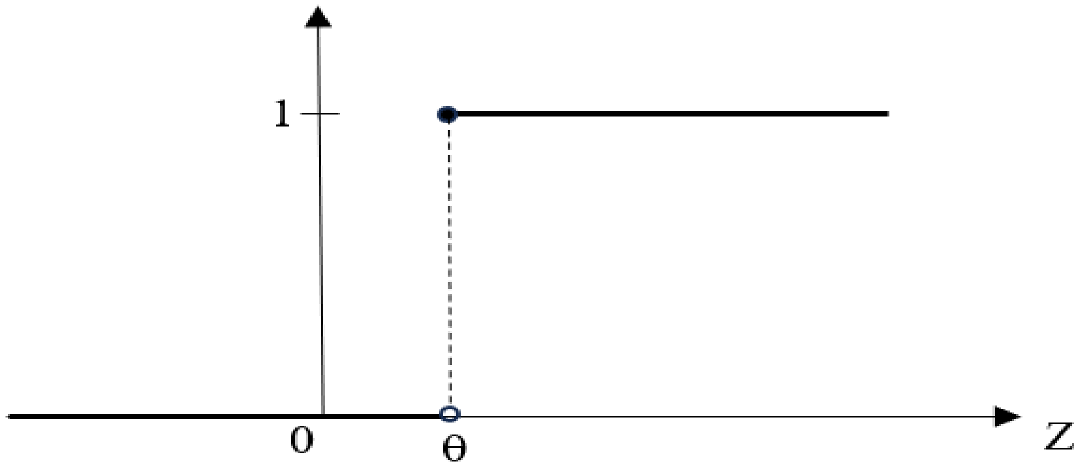
$$\sum_{i=1}^{3} W_i X_i + \mathrm{b}$$

- Then computes the output

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

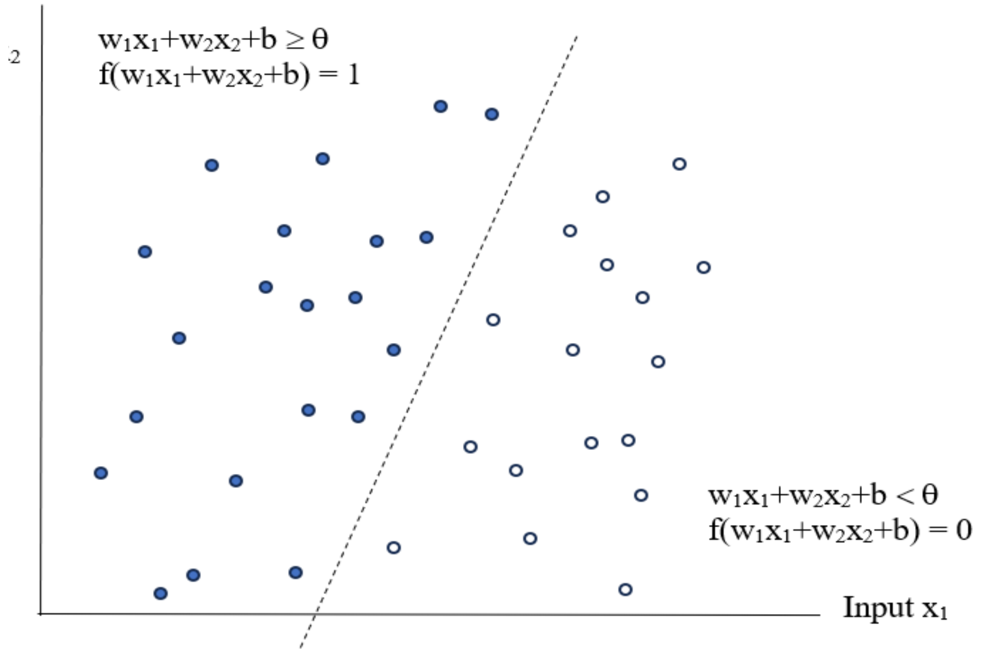
- The bias (or adjustment of mean) can be treated like an input of 1 with weight b.
- Weights W_j's in a NN are initiated typically randomly as small positive or negative numbers, e.g., between (-1,+1).

Example of Activation Function in a Single Layer Perceptron

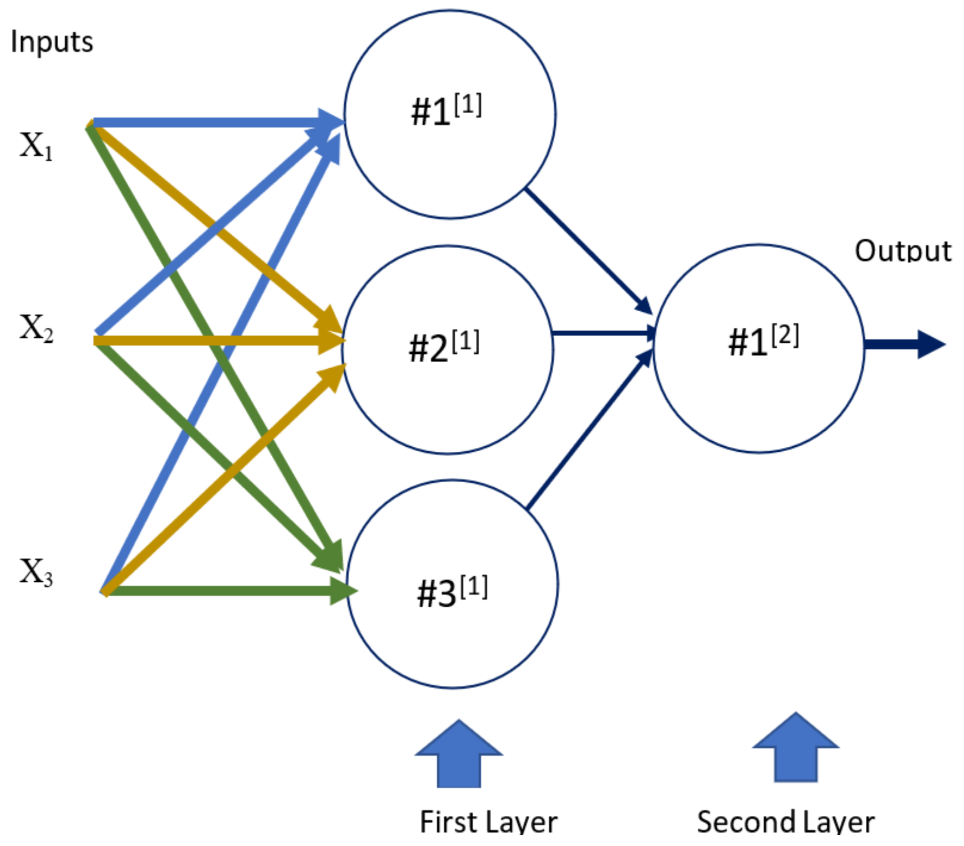


$f(z) = 1_{z \geq \text{threshold } \theta}$

This kind of binary output is typical in a linear binary classifier – here's an example where there are two inputs



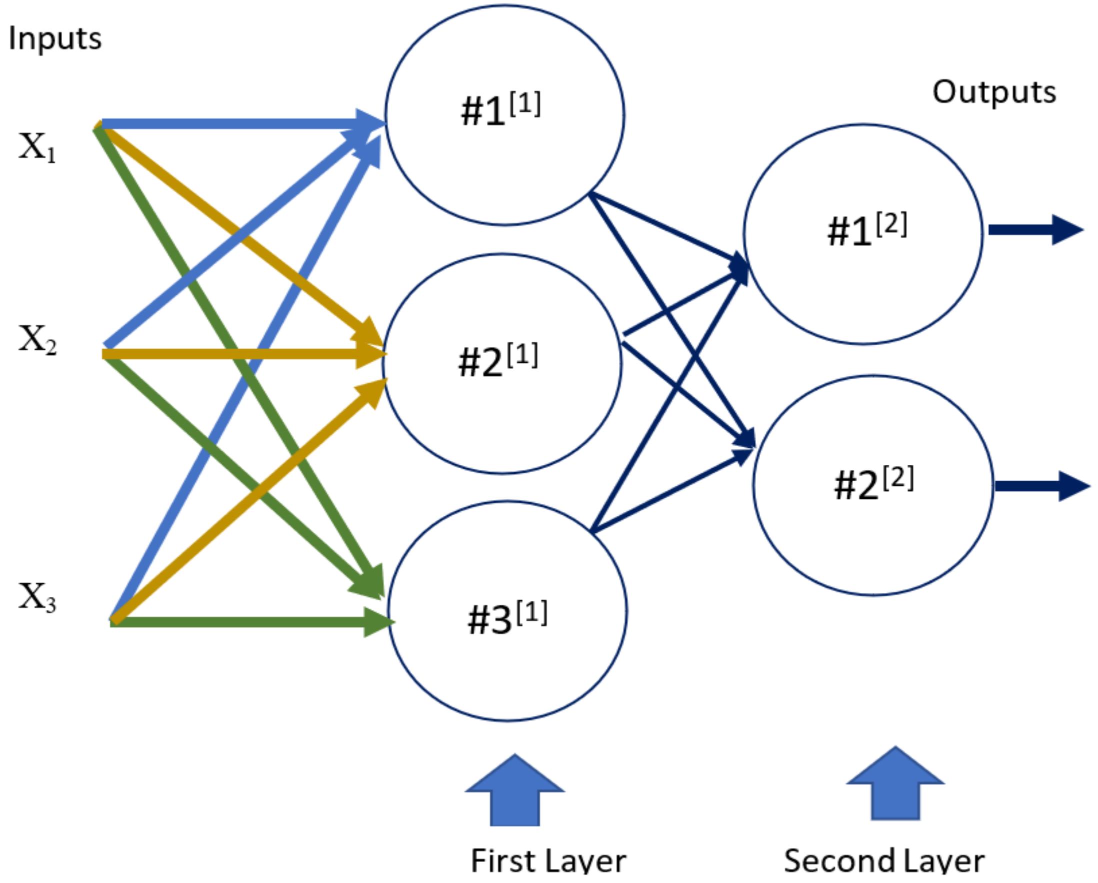
MultiLayer Perceptron (MLP)



Many-to-One ANN (Artificial Neural Network)

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.

MultiLayer Perceptron (MLP)

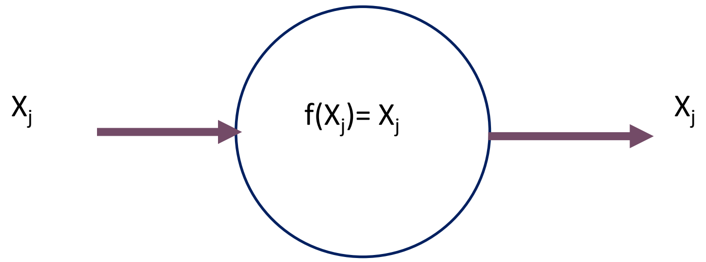


Many-to-Many ANN (Artificial Neural Network)

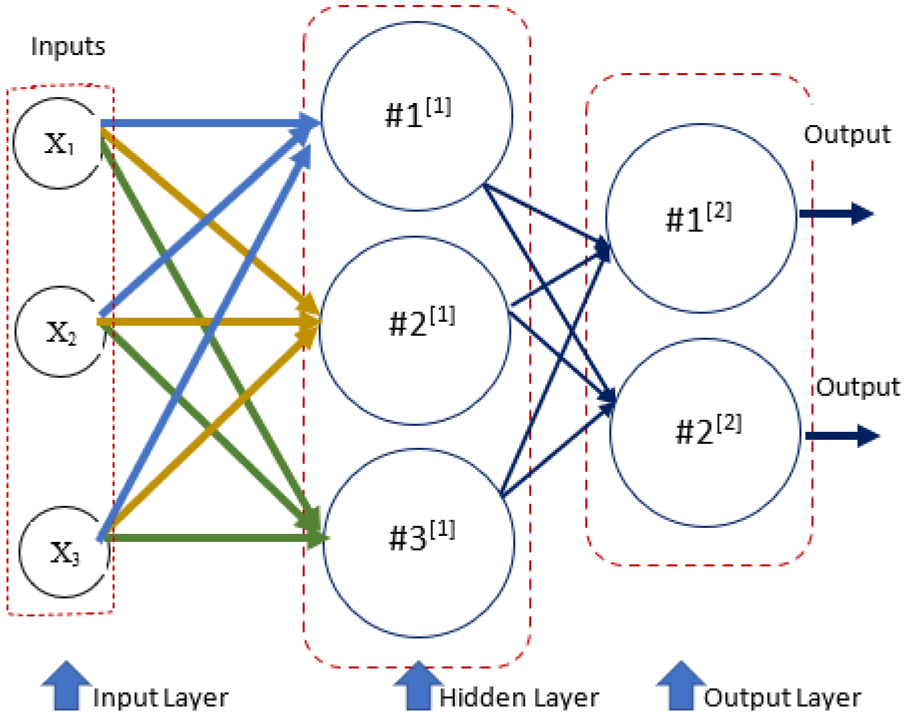
For a different problem, the structure could also be a Many-to-Many ANN where there are now two nodes or neurons in the second (output) layer and then two outputs accordingly from neurons #1[2] and #2[2].

No. of output nodes is one if the model is training to predict a single number or a binary classification. But no. of nodes can be more than one if the classification is not just binary but many categories.

MultiLayer Perceptron (MLP)



If we consider an input X_j producing identical output X_i (with a trivial identity activation function), then we can treat an input as a neuron.

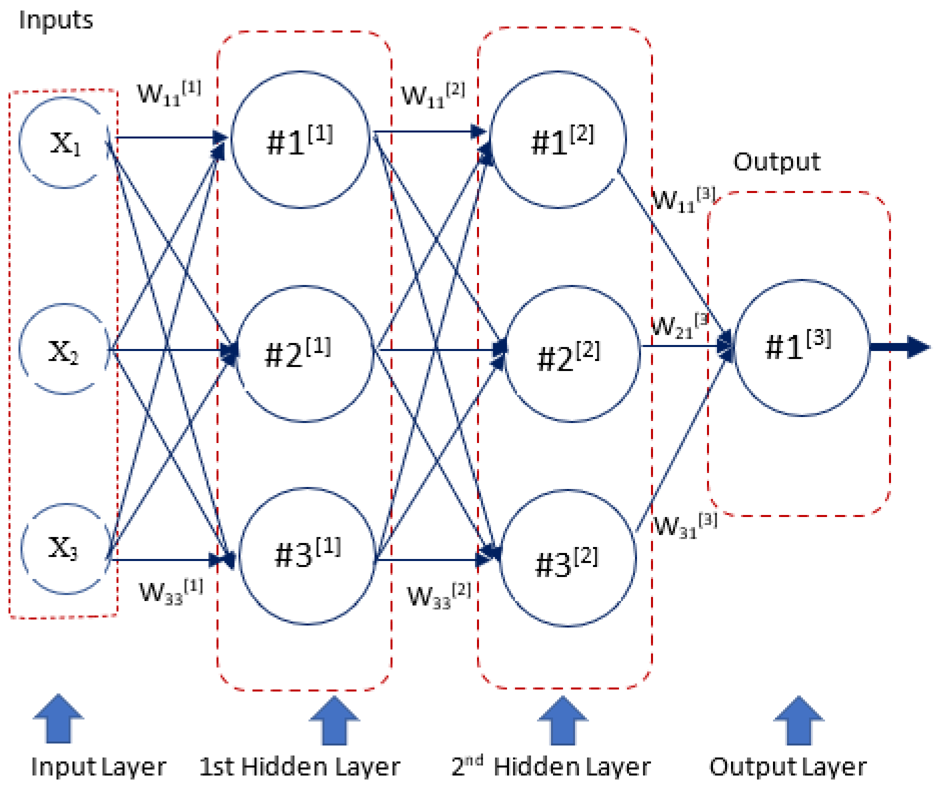


Middle layer called "hidden layer" as the computed outputs from the nodes in this layer are not explicitly observed

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).

Each neuron will connect with all neurons in the forward pass process.

MultiLayer Perceptron (MLP) – weights and biases



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 $W_{31}^{[3]}$. In general, the weight from nth hidden layer #j[n] neuron to the next layer #k[n+1] neuron is $w_{jk}^{[n+1]}$.

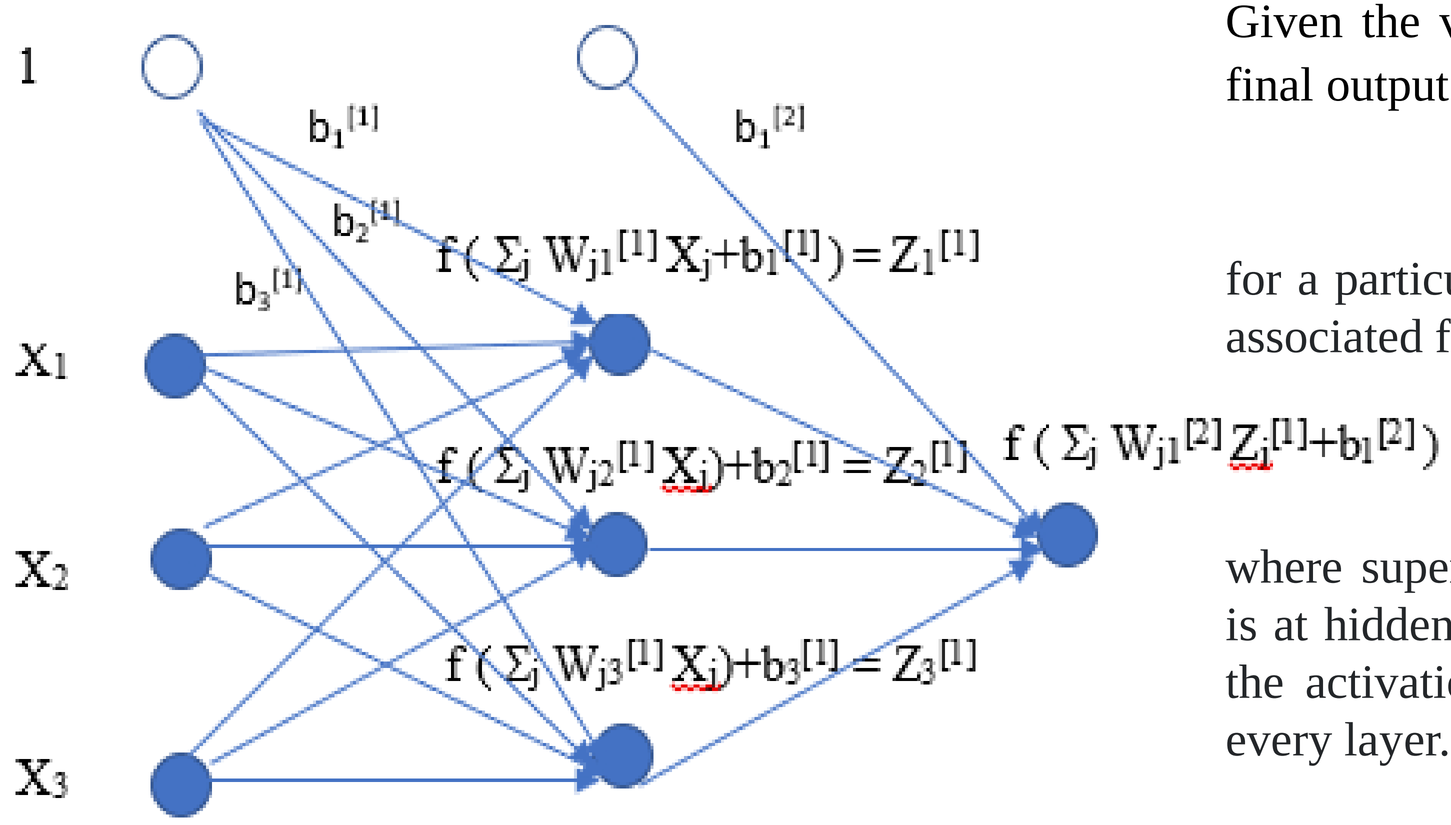
Artificial Neural Network

- Purpose is to perform prediction or classification on a set of data containing subjects or cases with associated features or characteristics.
- To prepare a model for prediction with new data, the model must be trained using many known cases/subjects with their known target statuses and with associated features/characteristics or conditions in the training data set.
- Once the model is adequately trained, the weights are determined optimally by this stage, and the ANN can perform the necessary computations based on these optimal weights (including optimal biases) and the fresh inputs to find the predicted output(s) or target value(s) in the test data set.

Artificial Neural Network

- The ANN computes the output(s) in the output layer in a forward pass process called Forward Propagation. 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.
- 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. The total number of iterations is equal to the number of epochs multiplied by the number of mini batches in each epoch.

Forward Propagation in a Single Hidden Layer MLP


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

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.

Examples of Nonlinear Activation Function in a Single Layer Perceptron

- Logistic Sigmoid Function (S-Curve)

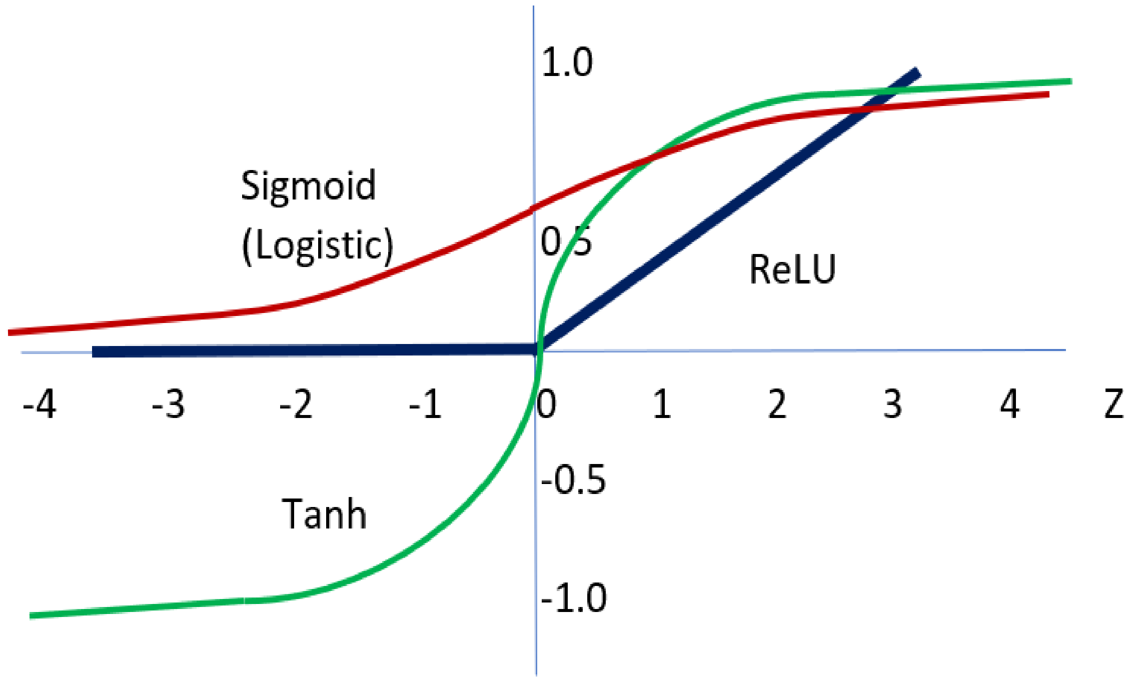
  $f\ (Z) = (1 + e^{-Z})^{-1}$

- Hyperbolic Tangent Function (S-Curve)

  $f\ (Z) = (e^{Z} - e^{-Z}) / (e^{Z} + e^{-Z})$

- Rectifier Linear Unit Function (ReLU)

  $f\ (Z) = \max\ (Z, 0)$



Error or Loss Function for Continuous Variable Target

- 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.
- 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.
- Commonly used error/loss function is the average least squares of deviations (or mean square error, MSE) L^2:

$$L\left(Y,\hat{Y}\right) \equiv \frac{1}{N}\sum_{k=1}^{N} L\left(Y_k,\hat{Y}_k\right) = \frac{1}{N}\sum_{k=1}^{N}\left(Y_k-\hat{Y}_k\right)^2$$

- An alternative is the mean absolute loss or L^1 loss function

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

- In scenarios where there are outliers, MSE >> MAE. 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.

Error or Loss Function for Categorical Variable Target

- 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

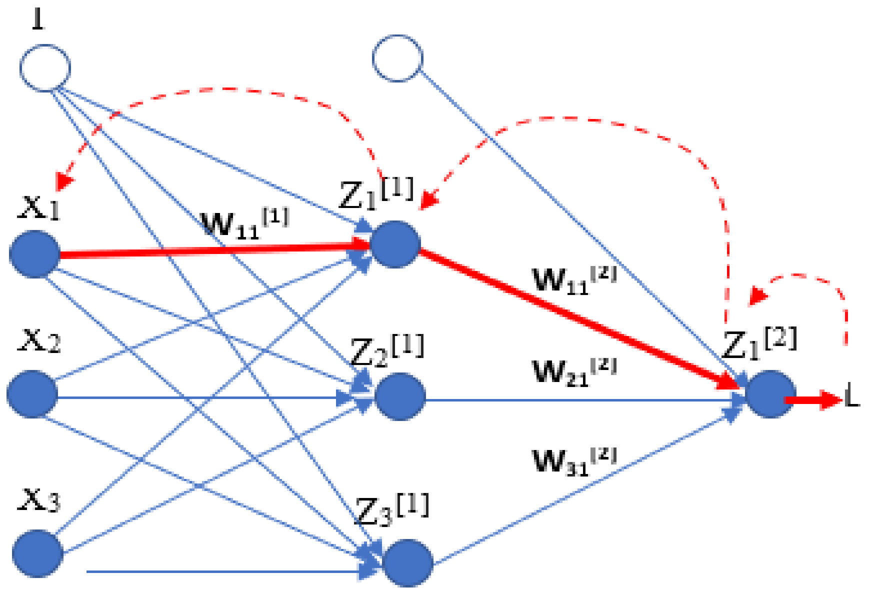
$$L(Y_k, \hat{P}_k) = -[\, Y_k\, ln\, \hat{P}_k + (1-Y_k) \ln (1 - \hat{P}_k)\,].$$

$\hat{P}_k$ 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.

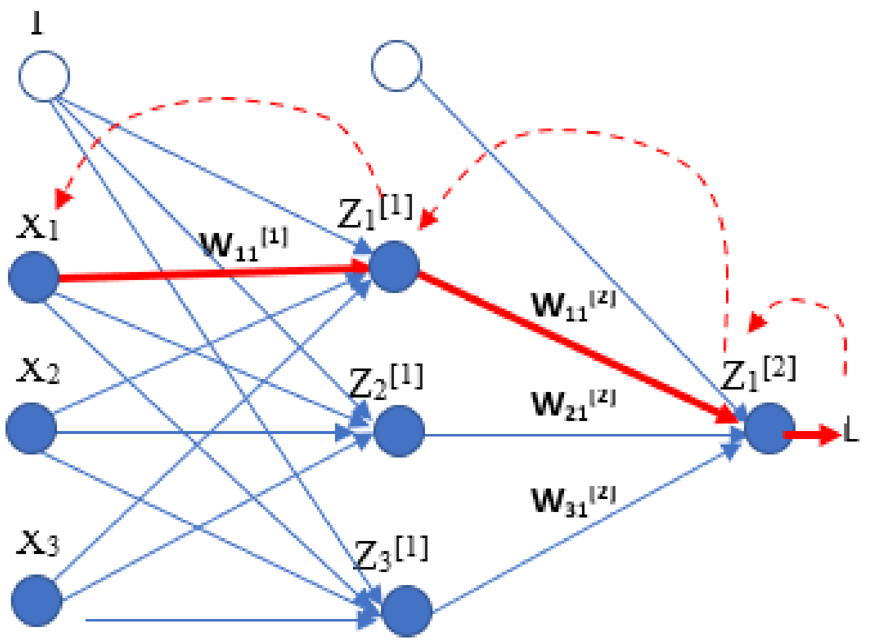
Backward Propagation

- 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.
- When loss is not acceptable, the model needs further improvement – we need to train the NN further, by updating/revising/fine-tuning the weights and bias and then run a second epoch, and so on until it hopefully ends 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 test data set.
- Backward propagation involves backwards pass (dotted red arrows) that performs computations by moving backward from the loss function at the final output to the weight or bias inputs.

Backward Propagation



- In this structure there are altogether 16 parameters of weights and biases to update/revise/fine-tune: {$W_{11}^{[1]}$, $W_{21}^{[1]}$, $W_{31}^{[1]}$, $b_1^{[1]}$, $W_{12}^{[1]}$, $W_{22}^{[1]}$, $W_{32}^{[1]}$, $b_2^{[1]}$, $W_{13}^{[1]}$, $W_{23}^{[1]}$, $W_{33}^{[1]}$, $b_3^{[1]}$, $W_{11}^{[2]}$, $W_{21}^{[2]}$, $W_{31}^{[2]}$, $b_1^{[2]}$}.
- 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

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

For each k, for m = 1 ,2 3

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

Backward Propagation

Similarly,

$$\frac{\partial L\left(Y_k,\hat{Z}_k\right)}{\partial W_{m2}^{[1]}}$$

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

for m = 1,2,3.

$$\frac{\partial L\left(Y_k,\hat{Z}_k\right)}{\partial W_{m3}^{[1]}}$$

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

for m = 1,2,3.

Backward Propagation

And

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

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

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

and so on.

Backward Propagation

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

Note that if f(X) = 1/(1 + e^{-X}), df(X)/dX = e^{-X}/(1 + e^{-X})^2 = (1-f(X)) f(X).

Note that $Z_{1,k}^{[2]} = \text{f}(\sum_j (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]}),\ Z_{1,k}^{[2]} \equiv \hat{Y}_k$. Then,

$$\frac{\partial Z_{1,k}^{[2]}}{\partial \sum_j (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})} = (1-\hat{Y}_k)\, \hat{Y}_k \qquad \frac{\partial \sum_j (W_{j1}^{[2]} Z_{j,k}^{[1]} + b_1^{[2]})}{\partial Z_{1,k}^{[1]}} = W_{11}^{[2]}$$

$$\frac{\partial Z_{1,k}^{[1]}}{\partial \sum_j (W_{j1}^{[1]} X_j + b_1^{[1]})} = (1-Z_{1,k}^{[1]})\, Z_{1,k}^{[1]} \qquad \frac{\partial \sum_j (W_{j1}^{[1]} X_j + b_1^{[1]})}{\partial W_{11}^{[1]}} = X_1.$$

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

Backward Propagation

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, $\partial L/\partial W > (<) \ 0$, 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) 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 small loss function form the gradient descent approach in the loss function minimization.

Convergence Strategy to Minimum Loss

- To attain a (reasonably) minimum error/loss function with not a massive number of epochs, 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 $n < N$. In this way, the computational memory burden for each iteration involving only n sample points and correspondingly fewer partial derivatives is lesser. This is called the ‘mini-batch’ gradient descent approach.
- 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 $E \times N/n$ number of iterations in updating the parameters in the training. This is called the ‘mini-batch’ gradient descent approach.
- Another popular variant is the stochastic gradient descent (SGD) that randomly and 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. However, there are far more iterations.

Convergence Strategy to Minimum Loss

- Another approach to speed convergence is to adjust the learning rate. One such popular algorithm is the 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. M_t and V_t are moving averages. $G_t = \partial L / \partial W_{ij,t}^{[r]}$. M_0 and V_0 are typically initialized at 0. M_t and V_t are replaced by unbiased $\widehat{M}_t = M_t/(1-\beta_1^t)$ and $\hat{V}_t = V_t/(1-\beta_2^t)$ so $E(\widehat{M}_t) \approx E(G_t)$ and $E(\hat{V}_t) \approx E(G_t^2)$.

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

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

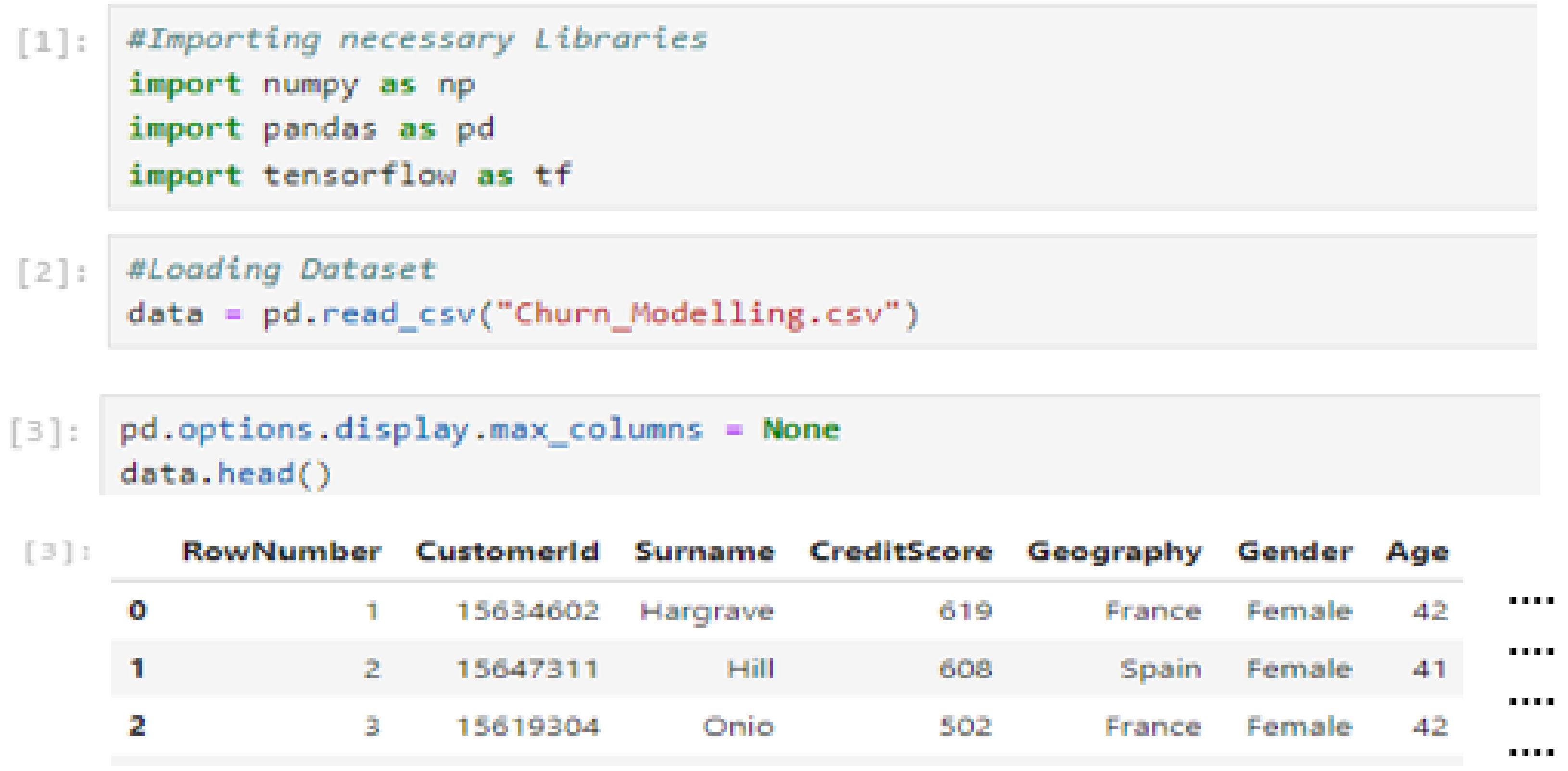
where given a step size θ (hyperparameter to be tuned – it is typically a small positive number), adjustment or descent is faster with higher $\widehat{M}_t$ and lower $\hat{V}_t$ (more persistent gradient in one direction with less variability). In most implementations, default values for β_1 and β_2 are 0.9 and 0.999. ε is included to prevent G_t (hence also G_t^2) disappearing to 0 when division becomes not possible in the second term. Typical value for ε is 10^{-8}.

Worked Example – Data

Please upload Chapter7-1.ipynb and follow the computing steps in Jupyter Notebook

- 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), bank balance in the period of the data, the number of 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 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".
- So we are dealing with a binary classification prediction.

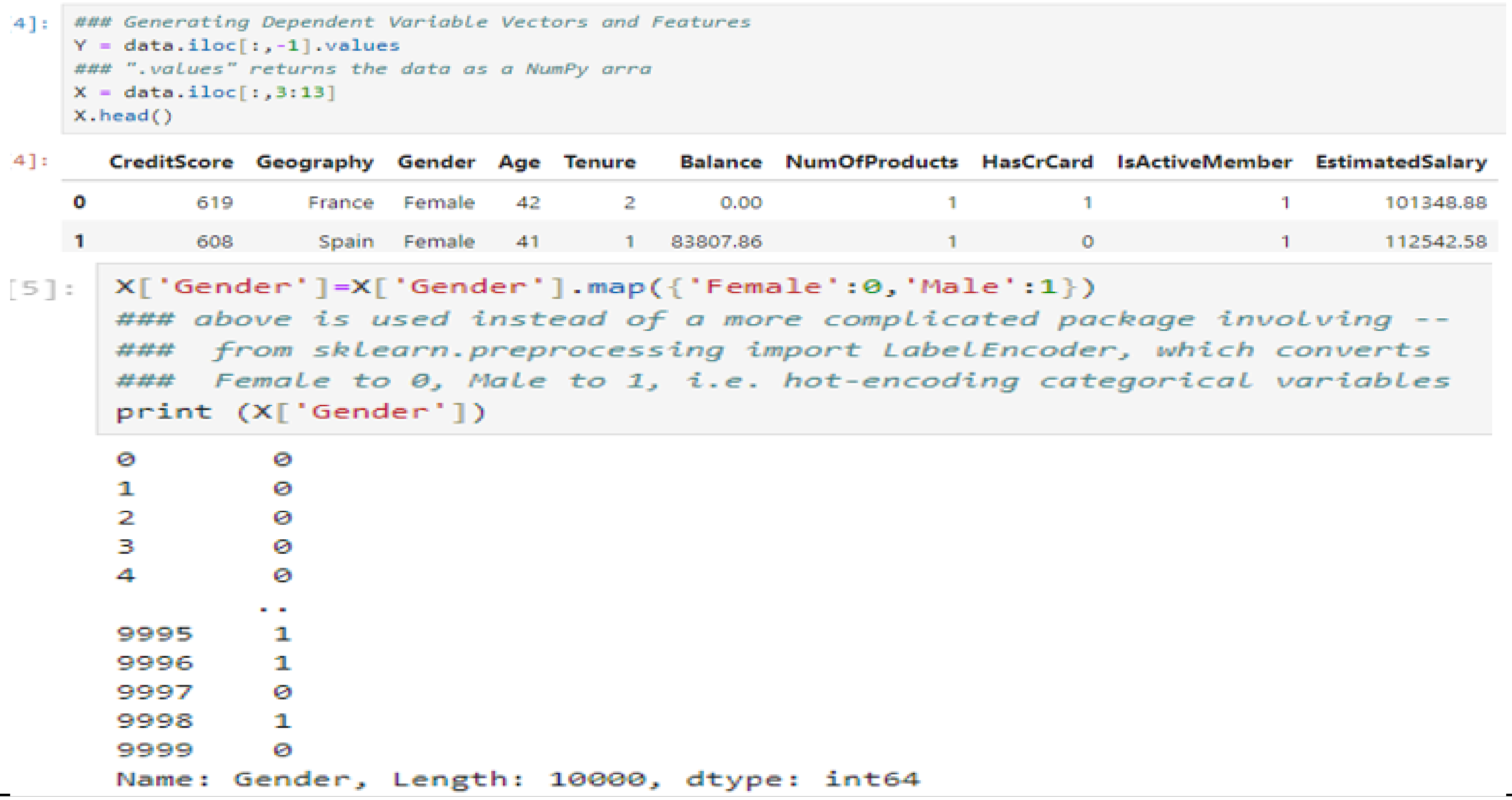
Please upload Chapter7-1.ipynb and follow the computing steps in Jupyter Notebook



```
[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   ....
                                                                                ....
```

Please upload Chapter7-1.ipynb and follow the computing steps in Jupyter Notebook

In code lines [4] and [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

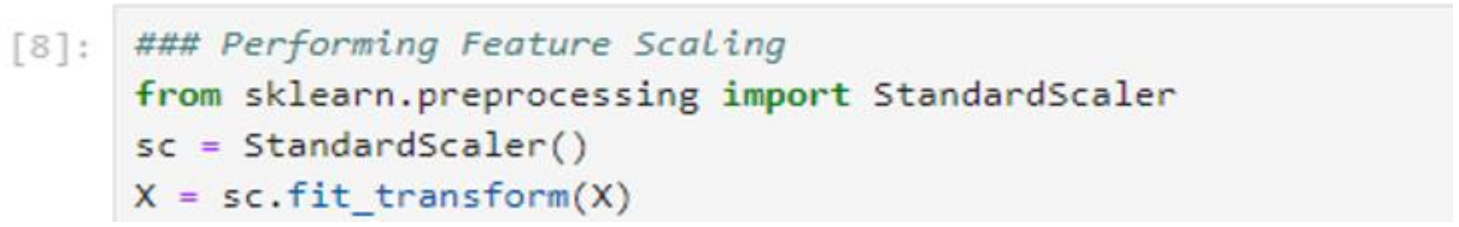
```
[5]: X['Gender']=X['Gender'].map({'Female':0,'Male':1})
     ### above is used instead of a more complicated package involving --
     ###  from sklearn.preprocessing import LabelEncoder, which converts
     ###  Female to 0, Male to 1, i.e. hot-encoding categorical variables
     print (X['Gender'])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

```
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
```

Please upload Chapter7-1.ipynb and follow the computing steps in Jupyter Notebook

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.

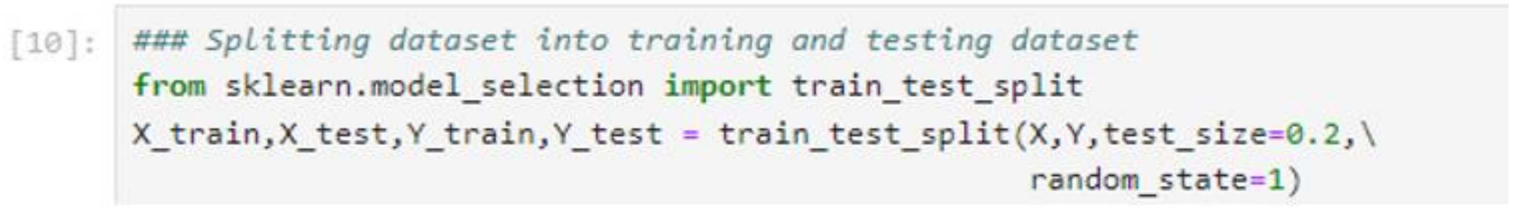
Each feature in the training set is scaled to mean 0 and variance 1.
In sklearn.preprocessing.StandardScaler(), centering and scaling operations happen independently on each feature. The fit method is calculating the mean and variance of each of the features present in the data. The transform method 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.



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

Please upload Chapter7-1.ipynb and follow the computing steps in Jupyter Notebook

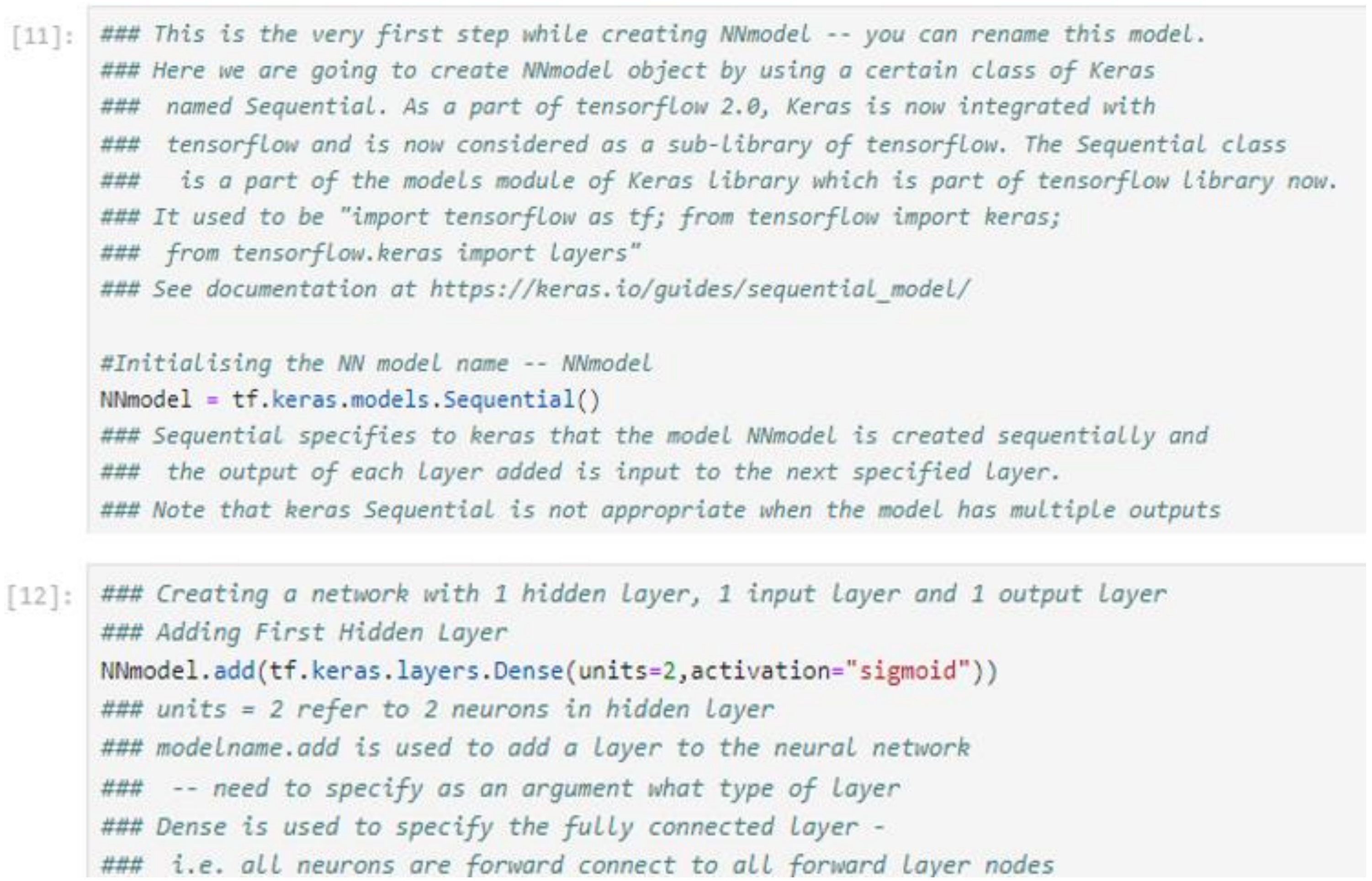
Code line [10] shows split of the dataset into training (80%) and test (20%) data sets. Note that it is assumed a validation set would have been already used to find appropriate hyperparameters, including the type of hyperparameter computation such as “Adam”. Then the training and validation data sets are re-grouped into this current training data set for training and then testing.



```
[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)
```

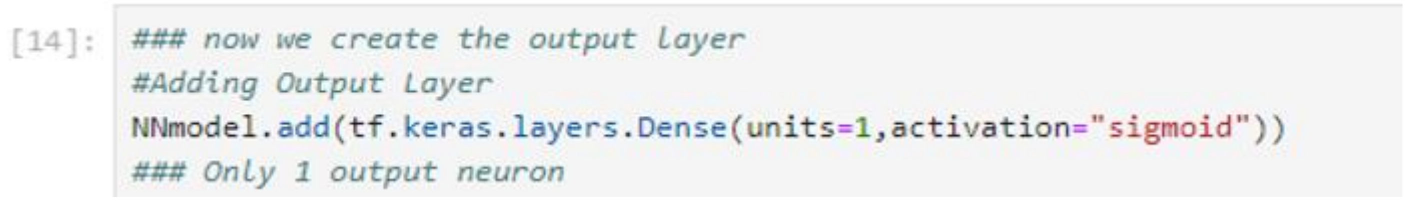
Code lines [11], [12] use the tf.keras.models.Sequential() API to build the MLP neural network model.

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. For the second input, we try the sigmoid as activation function for hidden layers.



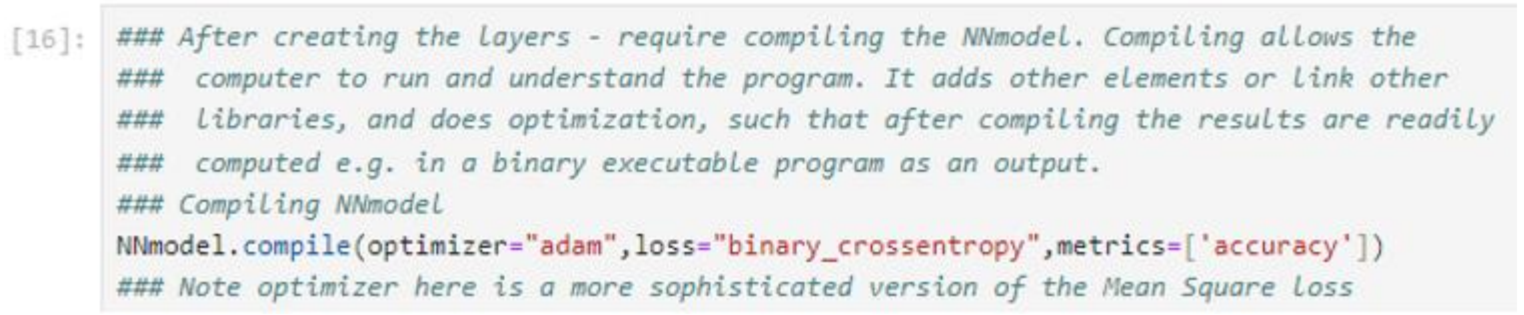
```
[11]: ### This is the very first step while creating NNmodel -- you can rename this model.
      ### Here we are going to create NNmodel object by using a certain class of Keras
      ###  named Sequential. As a part of tensorflow 2.0, Keras is now integrated with
      ###  tensorflow and is now 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
```



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

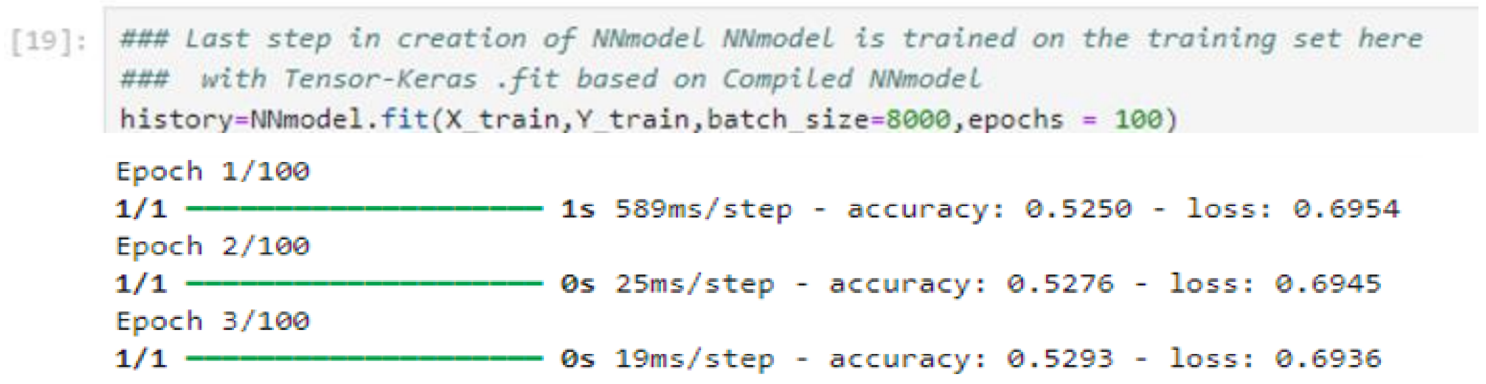
For a binary classification problem as above, actual case output is 1 or 0. Hence we require only one neuron to output layer - output could be estimated probability of case's actual output = 1. In the binary output case, the suitable activation function is the sigmoid function.



```
[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
```

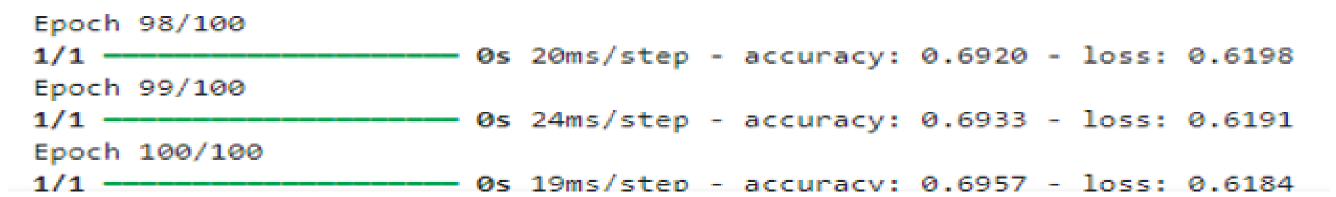
Compile method above accepts inputs -- (1) optimizer: specifies which optimizer to be used in order to perform gradient descent (2) error/loss function, e.g., 'binary_crossentropy' here. For multiclass classification, it should be categorical_crossentropy, (3) metrics is the performance metrics to use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

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)

      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
```

……………………………………………………………………………………………

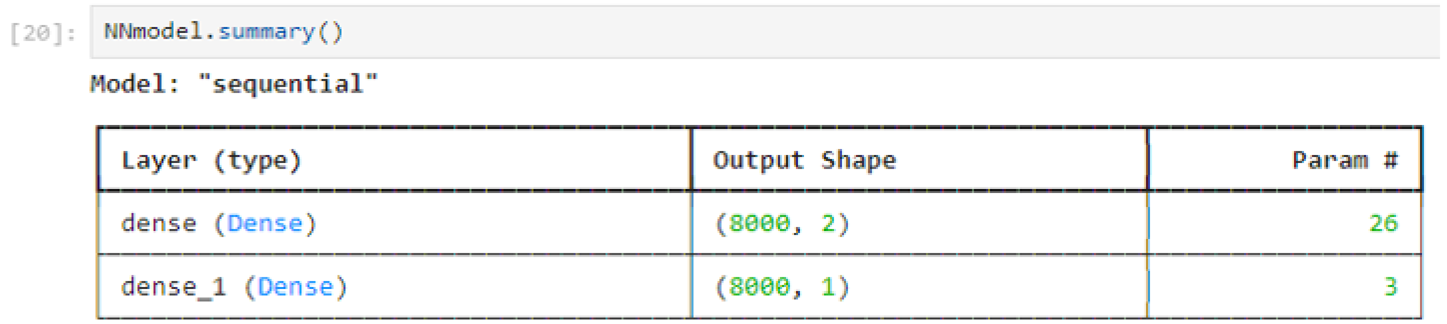


```
Epoch 98/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.6920 - loss: 0.6198
Epoch 99/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.6933 - loss: 0.6191
Epoch 100/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.6957 - loss: 0.6184
```

It is noted that after 100 epochs, the loss is 0.6184 and accuracy (% of correct predictions of exit or retention) is 0.6957 (69.57%).

Please upload Chapter7-1.ipynb and follow the computing steps in Jupyter Notebook

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



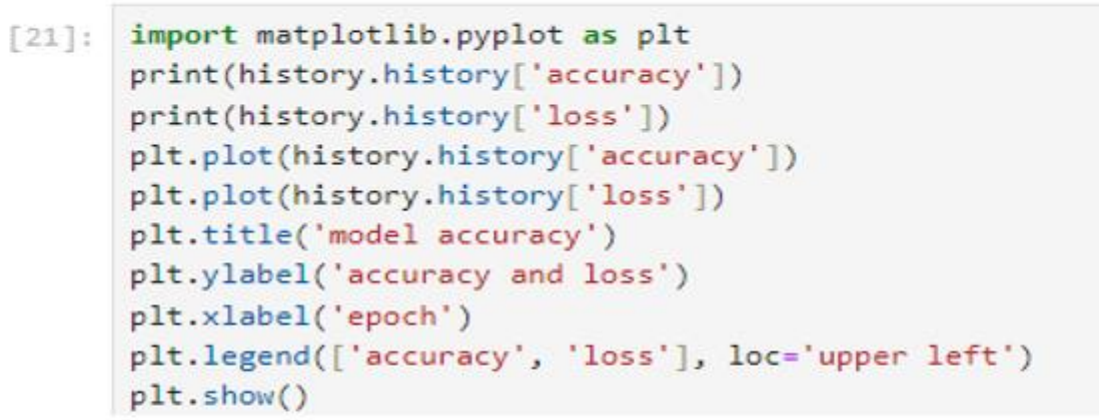
```
[20]: NNmodel.summary()

Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(8000, 2)	26
dense_1 (Dense)	(8000, 1)	3

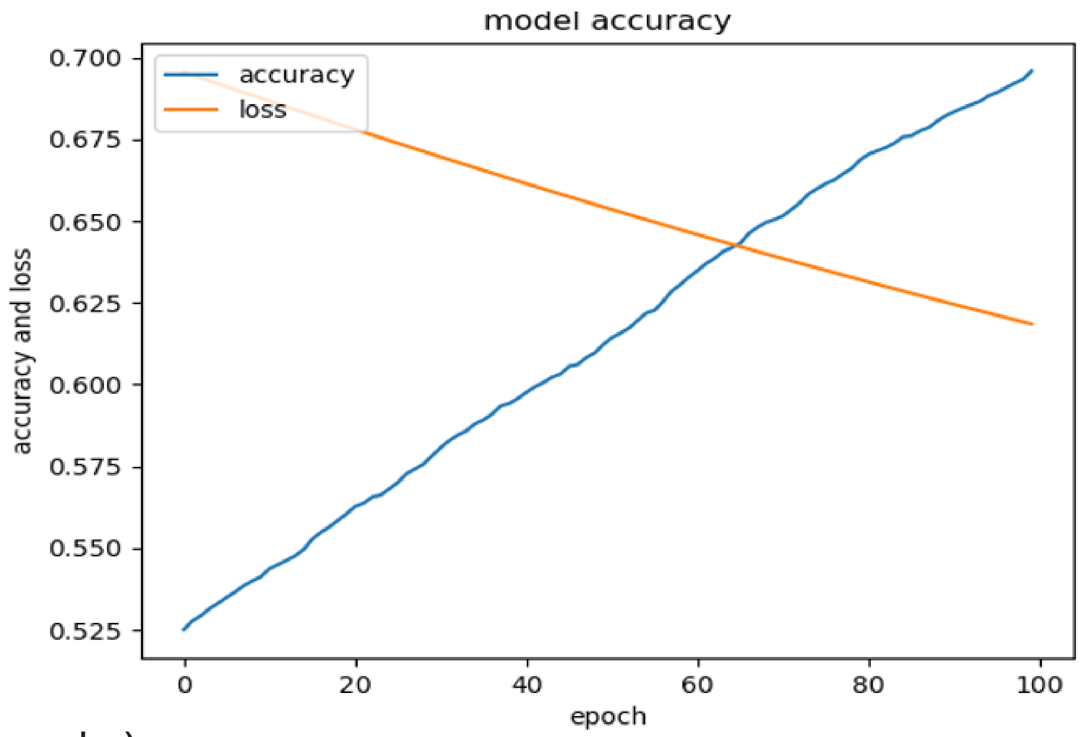
As there are 12 features as inputs, and one bias as input to each of the two neurons in the first hidden layer, there are (12+1) × 2 = 26 weight parameters in the first dense (fully connected) hidden layer. In the 2 outputs of the first hidden layer, adding the next bias, there are 2+1 = 3 weight parameters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

Please upload Chapter7-1.ipynb and follow the computing steps in Jupyter Notebook

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].



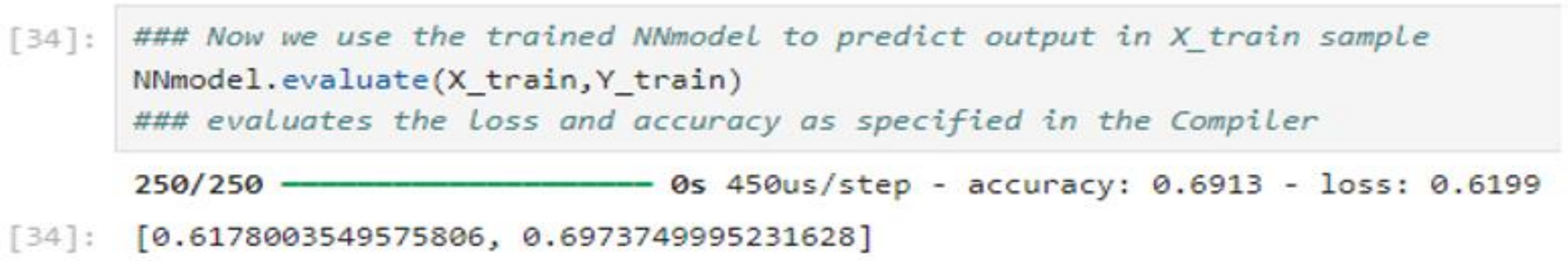
```
[21]: import matplotlib.pyplot as plt
      print(history.history['accuracy'])
      print(history.history['loss'])
      plt.plot(history.history['accuracy'])
      plt.plot(history.history['loss'])
      plt.title('model accuracy')
      plt.ylabel('accuracy and loss')
      plt.xlabel('epoch')
      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%.

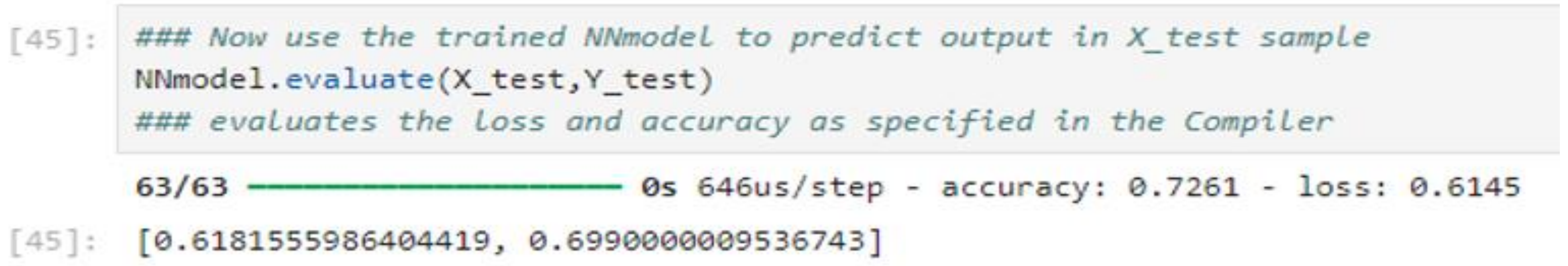
Please upload Chapter7-1.ipynb and follow the computing steps in Jupyter Notebook

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 0.6199 and accuracy of 0.6913.



```
[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

      250/250 ━━━━━━━━━━━━━━━━━━━━ 0s 450us/step - accuracy: 0.6913 - loss: 0.6199
[34]: [0.6178003549575806, 0.6973749995231628]
```

In code line [45], we finally use the trained NN to perform prediction of exits in the test sample (of 2000 points). This is verified in [46], [47] as 69.90%.



```
[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

      63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 646us/step - accuracy: 0.7261 - loss: 0.6145
[45]: [0.6181555986404419, 0.6990000009536743]
```

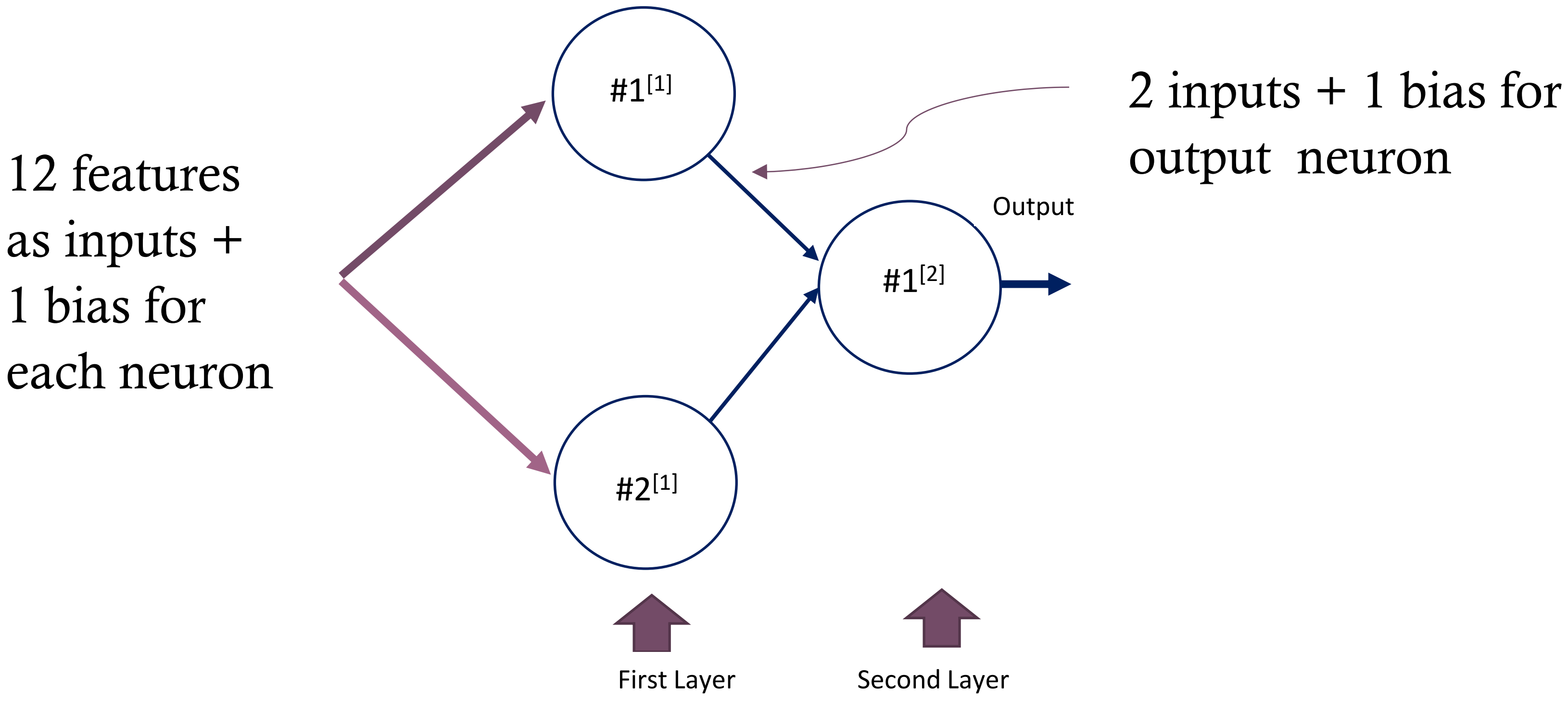
The TE1 in code line [46] of keras provides the vector of predicted probabilities in the context of binary cross entropy loss in predicting the binary class of “1” or “0”.

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.

So far the architecture of ANN is

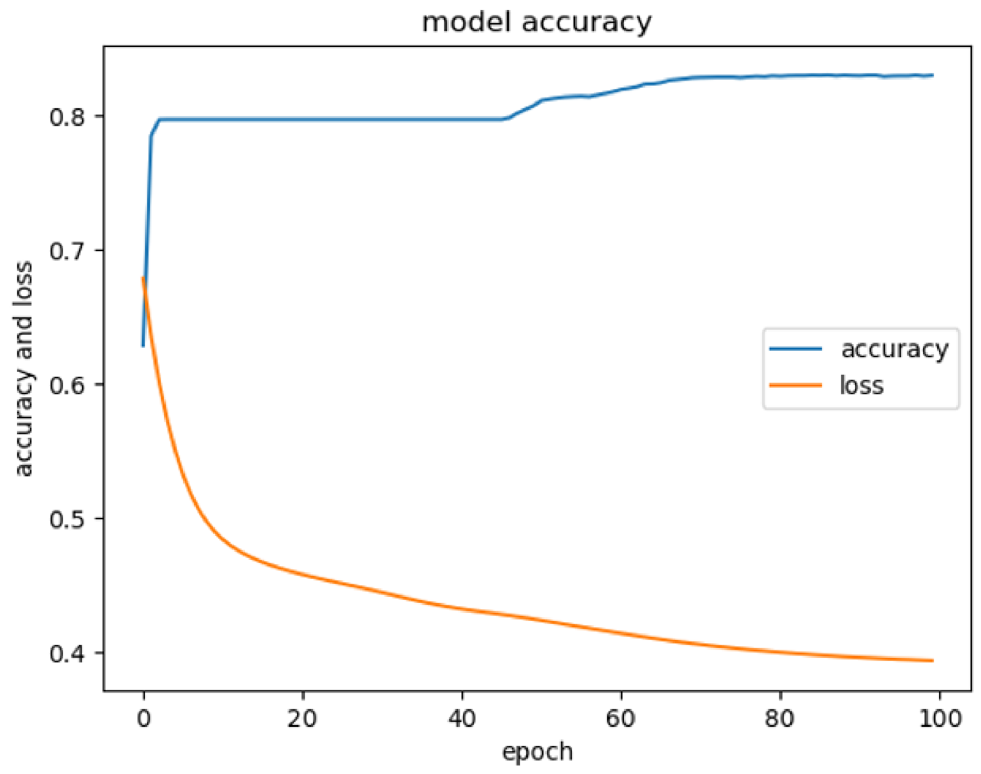
```
NNmodel.add(tf.keras.layers.Dense(units=2,activation="sigmoid"))
NNmodel.add(tf.keras.layers.Dense(units=1,activation="sigmoid"))
NNmodel.compile(optimizer="adam",loss="binary_crossentropy",metrics=['accuracy’])
history=NNmodel.fit(X_train,Y_train, batch_size=8000, epochs = 100) ### 100 iterations
```



12 features as inputs + 1 bias for each neuron
#1[1]
#2[1]
#1[2]
Output
2 inputs + 1 bias for output neuron
First Layer
Second Layer

Improving the NN and Understanding the Results

Sometimes a single batch may not yield robust or stable results. 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. This yields the following loss and accuracy graph. See demonstration file Chapter7-2.ipynb.

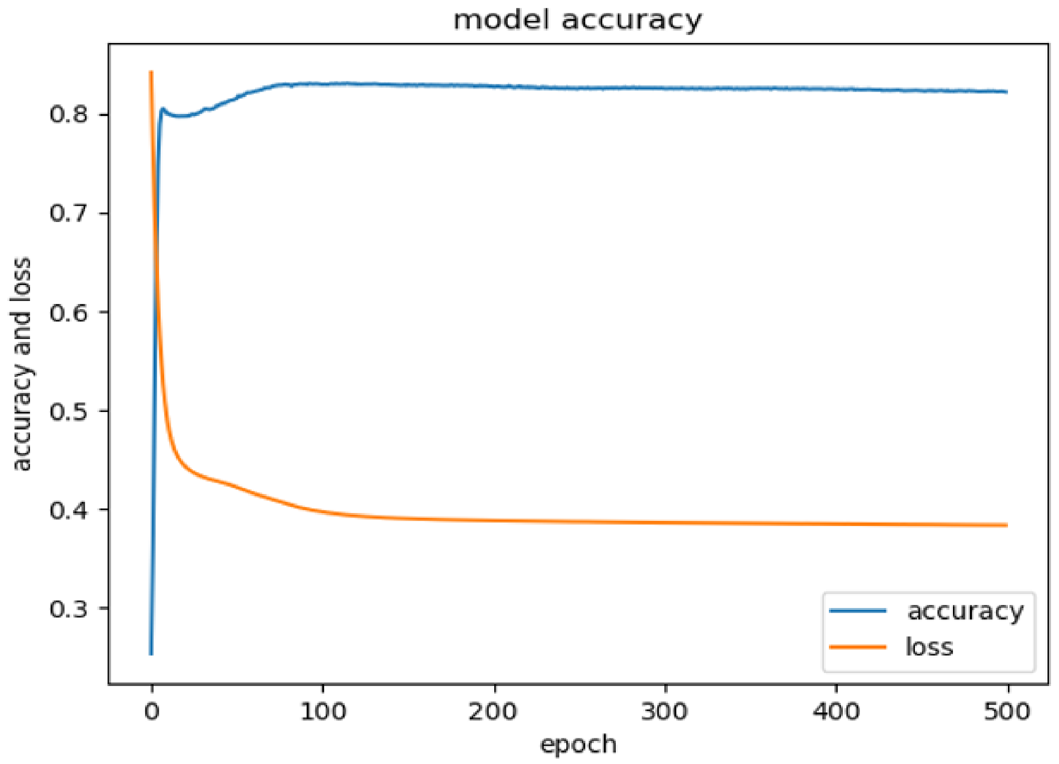


history=NNmodel.fit(X_train,Y_train, batch_size =100, epochs = 100) There are now 80 mini batches per epoch. Total of 80 x 100 = 8,000 iterations

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.

Improving the NN

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. This yields the following loss and accuracy graph. See demonstration file Chapter7-3.ipynb.



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.

Predictions for the additional models/hyperparameters: (1), (2), (3)

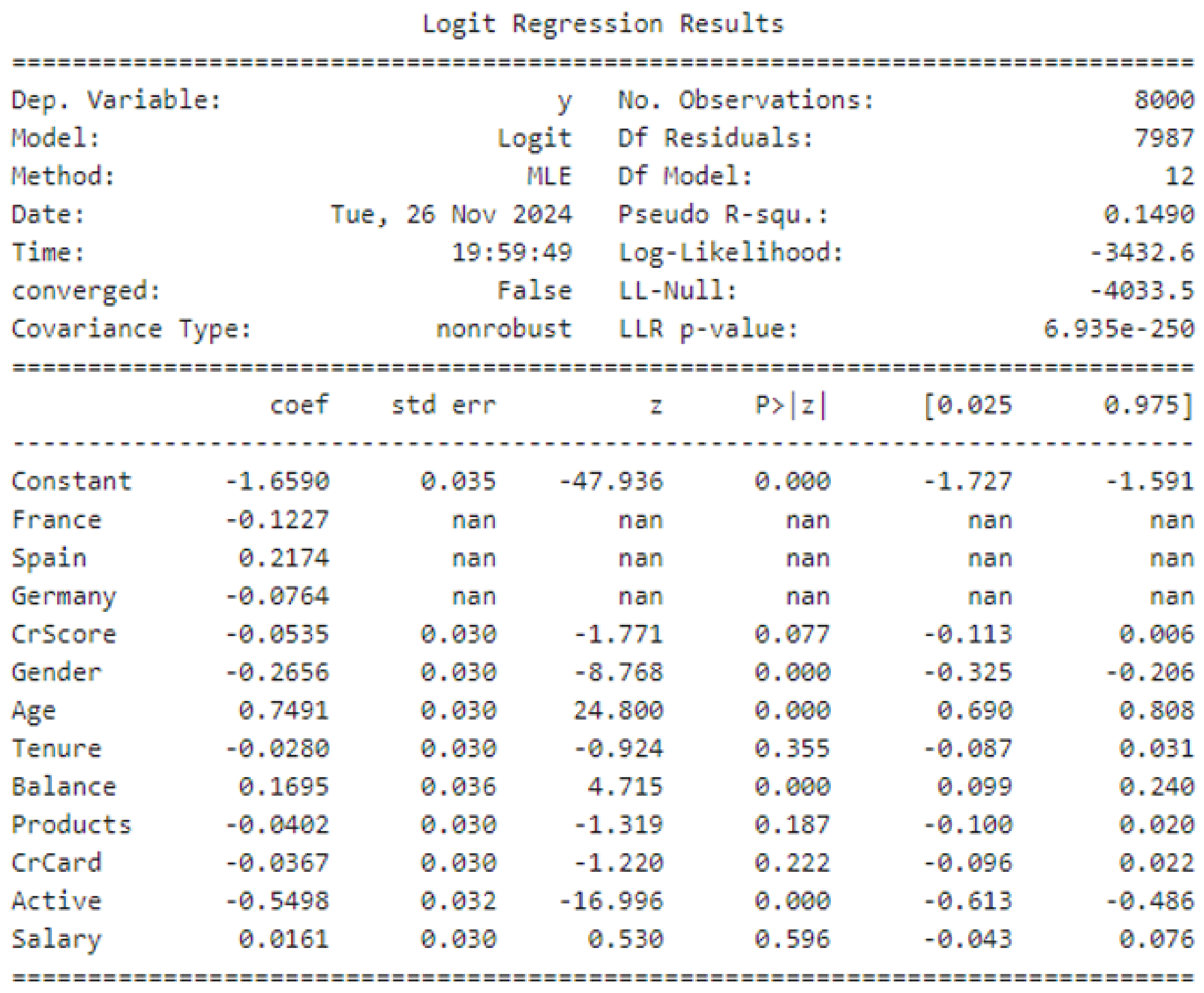
We employ the same batch size = 100 and 500 epochs, but (1) adjust the optimizer to SGD instead of Adam, (2) add a second hidden layer with now 4 neurons in both the first and second hidden layers. We create an even deeper NN in (3) where the number of neurons in the first and the second hidden layers are increased from 4 to 8. See demonstration files Chapter7-4,-5,-6.ipynb.

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%

The prediction accuracy using a benchmark logit regression (where estimated probability > 0.5 is designated as predicting 1, otherwise 0) shows 81.25%.

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] in demonstration file Chapter7-7.ipynb. The estimated coefficients and their statistical significance provide some observations to help understand how the features influence the prediction of ‘exit’ in the NN.

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.



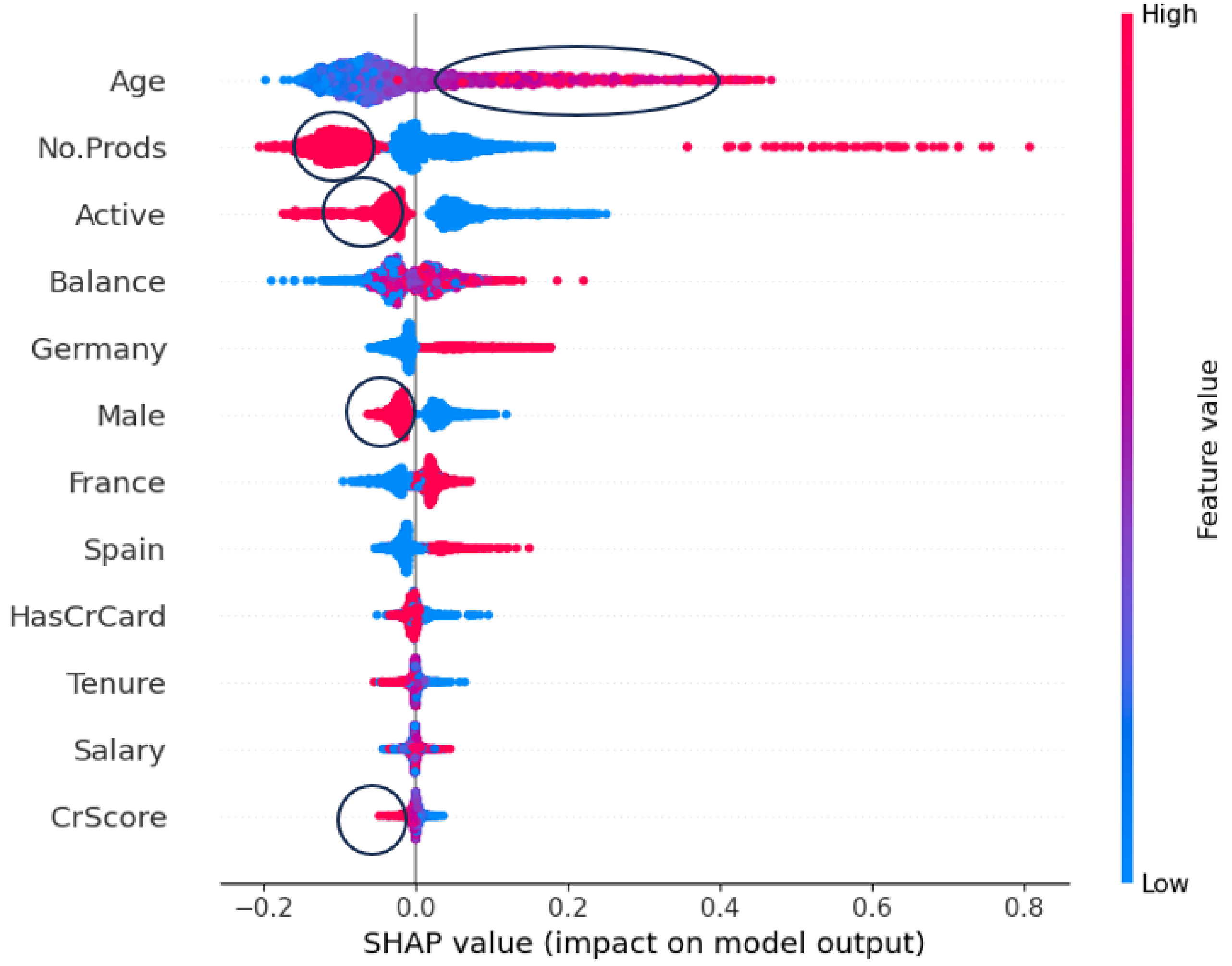
```
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 8000
Model:                          Logit   Df Residuals:                     7987
Method:                           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
==============================================================================
```

Understanding the Results

In Chapter7-8.ipynb, model (3), 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. 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.



Homework 4 Graded Exercise:

Chapter7-9.ipynb (this file is provided only after grading)

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., credit score (higher score means more credit-worthy), geography (nationality), gender, age, tenure (years with the bank), bank balance in the period of the data, the number of bank products used by the customer, whether the customer has the bank credit card (yes = 1, no = 0), is active bank customer (yes =1, no = 0), and estimated USD salary of the customer. The 10 features of each customer are used to fit/predict the target variable "exited" – which is whether the customer exited or left the bank. The classification of "exited" is 1 if the customer left the bank. "exited" = 0 if the customer continues with the bank.

The data follow that used in Chapter7-1 notebook. You should use the following as code line [1] in your solution. It sets a constant seed "5" for the random resampling for each epoch. This will ensure your computed numbers stay the same each time you re-run the program.

Homework 4 Graded Exercise:

In [1]:
```
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)
tf.config.experimental.enable_op_determinism()
```

Next you should use the same code lines [2] to [8] in Chapter7-1.ipynb to feature-engineer your data such as hot-encoding, including preprocessing via standard-scaling. Then split your data into 80% training set and 20% test set with random_state=1.

After this, deviate from the demonstration case in Chapter7-1. Employ tf.keras.models.Sequential() to construct a MultiLayer Perceptron with 3 hidden layers each with 6 neurons. Use sigmoid activation function. Compile using optimizer="adam",loss="binary_crossentropy",metrics=['accuracy']. Use (mini) batch_size = 100 and number of epochs = 500 to train/fit the MLP.

Homework 4 Graded Exercise:

Q1. What is the total number of parameters in this constructed MLP?

Q2. What is the training accuracy after 500 epochs?

Q3. What is the test accuracy (prediction accuracy on test sample) using the trained MLP?

Q4. What is the number of iterations before the training accuracy climbed above 80%?

Q5. A generalized data (not in the data set) of a customer is a Spanish gentleman, 60 years old, with credit score 650, tenure 2 years with the bank, a balance of USD 300,000, estimated salary of USD 80,000, subscribing to two products, has a credit card, is assessed not to be an active member/customer. For ease of interpretation, assume the exit prediction is for a year forward. Does this MLP model predict if this customer will exit within the year?

End of Class