COURSEWORK(COMPUTATION INTELLIGENCE)

Msc Data science and AI

Usman Shoukat
201537600

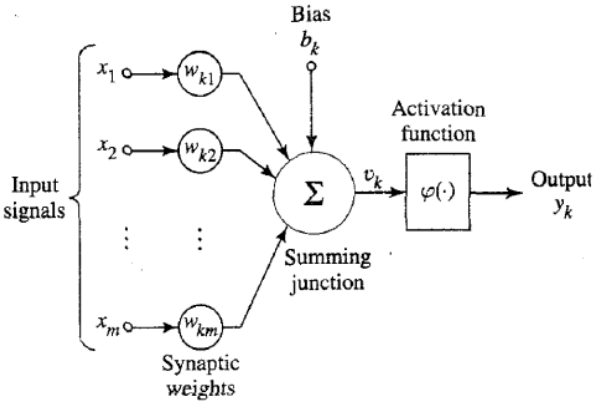
Q1. Use the basic equations for its training from our notes, to implement a simple Perceptron for a given classification problem. The number of inputs and the classi_cation problem to solve will be specifiable by the user. Implement this from scratch using, for example, for-loops to iterate around epochs and patterns as in the lecture notes. Try to make the code as simple as possible by using arrays/matrices/vectors. The training should carry on until some maximum number of epochs is reached or until all patterns are learned. Include the design for this stage (that is basic equations and structure of the algorithm), the code, and also some experiments for 2-3 low- or high-dimensional example problems of your choice (you can download a dataset from the internet, or de_ne one directly, or generate one easily yourselves). Display the weight adaptations during the epochs to show the learning process. Extra marks if your program displays the boundary (for 2D or 3D only!).

Ans:

Perceptron Algorithm:

Perceptron algorithm takes the motivation from the human nervous system. This algorithm helps in binary classification.

The model is trained by the external stimuli in the form of training data, and learning is done by changing the synaptic connections(weights) between the neuron and input data. The number of connections depends upon the number of features of input data. Then the neuron is trained to fire on some specific threshold which means that the neuron fires if the activation score gained through the linear equation (a = $\mathbf{W^TX}$ + b) is greater than that threshold θ, otherwise it does not fire. It is convenient to make the threshold equal to zero; for that purpose, we use bias and make it equal to - θ, which adjusts the threshold equal to zero.



The algorithm works in two phases;

Training mode: In this mode, we train our model by adjusting the parameters according to the training data.

Testing mode: In this mode, we test our model on some unseen data and quantifies its performance on the unseen data.

Below is the pseudocode for both phases:

Training Phase:

1. Training(dataset, iterations, η):
2. **w** = np.zeros(X.shape[1]) (initialising the weights)
3. b = 0 (initialising the bias)
4. for _ in range(iterations):
5. for **x**, d ε (dataset):
6. $y = signum($ $\mathbf{w^Tx}$ + b)
7. $\boldsymbol{w} = \boldsymbol{w} + \eta * (d - y) * \boldsymbol{x}$
8. $b = b + \eta * (d - y)$
9. return **w**, b

From the above pseudocode, you can see we pass on the dataset, learning rate and the number of iterations for which we want to train our model to the training function.
First, we initialise the weights vector **W** with size depending upon the number of dimensions of input data, either with zero or some random values. We also initialise the bias with either zero or some random value.
Then we run a loop for the number of iterations with which we want to train our Perceptron. We can put a condition to stop earlier if the model converged before the given number of iterations.
Then we have two options to send our data to the model to train it: online algorithm and batch training. In the online algorithm, we send our data points one by one to train our model, but for the batch training model, we send our data in batches. After sending the data, we get its activation score using the linear equation $a = \mathbf{W}^T\mathbf{X} + b$. And as we have set a threshold function for the neuron to fire, if it receives a score greater than this threshold, the neuron fires and returns 1, and if the activation score is less than the threshold, then the neuron does not fire and returns -1.
After getting this activation score, we compare it with the given labels, if the predictions are correct, then we do not do anything, and if the predictions are wrong, then we adjust(update) our weights accordingly depending upon which side was predicted wrong.
Once we have gone through all the data for enough iterations or if the model is converged, then we stop training and returns the weights and bias, which are actually our trained parameters.

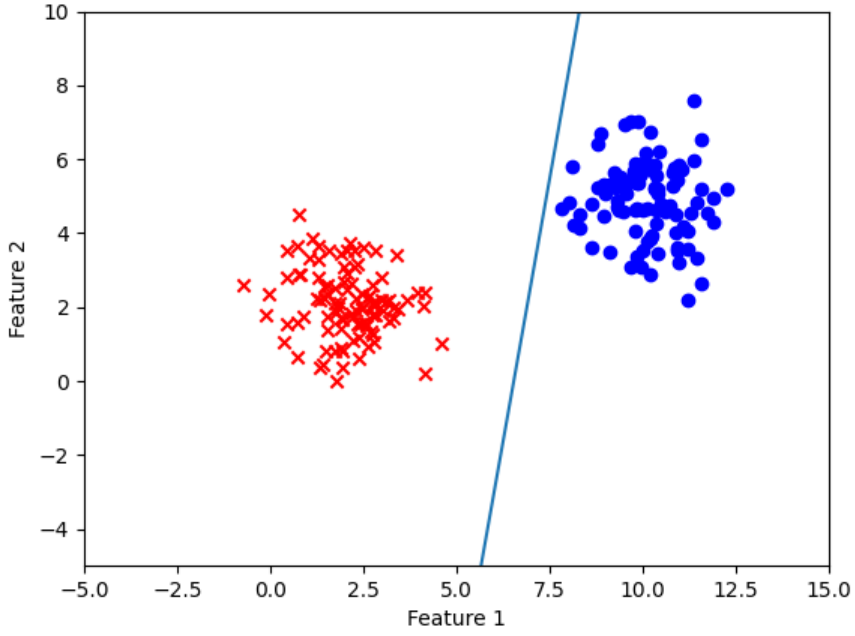
Testing Phase:

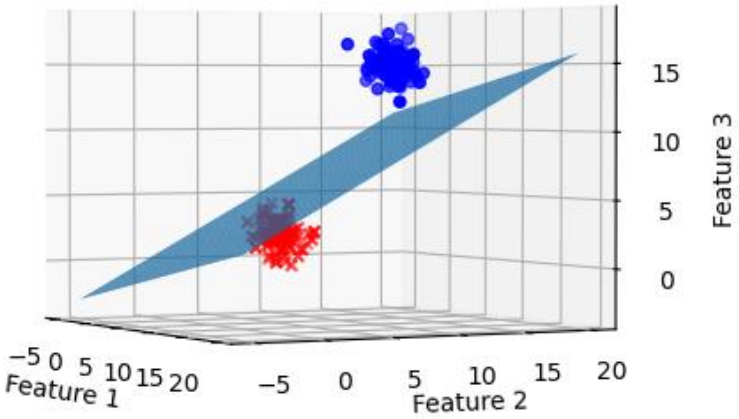
```
1. Testing(testingDataPoint, W, b):
2.     a = WTX + b
3.     If a >= 0:
4.         return 1
5.     else:
6.         return -1
```

Once done with training then we start testing our model. We send trained parameters (**W**, b) and the data which we want to test in the testing function. For the given test data, we compute its score using the linear equation ($a = \mathbf{W}^T\mathbf{X} + b$) and check it against threshold 0. If the score is greater than zero, then we declare it as a positive class. Otherwise, we declare it as a negative class. We also take account of how many correct predictions have been given by the model to evaluate the accuracy of our model.

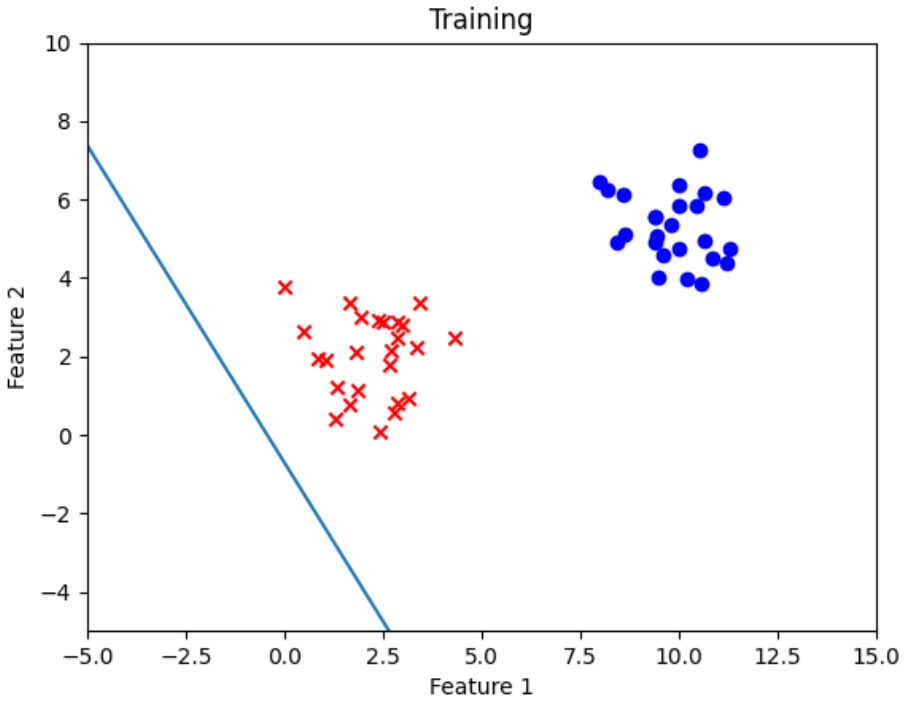
Experiments: In my experiments, I have trained the Perceptron on three different datasets, I created 1 2D and 1 3D dataset myself and the third one was Iris dataset in which I was classifying between setosa and versicular.

I have plotted and visualised the boundries for 2D and 3D trained models which is as follows:

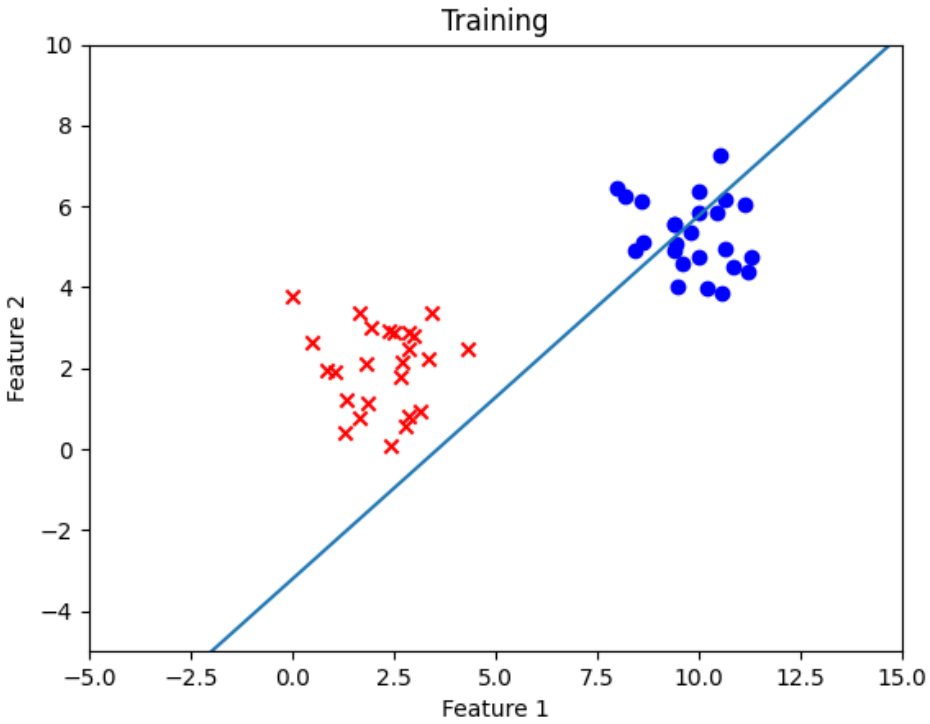




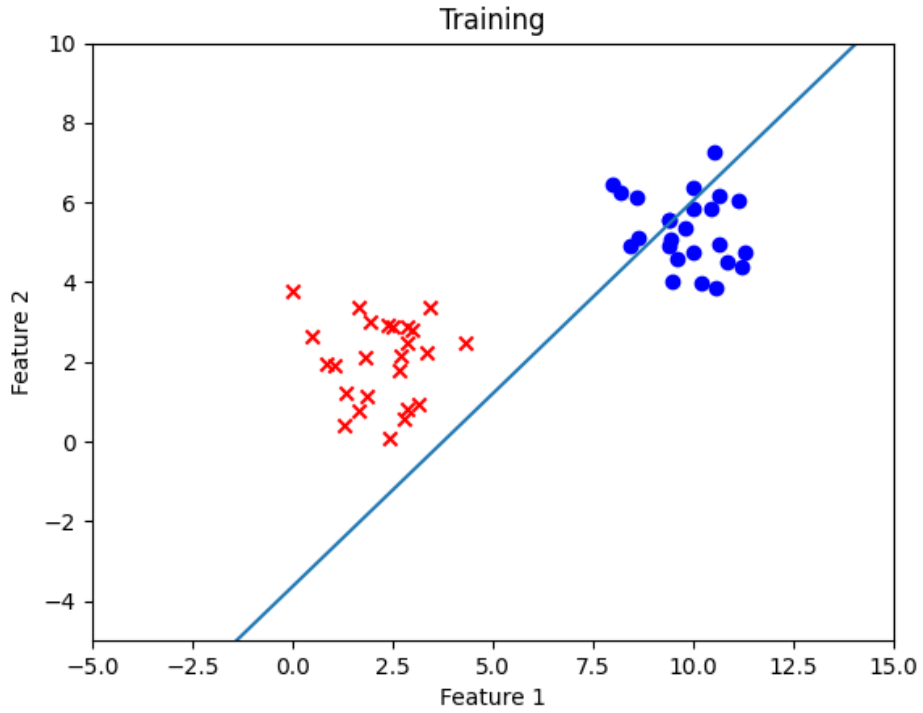
I also visualised the weights adaptation during the training, I am only attaching the results for 2D

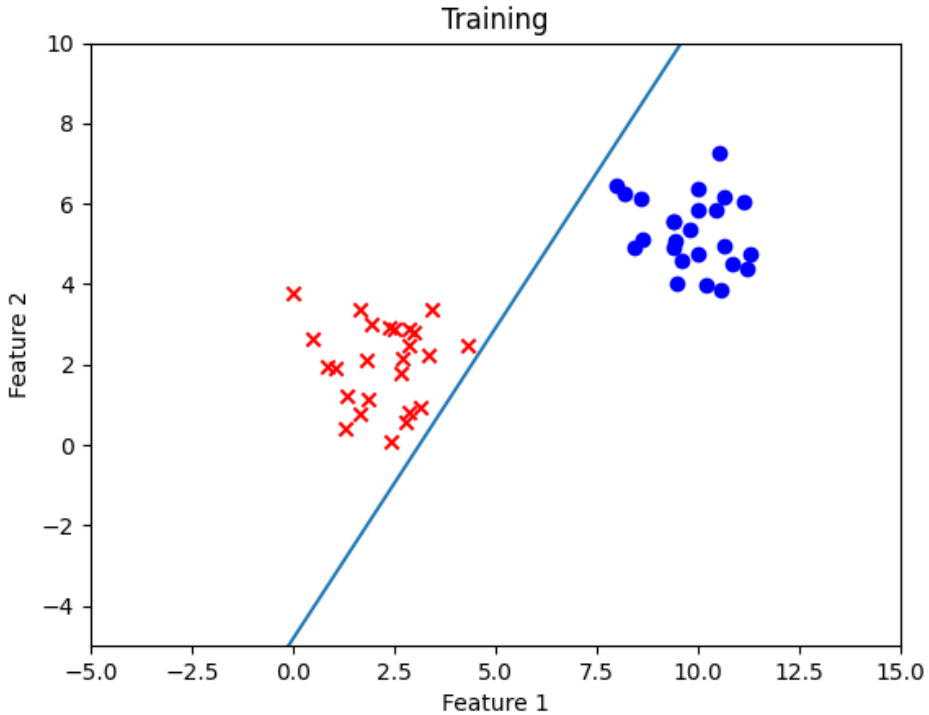


After Epoch 1



After Epoch 2





After Epoch 4

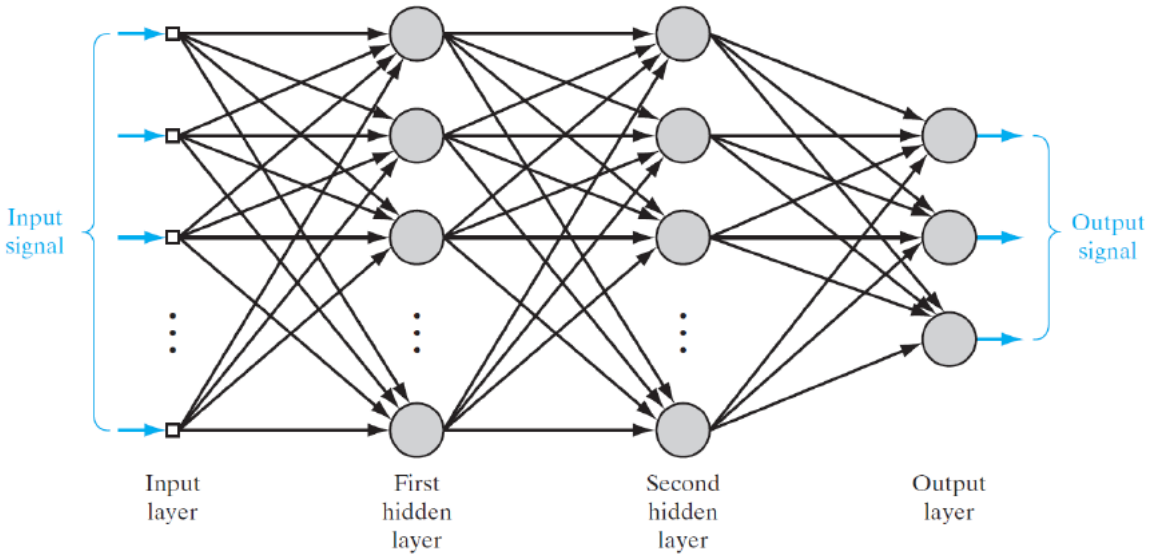
The model got fully trained after 4 Epochs.

Q2 Use the basic equations of Backpropagation from our notes to implement and simulate a simple MLP network. The architecture (number of inputs, number of layers and number of nodes per layer, and number of outputs) can be given by the user or be fixed inside your program. Implement both the forward and the backward passes separately, and use a simple activation function (e.g., logistic, relu, etc.). Include the design for this stage (that is basic equations, structure and sequencing of operations), the code, and also some experiments for 2-3 simple problems of your choice (e.g., toy problems (XOR, symmetry checking), or using random points from a multi-dimensional function you randomly create). Discuss the learning ability in terms of reducing the fitting error in your patterns and experiment with di_erent architectures (you can ignore generalisation issues and just focus on weight adaptation only). Extra marks will be given for momentum incorporation, or experimentation with different activation functions, or use complexity control.

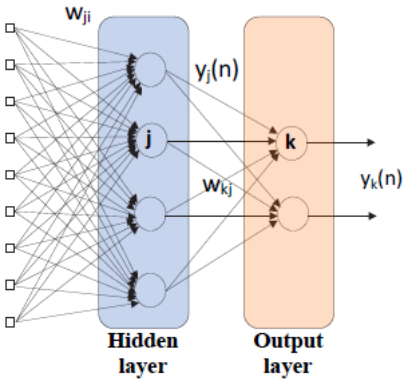
Ans:

Neural Networks:

As Perceptron can only do the linear binary classification, so we need a complex network to do the classification where data in non-linear and there are more than two classes to separate. This complex network includes one or more hidden layers and neurons between input and output layer. These neurons and hidden layers are hyperparameters and are set according to the complexity of data. The number of output neurons depends upon the number the classes in which we want to classify our data. A sample neural network with two hidden layers is shown in the figure below.



The training of neural network is done in two phases. In first phase we give the input signal too the neural network, take its prediction through forward and compute the error between the predicted value and actual value. And in the second phase we try to adapt the weight according to the error through backward propagation. I used the model with only one hidden layer which seems like this:



Forward Path: In the forward path, at each layer we take the dot product of weights and inputs, add the bias terms of that layer to corresponding neuron output and then pass it through the non linear activation function. The general equation seems like this:

$$\boldsymbol{v_i = W^T x + b}$$

Where $\boldsymbol{v_i}$ is the vector of length equal to number of neuron in the layer. This vector contains the induced local fields of the neurons of the layer which have been given the signal from the previous layer. $\boldsymbol{W^T}$ is the weight and is the synaptics connection between the two layers. It is in the form of a matrix and its order s equal to number of neurons in the layers from we passing the signal times number of neurons in the layer which is receiving the signal. And $\boldsymbol{b}$ is a vector of terms of all the neurons in the corresponding layer.
After calculating the induced local fields we pass it to non-linear activation function.

$$\boldsymbol{y_i = \phi(v_i)}$$

There are many activation functions that are being used e.g., logistics activation function, relu activation function and sigmoid activation function. I used the sigmoid function, and its equation is as follows:

$$\boldsymbol{y_i} = \frac{1}{1 + e^{-v_i}}$$

Forward path takes the input signal and works like this:

- forward path:
- $\boldsymbol{v_j = W_{ji} x_i + b_j}$
- $\boldsymbol{y_j} = \frac{1}{1+ e^{-v_j}}$
- $\boldsymbol{v_k = W_{kj} y_j + b_j}$
- $\boldsymbol{y_k} = \frac{1}{1+ e^{-v_k}}$
- $return\ \boldsymbol{v_k, y_k}$

Backword Propagation: At the end of the forward path we compute the error between the desired values and output signal. Different error functions could be used for this purpose e.g., mean squared error funcation and cross entropy error function. I used the mean squared error in my model. For thisis the error of the neuron is

$$e = d - y$$

Where d is the desired value and y is the predicted value at the particular neuron and e is the error between these values. And the error of the network is given by

$$E = \frac{1}{2} \sum_{i\ \varepsilon\ output} (e_i)^2$$

So, this error is sum of squares of error of each neuron in the output layer.
After having the error value at each output neuron we backpropagate this error and try to adapt our weight accordingly. We use the chain rule to do this back propagation. For one hidden layer neural network, the change in weights between hidden layer and output layer are given by:

$$\frac{\partial E(n)}{\partial w_{kj}(n)} = \frac{\partial E(n)}{\partial e_k(n)} \frac{\partial e_k(n)}{\partial y_k(n)} \frac{\partial y_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial w_{kj}} \qquad (1)$$

Where

$$\frac{\partial E(n)}{\partial e_k(n)} = e_k(n)$$
$$\frac{\partial e_k(n)}{\partial y_k(n)} = -1$$
$$\frac{\partial y_k(n)}{\partial v_k(n)} = \phi'(\boldsymbol{v_k})$$

And during these derivations i subscript shows the input layer and j and k represents hidden and output layer respectively.
As I have used the sigmoid function, so its derivative after simplification is equal to

$$\frac{d}{dv}\frac{1}{1+e^{-v}} = \left(\frac{1}{1+e^{-v}}\right) * \left(1 - \frac{1}{1+e^{-v}}\right)$$

And

$$\frac{\partial v_k(n)}{\partial w_{kj}} = y_j(n)$$

So, the equation 1 can be written as:

$$\frac{\partial E(n)}{\partial w_{kj}(n)} = -e_k(n)\left(\frac{1}{1+e^{-vk(n)}}\right)\left(1 - \frac{1}{1+e^{-vk(n)}}\right)y_j(n)$$

For simplicity, we write

$$-e_k(n)\left(\frac{1}{1+e^{-vk(n)}}\right)\left(1 - \frac{1}{1+e^{-vk(n)}}\right) = -\delta_k(n)$$

Here $\delta_k(n)$ is the local gradient of kth neuron of output layer and is responisble for giving the direction which minimises our error.

So, the overall partial derivative of error with respect to the weights between hidden and output neurons can be written as:

$$\frac{\partial E(n)}{\partial w_{kj}(n)} = -\delta_k(n)y_k(n)$$

The change in weights between output and hidden layer can be written as:

$$\triangle w_{kj}(n) = -\eta\frac{\partial E(n)}{\partial w_{kj}(n)}$$
$$\triangle w_{kj}(n) = -\eta\left(-\delta_k(n)y_j(n)\right) = \eta\delta_k(n)y_j(n)$$

Now we do not have direct error at the hidden layer, so we will need to back propagate that error from the output layer. By using the chain rule the partial derivate of error function with respect to weights of between input and hidden layer can be written as follows:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}} \qquad (2)$$

The local gradients of the hidden layer will be equal to:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)}\left(\frac{1}{1+e^{-vj(n)}}\right)\left(1-\frac{1}{1+e^{-vj(n)}}\right)$$

Now for $\frac{\partial E(n)}{\partial y_j(n)}$ we know that the error function is equal to $E = \frac{1}{2}\sum_{k\ \varepsilon\ output}(e_k)^2$
After taking the partial derivative of error function with respect to output of hidden jth neuron, and simplyfying it we get,

$$\frac{\partial E(n)}{\partial y_j(n)} = -\sum_{k\ \varepsilon\ output} e_k(n)\left(\frac{1}{1+e^{-vk(n)}}\right)\left(1-\frac{1}{1+e^{-vk(n)}}\right)w_{kj}(n)$$

Where $e_k(n)\left(\frac{1}{1+e^{-vk(n)}}\right)\left(1-\frac{1}{1+e^{-vk(n)}}\right) = \delta_k(n)$ is the local gradient of the output layer.
Now the local gradient of hidden layer can be simplified as:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} = \left(\frac{1}{1+e^{-vj(n)}}\right)\left(1-\frac{1}{1+e^{-vj(n)}}\right)\sum_{k\ \varepsilon\ output}\delta_k(n)\,w_{kj}(n)$$

Putting this in (2)

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \left(\frac{1}{1+e^{-vj(n)}}\right)\left(1-\frac{1}{1+e^{-vj(n)}}\right)\sum_{k\ \varepsilon\ output}\delta_k(n)\,w_{kj}(n)$$

And

$$\triangle w_{ji}(n) = -\eta\frac{\partial E(n)}{\partial w_{ji}(n)}$$
$$\triangle w_{ji}(n) = -\eta\left(-\delta_j(n)x_i(n)\right) = \eta\delta_j(n)x_i(n)$$

From these derivation the equation I used in sequence are:

- Backward path:
- For each epoch:
- For each data point:
- $v_{k,}y_k = forward(dataoint)$
- $e_k = d_k - y_k$
- $\delta_k(n) = e_k(n)\left(\frac{1}{1+e^{-vk(n)}}\right)\left(1-\frac{1}{1+e^{-vk(n)}}\right)$
- $\triangle w_{kj}(n) = \eta\delta_k(n)y_j(n)$
- $w_{kj}(n) = w_{kj}(n) + \eta\delta_k(n)y_j(n)$
- $b_k(n) = b_k(n) + \eta\delta_k(n)$
- $\delta_j(n) = \left(\frac{1}{1+e^{-vj(n)}}\right)\left(1-\frac{1}{1+e^{-vj(n)}}\right)\sum_{k\ \varepsilon\ output}\delta_k(n)\,w_{kj}(n)$
- $w_{ji}(n) = w_{ji}(n) + \eta\delta_j(n)x_i(n)$
- $b_j(n) = b_j(n) + \eta\delta_j(n)$

Here i represents input layer, j represents hidden layer and k represents output layer.

Experiments and Analysis:

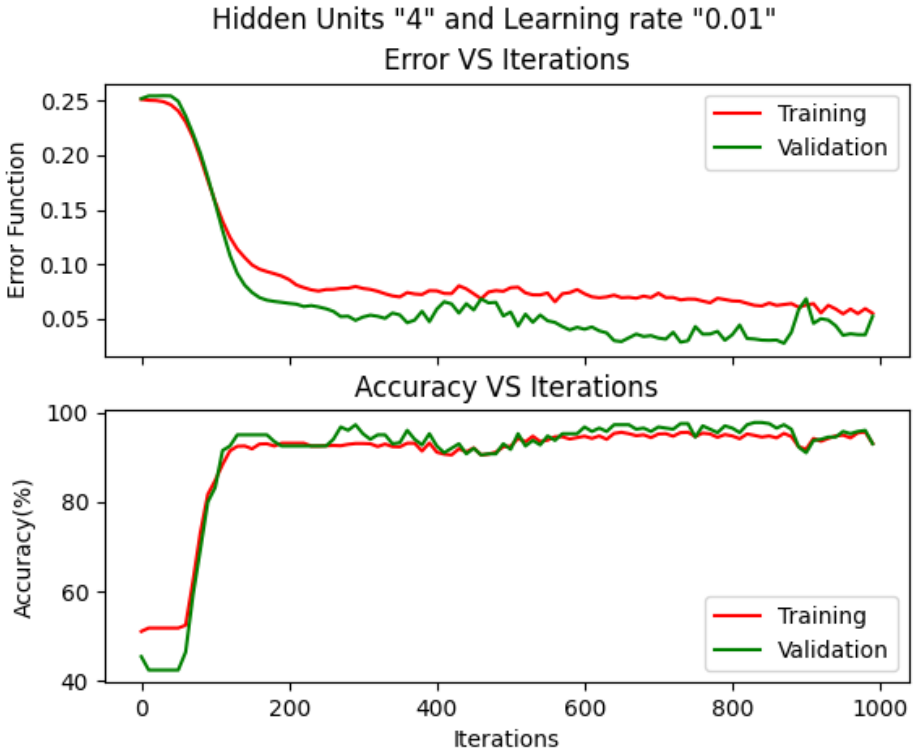
Now I had different choices to choose between different hyperparameters like learning rate, number of epochs and number of hidden units.

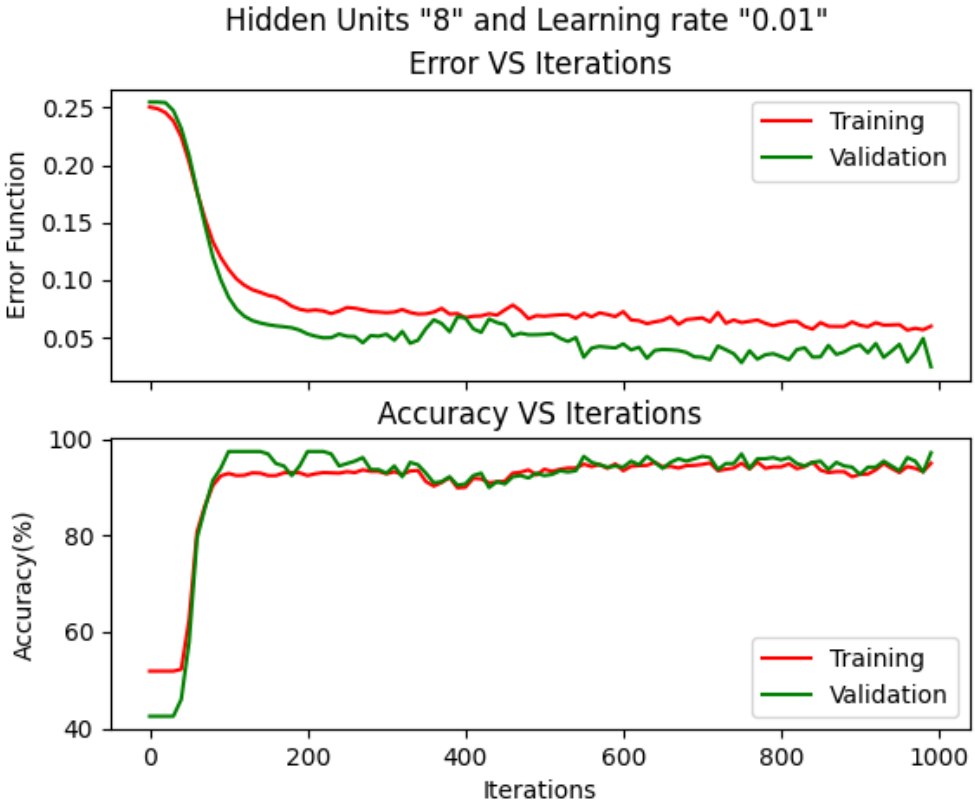
In general, If we are training our network with less number of neurons or hidden layers than that is needed to train the model considering the complexity of data, then the model would be underfitted and the hyperparameters needs to be increased. If the neurons and hidden layers are more than needed to satisfy the complexity of data then the model would be overfitted and would be behaving well on training data and would not be generalising well. So, these units will need to be decreased. The underfitting and overfitting can be seen by plotting the loss vs epoch curves for different neurons and hidden layers and the model which is giving lower average error and higher accuracy on both training and validation data can be taken as finalised network model. These loss vs epochs and accuracy vs epoch curves also helps in deciding the number of epochs and learning rate.

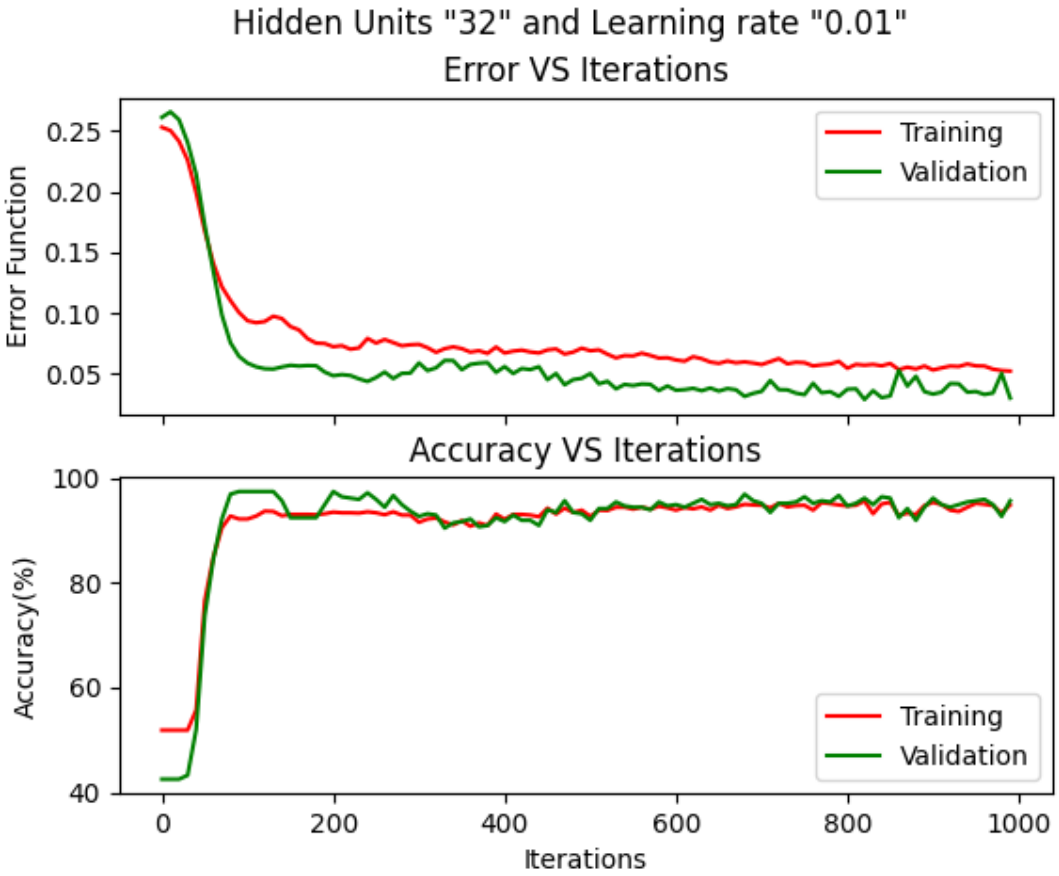
I used Crab dataset and Iris Data set to train my model. In crab dataset there are two classes male and female in which we need to classify and in Iris dataset there are three types of flowers (setosa, versicular, verginica) in which we need to classify.

First for Crab dataset, I divided my dataset into training and validation dataset. Then I took $\eta = 0.01$ and number of epochs 1000 and trained my model with different number of hidden units. I plotted their accuracy vs iteration and loss vs iteration graphs. Some of them I am attaching them here.

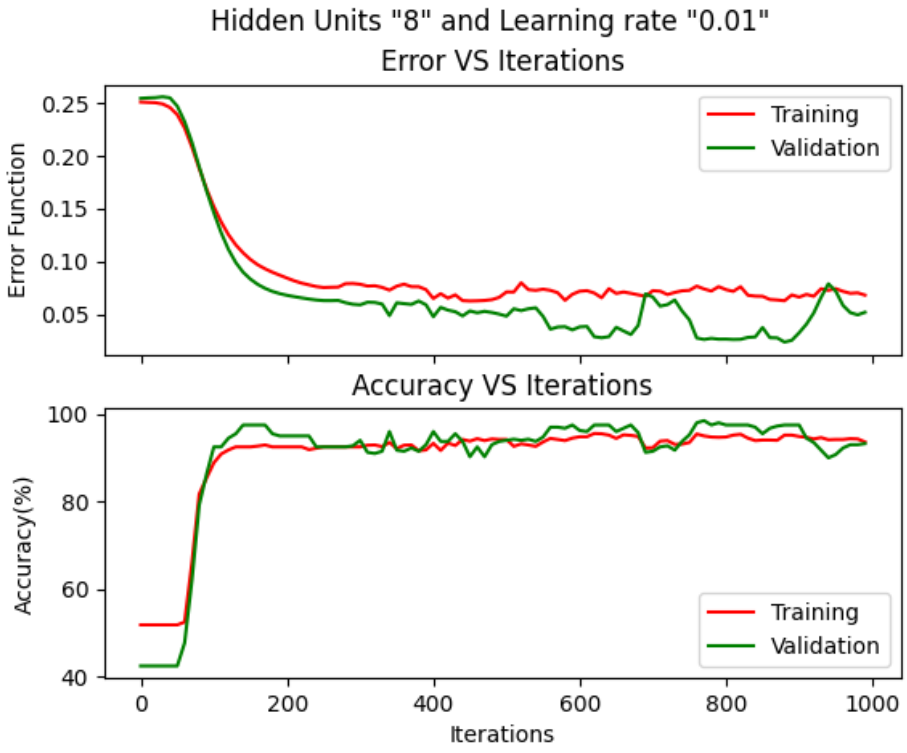
With 2 hidden units the model was underfitting and was not behaving well both on training as well as validation data. Then I increased the number of hidden units.



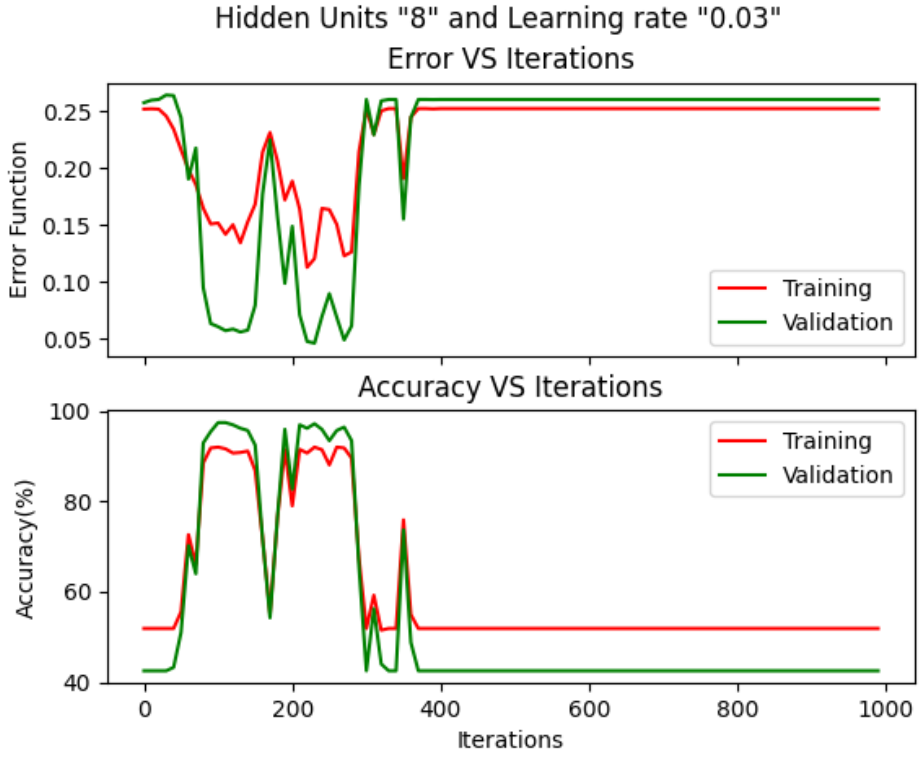




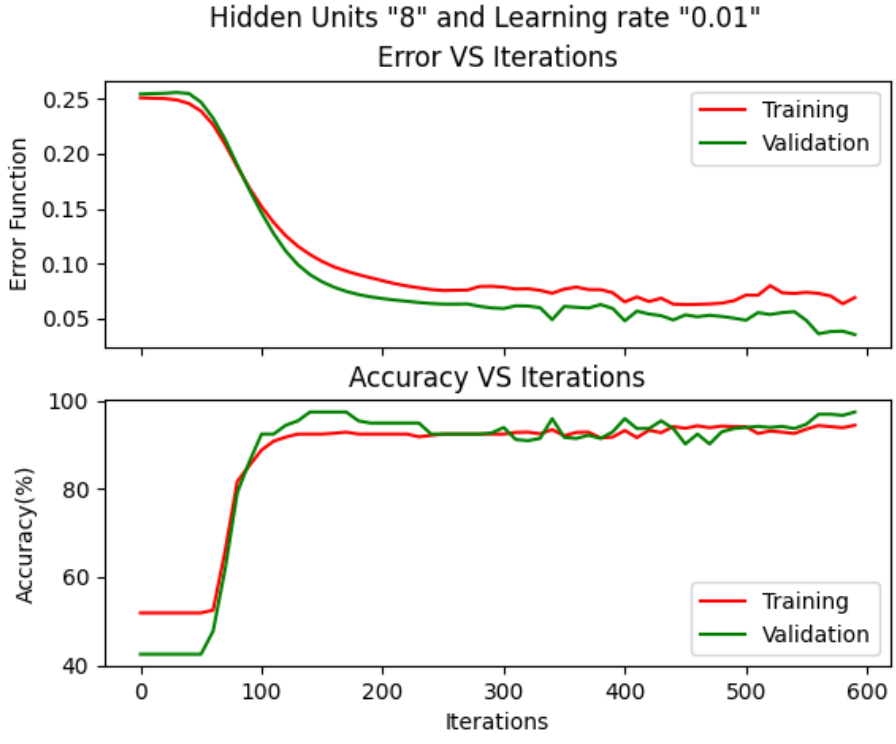
As the model was behaving almost similar for the increased number of hidden units, so I picked 8 hidden units for my model.



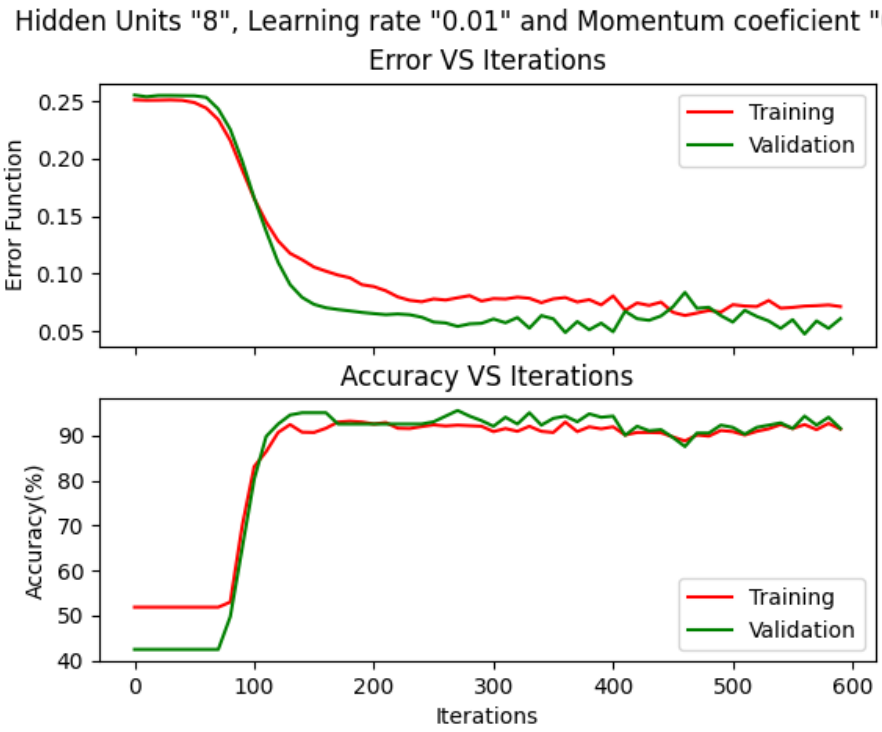
Now as Learning rate is another hyperparameter which can be decided by plotting loss vs epoch and accuray vs epoch curves with different values of learning rate so I plotted the graphs with 8 hidden units and 1000 epochs with different learning rates. I am attaching some sample graphs here

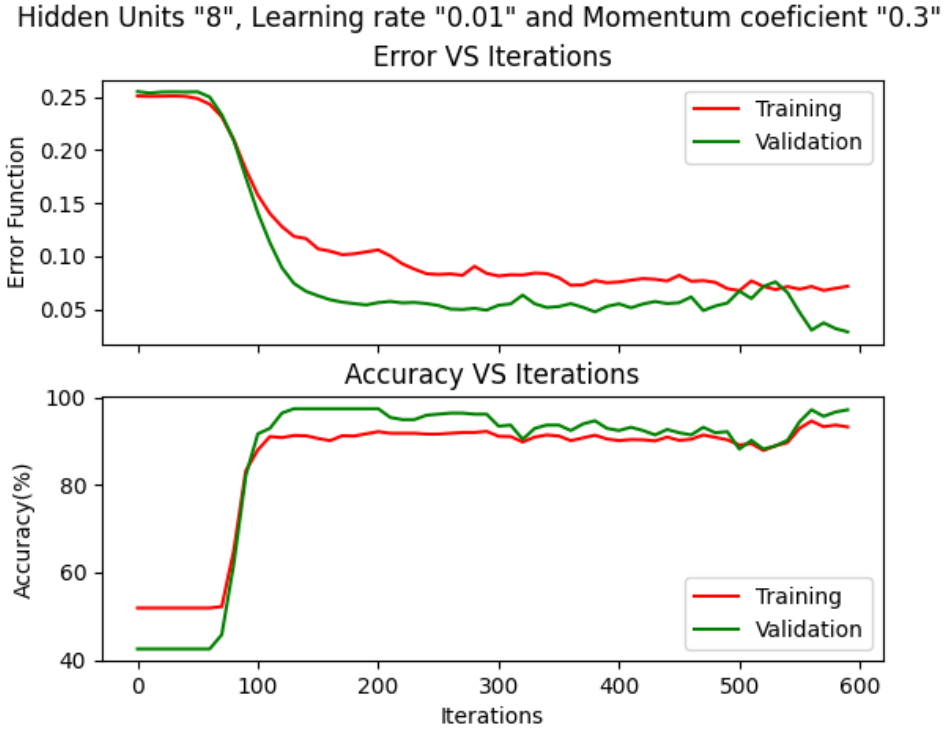


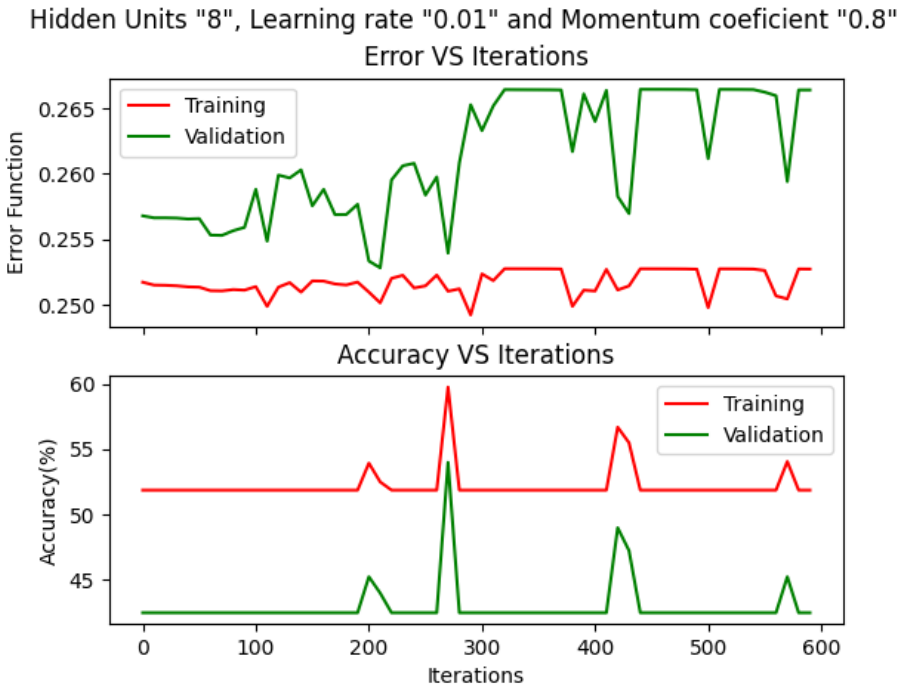
The model was behaving well learning rate 0.01,but it started under fitting as I increased the values beyond 0.01, so I kept it be 0.01. We can also see that the model is almost converged after 500-600 epochs so we can take 600 as the number of epoch for which we need to train our model.
After training the model on these hyperparameter η= 0.01, epochs= 600 and hidden units = 8, the loss vs epoch and accuracy vs epoch curve seems like this:



We can see that the model is converging even before 600 epochs, but as the model is not overfitting so it is safer to train it for enough number od epochs.
I also did some experiments by including the momentum coefficient. It did not give any improvement for crab dataset, so some sample graphs are as follows:

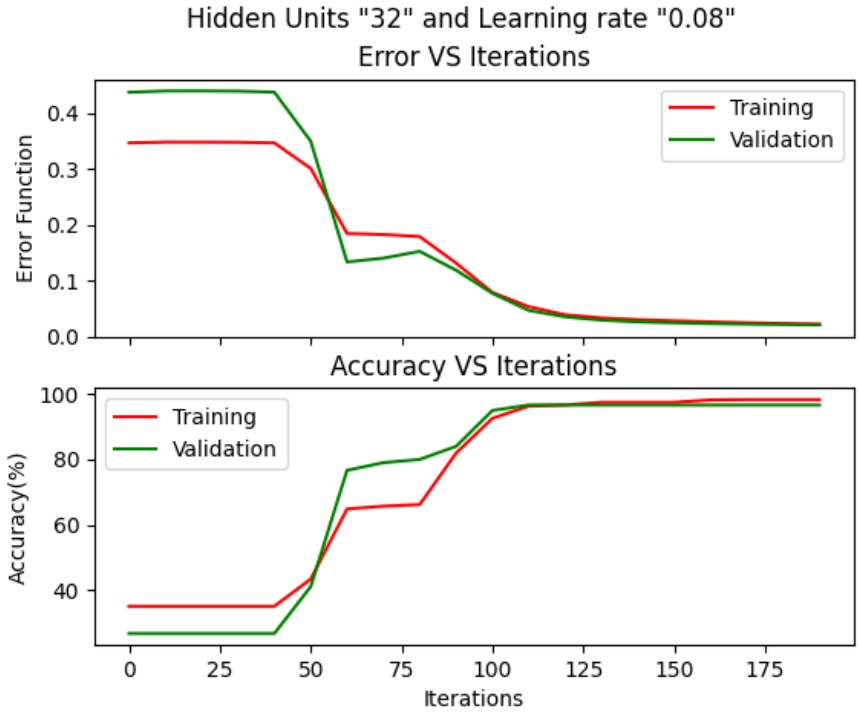






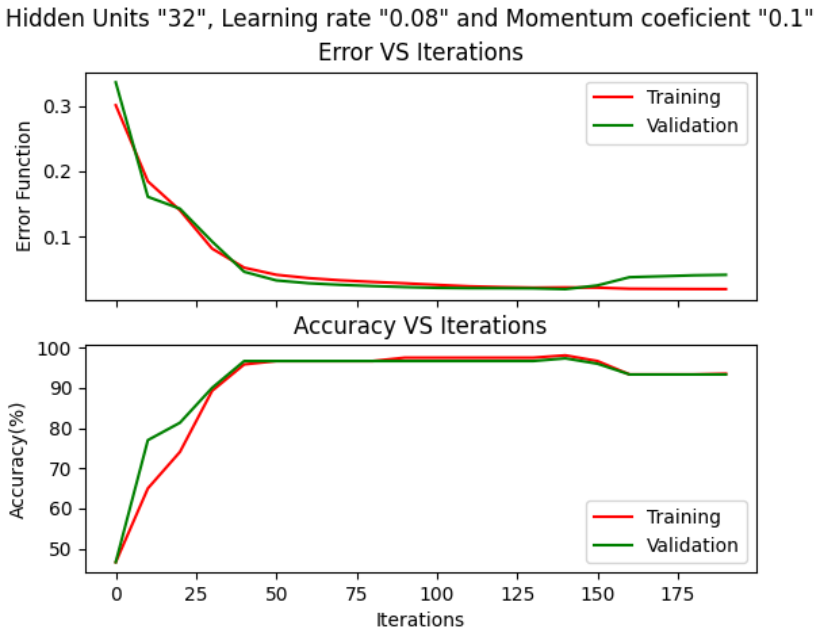
So it is better to not to use momentum co efficient for crab dataset.

I repeated the same process for Iris dataset that I followed for crab dataset to decide the hyperparameters. And the hyperparameters that I chose for Iris dataset were η= 0.08, epochs= 200 and hidden units = 32. The loss vs epoch and accuracy vs epoch curves with these hyper parameters



were:

The momentum coefficient for was helping the model to converge for iris dataset, which can be seen from the following graph:



By using this momentum coefficient we can see that the model has started overfitting after 150 epoch. So b using momentum coefficient 0.1 I can stop training after 75 epochs. The rest of the parameters will remain same.

Q3 This part requires for the student to obtain a little bit of self-familiarisation with some existing standard libraries (just to avoid implementing the evolutionary optimisers from scratch). Therefore, use a tool- box/library/package for the platform of your choice (various options are given below) to create a genetic algorithm (GA) and a particle swarm optimisation (PSO) to train the multilayer neural network from the previous Part 2. The GA and PSO optimisers should search for the best weights for the network to solve a simple problem (such as one from Part 2, or any different classification/regression problem of your choice). Only the forward pass and the overall output error need to be evaluated for the objective function of the GA and the PSO. For the GA, you can use any type of chromosome encoding you like (binary or real-valued, or both!) and any fitness scaling procedure (scaling/ranking), or genetic crossover/mutation operator you think is the most effcient for your problem. Try to briey justify your decisions for choosing the specific mechanisms or operators. Compare a few different options your library supports (e.g., different populalation/swarm sizes, generations, mutation/crossover rates, stopping criteria or whatever is supported by the implementation you use). Include a design section that explains how you designed your evolutionary-based MLP training, its main components, alternatives, and how you tested them. Extra marks will be given for experimentations with different GA and PSO operators. You can also experiment optionally with any other evolutionary optimisation method (other than PSO or GA) your library supports. Feel free to use whatever your libraries support, but please, explain what they do.

Ans:

Genetic Algorithms:

Genetic Algorithm is a stochiastic search algorithm which is based on natural selection process. It is used to optimise both the constrained as well as unconstrained problems.

The algorithm works in the following:

- We initialise the population of individuals with random values
    - We compute the fitness of each individual
    - The individuals with better fitness values are selected as parents to crossover and produce new children.(This corresponds to expoitation of good characteristics of available solution)
    - We also do the mutation of the offsprings (This corresponds to exploration for new chararcteristics of solution which current solutions do not have)
    - Replace the unfit parents with the fit offsprings
    - We repeat the whole process untill we meet some termination criteria which could be either the maximum number of generations have reached or the difference between the average fitness and the best fitness among the population is not significant.

Design:

As this alogrithm helps in optimising a function so we can merge it with multi layer neural network to train the parameters(weights and biases) of the model. The combination works in the following way;

1. We take the activation values of the output layer from the neural network using the forward path
2. Then we define an error function using these activation scores and actual label values of the data. This error function here is the function which we need to optimise or minimise using the genetic algorithm.
3. The parameters of the error function are actually the weights and biases which have been used in the forward path. Weights are in matrix form in the neural network, so they are flattened before passing on in the genetic algorithm.

4. When the error function is optimised then, it returns the values of parameters(weights and biases) with which model has been optimised. Then these parameters are converted back in to their original form(weights and biases in the respective matrix and vector form).

Genetic Algorithm Experiments:

For genetic algorithm, I have experimented on Iris dataset by chosing the different values of the hyperparameters and noted the corresponding lowest value of error function as well as the training and testing accuracy of the model. When I trained the neural network model in the Iris dataset, I could see that the range of weights and biases was beween -15 and 15. By consdering that I have the set the lower bound and the upper bound of the parameters equal to -15 and 15 respectively. I created a table for all the experiments, which is a follows:

Sr No.	Iteration	Population size	Mutation Probability	Elit Ratio	Crossover Probability	Parents Portion	Crossover Type	Variable Type	Objective functon	Accuracy
1	100	100	0.2	0.1	0.5	0.3	uniform	real	4.58	Training: 96% Testing: 100%
2	150	100	0.2	0.1	0.5	0.3	uniform	real	3.4	Training: 97.5% Testing: 97%
3	100	150	0.2	0.1	0.5	0.3	uniform	real	4.46	Training: 97% Testing: 100%
4	100	150	0.3	0.1	0.5	0.3	uniform	real	6.99	Training: 95% Testing: 97%
5	100	100	0.1	0.1	0.6	0.3	uniform	real	3.25	Training: 97% Testing: 97%
6	100	100	0.2	0.1	0.6	0.3	uniform	real	11.13	Training: 88% Testing: 87%
7	100	150	0.2	0.1	0.5	0.2	uniform	real	3.25	Training: 97.5% Testing: 97%
8	100	100	0.2	0.1	0.5	0.4	uniform	real	20.31	Training: 70% Testing: 53%
9	100	100	0.3	0.1	0.4	0.3	uniform	bool	45	Training: 35% Testing: 27%
10	100	100	0.2	0.1	0.5	0.3	two point	real	6.34	Training: 93.3% Testing: 93.3%
11	100	100	0.2	0.2	0.5	0.3	uniform	real	18.84	Training: 78.3% Testing: 63.3%

12	150	100	0.4	0.1	0.6	0.3	uniform	bool	45	Training: 35% Testing: 26.6%
13	100	150	0.2	0.1	0.5	0.3	Two point	real	3.69	Training: 97.5% Testing: 97%
14	100	150	0.3	0.1	0.5	0.2	uniform	real	2.9	Training: 97.5% Testing: 97%
15	100	100	0.2	0.1	0.5	0.3	One point	real	13.76	Training: 92% Testing: 90%

I was having the maximum accuracy and lowest error for the highlited combinations, I am attaching the graphs for these combinations only.

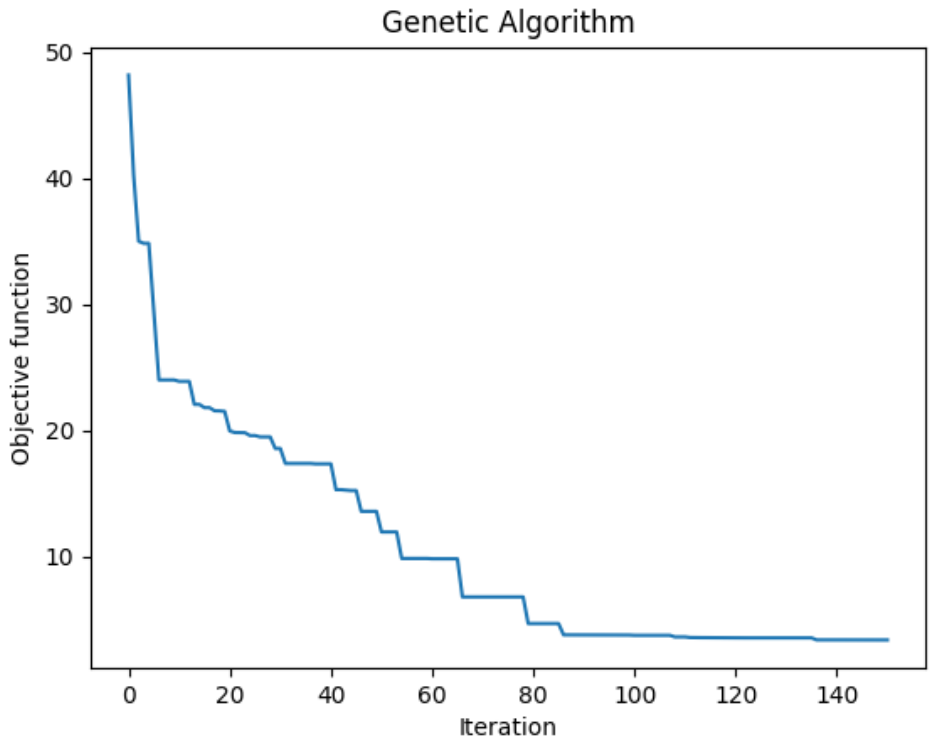


Fig 1. Iterations = 150, populationSize = 100, mutationProbability = 0.2, elitRatio = 0.1, crossoverProbability = 0.5, parentsPortion 0.3, crossoverType = “uniform”, variableType = “real”

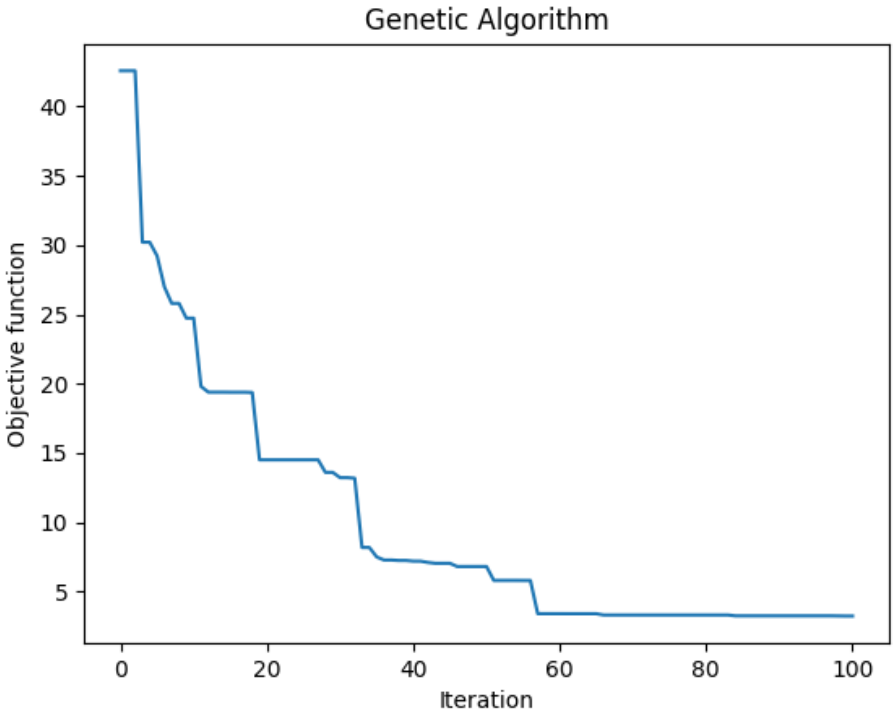


Fig 2. Iterations = 100, populationSize = 100, mutationProbability = 0.1, elitRatio = 0.1, crossoverProbability = 0.6, parentsPortion 0.3, crossoverType = “uniform”, variableType = “real

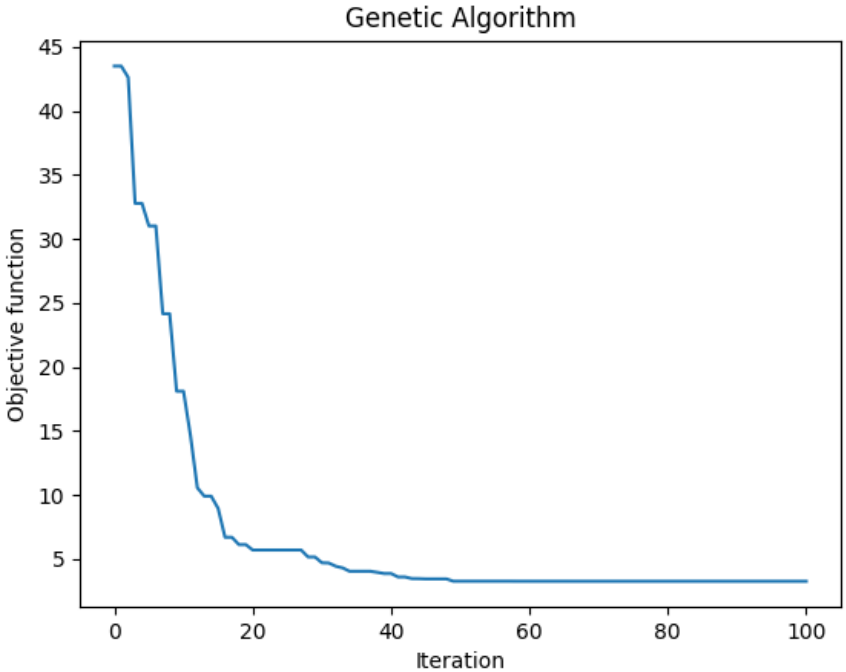


Fig 3. Iterations = 100, populationSize = 150, mutationProbability = 0.2, elitRatio = 0.1, crossoverProbability = 0.5, parentsPortion 0.2, crossoverType = “uniform”, variableType = “real”

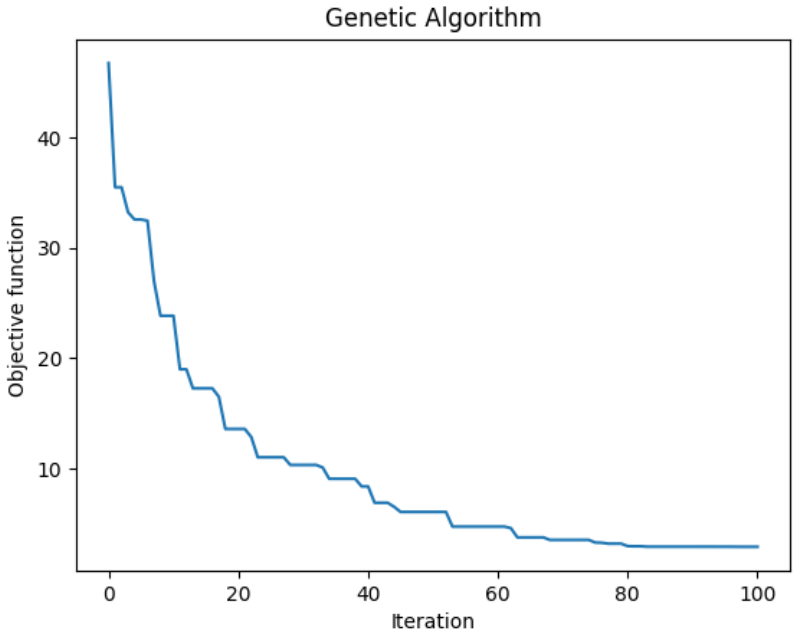


Fig 4. Iterations = 100, populationSize = 150, mutationProbability = 0.3, elitRatio = 0.1, crossoverProbability = 0.5, parentsPortion 0.2, crossoverType = “uniform”, variableType = “real”

Among all of these , Fig 3 is converging faster with the same accuracies as the other models, so I took the parameters of this model as the finalised parameters. The finalised parameter are Iterations = 80, populationSize = 150, mutationProbability = 0.2, elitRatio = 0.1, crossoverProbability = 0.5, parentsPortion 0.2, crossoverType = “uniform” and variableType = “real”.

Particle Swarm Optimisation:

PSO is a population based optimisation technique which is inspired by the social behaviour of the animals (like bird flocks and fish schools). The social interaction between the individuals of population helps in optimising the objective function. The population here is referenced as swarm and each individual is referenced as a particle.

The basic variant of the PSO works in the following way:

- The population is initialised with particles (typically the number of particles is 5 times the number of dimensions) and are assigned with random position and velocity. These random values are set to bound in some spefic range.
- The particles move around the in the search space, and memorise the information about their self known best position and swarm’s known best position.
- The movement of the particles in the next generation is guided by the known information about local best, global best and inertial component. This movement can be defined by the following equations:

  $$x_{ij} = x_{ij} + v_{ij}$$

  Where x_{ij} represents the position of ith particle and jth dimension. And v_{ij}represents the velocity of ith particle and jth dimension.

$$v_{ij} = wv_{ij} + c1.rand.\left(pbest_{ij} - x_{ij}\right) + c2.rand(gbest_{ij} - x_{ij})$$

Here wv_{ij} is the inertial component, $c1.rand.\left(pbest_{ij} - x_{ij}\right)$ is the cognitive component and $c2.rand(gbest_{ij} - x_{ij})$ is the social component.

- These positions and velocities are updated in every generation untill some specific criteria is met e.g., enough number of generations or the particles have found the global best position.

Design:

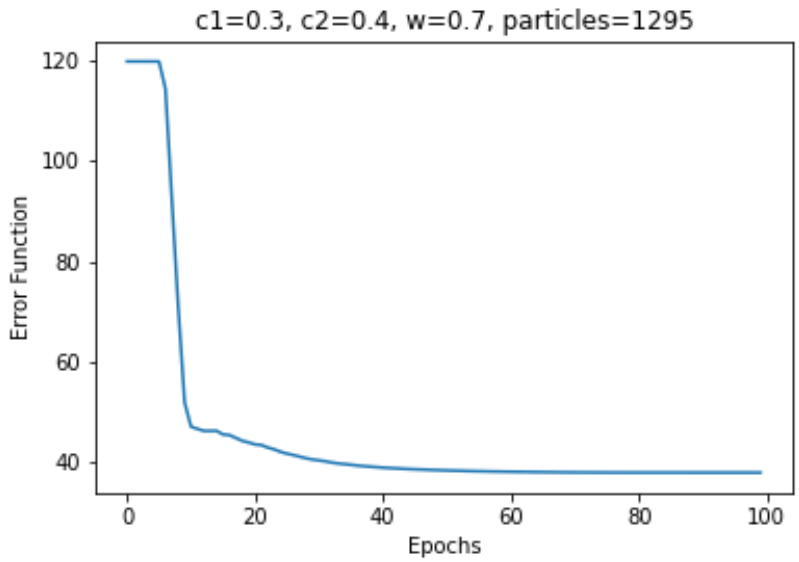
Like genetic algorithm, PSO can also be joined with the neural network to minimise the error function. The only difference in the design is that we use the PSO instead of GA as the optimisation technique. It works in the following way:

1. We take the activation values of the output layer from the neural network using the forward path
2. Then we define an error function using these activation scores and actual label values of the data. This error function here is the function which we need to optimise or minimise using the particle swarm optimisation.
3. The parameters of the error function are actually the weights and biases which have been used in the forward path. Weights are in matrix form in the neural network, so they are flattened before passing on in the PSO algorithm.
4. When the error function is optimised then, PSO algorithm returns the values of parameters(weights and biases) with which model has been optimised. Then these parameters are converted back in to their original form(weights and biases in the respective matrix and vector form).

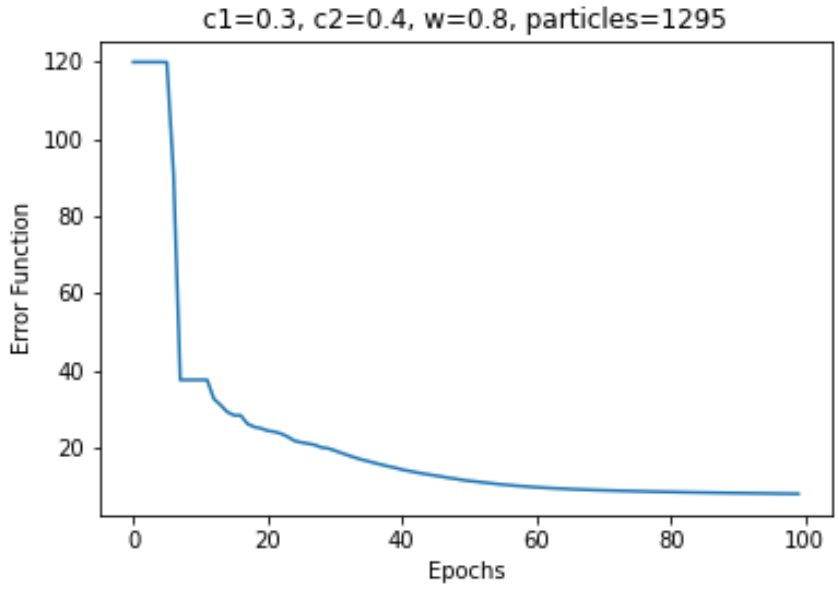
Experiments:

I experimented on Iris dataset. For my experiments I have taken the number of particles equal to 5 times the number of dimensions. I found with back propagation that 32 hidden units are optimal to use and by considering hidden units equal to 32, the number of dimensions for my model are 259 which makes the number of particles equal to 1295. I experimented with different values of acceleration constants and weight inertia.

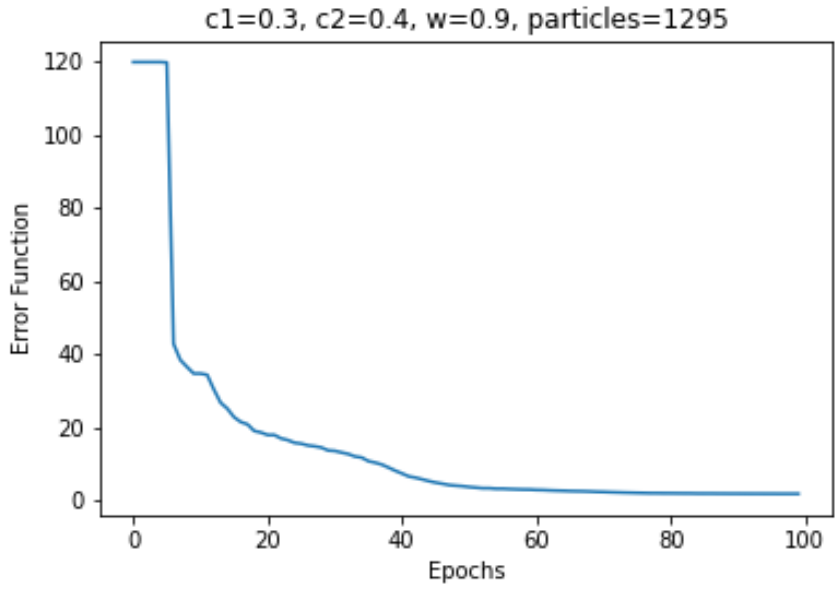
First I took constant values for c1 and c2 as 0.3 and 0.4 respectively and plotted the loss vs iteration graphs with different values of weight inertia. The results were as follows:



With Training Accuracy = 65% and Testing Accuracy = 73%

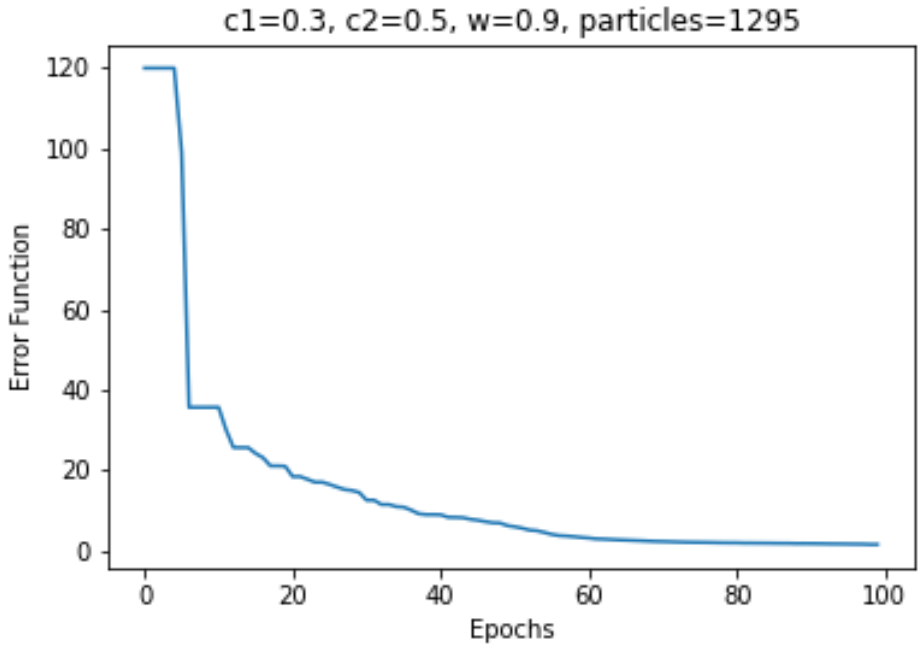


With Training Accuracy = 96% and Testing Accuracy = 100%

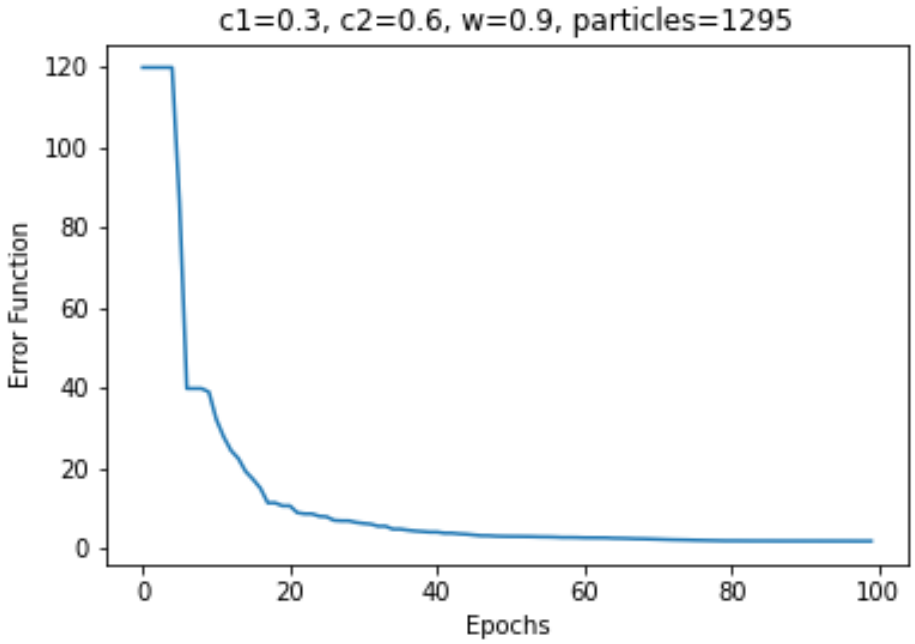


With Training Accuracy = 98.33% and Testing Accuracy = 100%

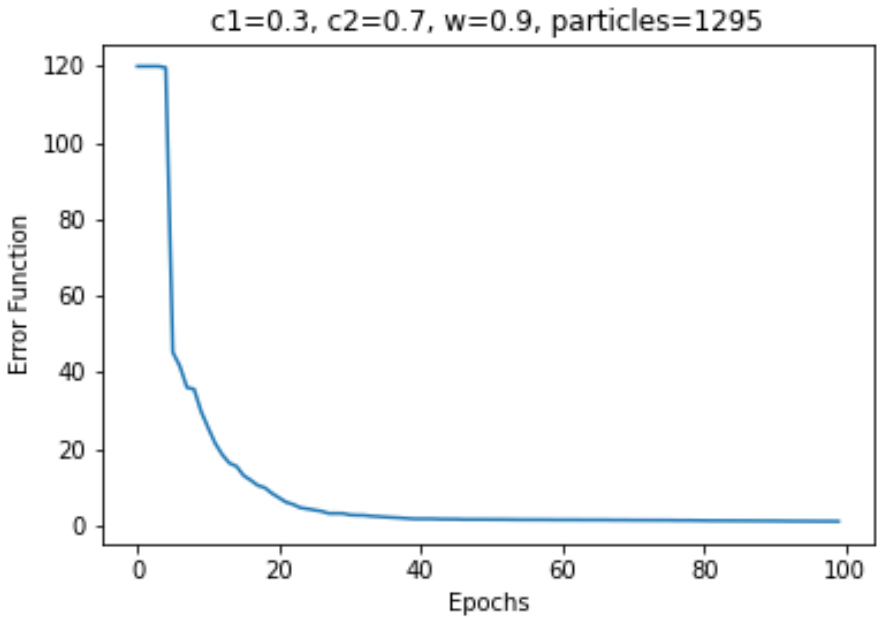
As I was having good accuracies with with w = 0.9, so I kept it constant now as 0.9 along with c1 = 0.3 and started varrying c2. The results I got were as follows:



With Training Accuracy = 99.16% and Testing Accuracy = 100%

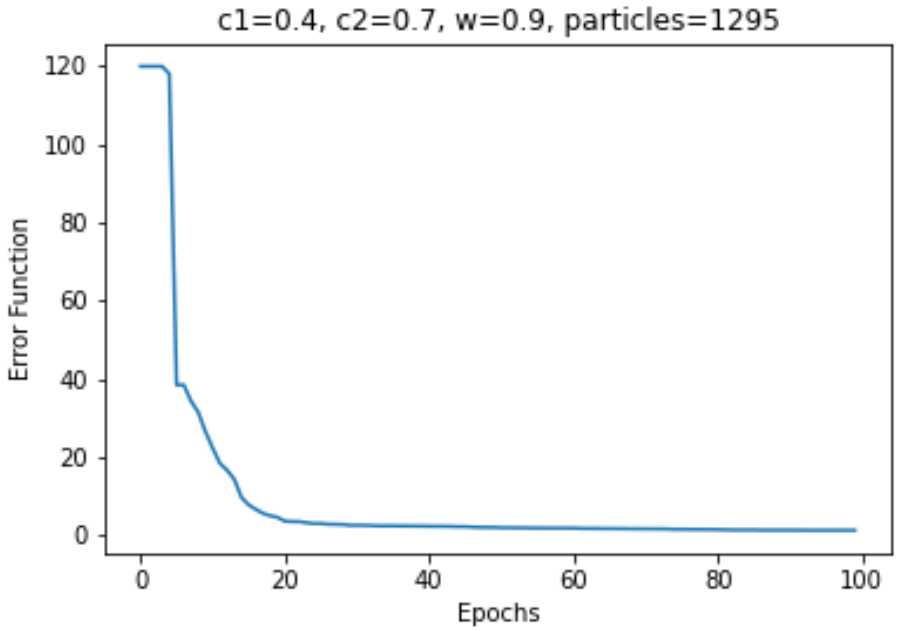


With Training Accuracy = 98.33% and Testing Accuracy = 100%

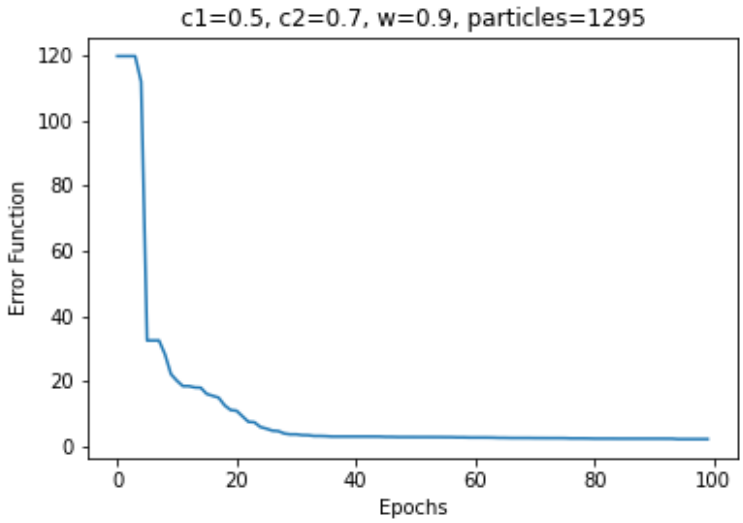


With Training Accuracy = 99.16% and Testing Accuracy = 100%

As I was having the better accuracy and earlier convergence with c2= 0.7, so I that constant along with w = 0.9 and started varying c1, and the results were as follows:



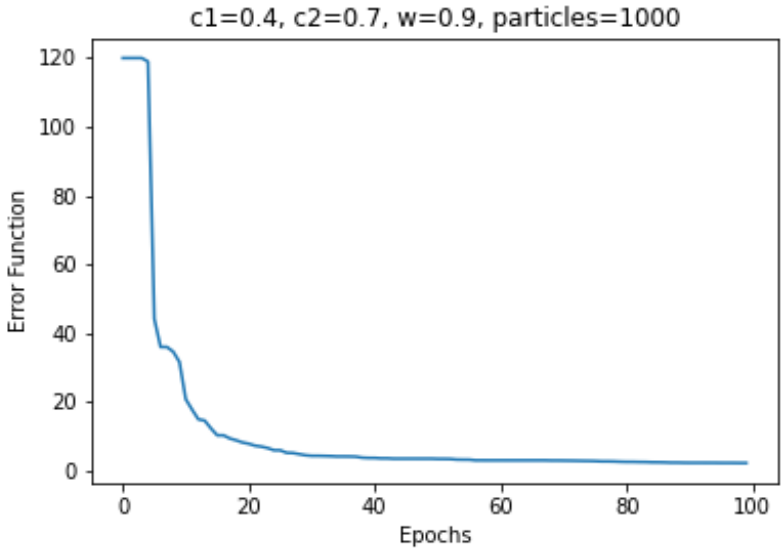
With Training Accuracy = 99.16% and Testing Accuracy = 100%



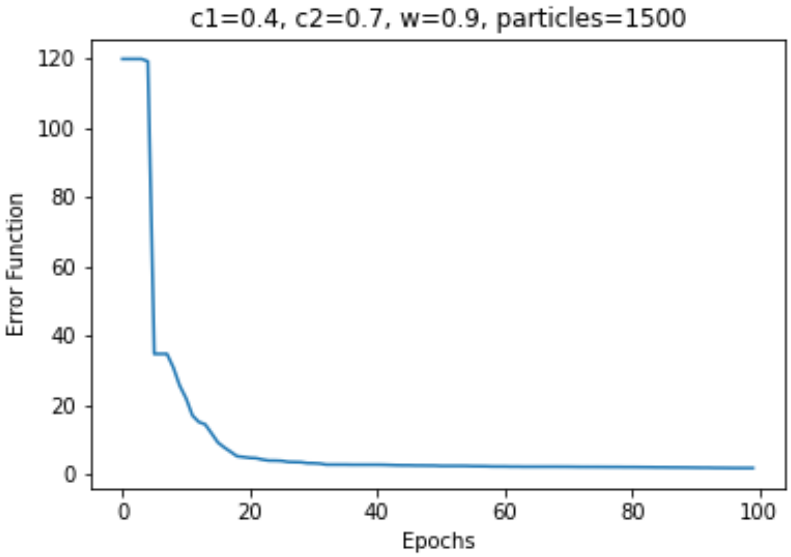
With Training Accuracy = 98.33% and Testing Accuracy = 100%

As the model was converging earlier with c1 = 0.4 as compared to c1 = 0.3, and was having the same accuracy so took 0.4 as finalised value.

Now I kept these coefficients constant and experimented with different values of the particles, and the results were as follows:



With Training Accuracy = 97.5% and Testing Accuracy = 100%

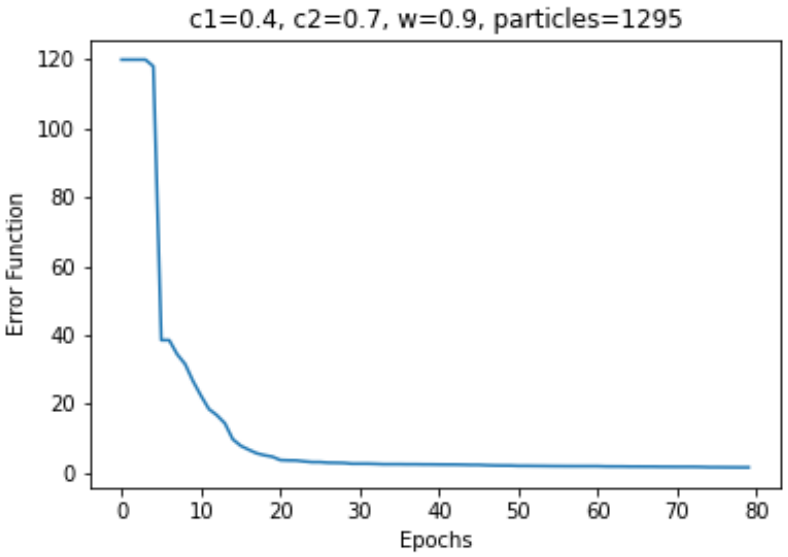


With Training Accuracy = 99.16% and Testing Accuracy = 100%

Accuracy on training dataset was decreasing by decreasing the number of particles below 5 times the number of dimensions. But it was not having significant impact beyond the 5 times number of dimensions. So I kept the number of particles equal to 5 times number of dimensions i.e., 1295.

Also we can see that the model is converging before 100 iterations, so the model can be trained for 80 iterations to be on safer side.

So, the finalised parameters are **32 hidden units, c1 = 0.4, c2 = 0.7, w=0.9, particles = 1295** and **iterations = 80**. The result of this model is as follows:



With Training Accuracy = 99.17% and Testing Accuracy = 10

Appendix:

Perceptron code:

```
from os import error
import numpy as np
import matplotlib.pyplot as plt
from numpy.lib.function_base import select
import pandas as pd
class Perceptron:
    """Perceptron classifier.

    Parameters:
    eta:
      Learning rate
    epoch
      The number of times the data is passed to train the model

    Attributes
    -----------
    weights : 1d-array
      Weights after fitting.
    bias :
      bias after fitting
    """
    def __init__(self, eta, epoch):
        self.eta = eta
        self.epoch = epoch

    def train(self, dataset, showLearning = False):
        """Fit training data.

        Parameters:
        dataset :
          training data
        showLearning :
          If True it plots the learned boundries after each epoch, other skip
it

        Returns
        self : object

        """

        X = dataset[:,:-1]
        label = dataset[:,-1]
        #Initialising the parameters
        self.weights = np.random.rand(X.shape[1])
        self.bias = np.random.rand()
```

```
        #Dictionary to store learned parameters for each epoch
        self.showLearning = {}

        for _ in range(self.epoch):#Going for each epoch
            errors = 0
            #This part is just to visualise the boundries while learning
            if showLearning == True:
                if X.shape[1] == 2:
                    x = np.linspace(-100,100,10)
                    y = (x * self.weights[0] + self.bias)/(-1*self.weights[1])
                    self.showLearning[_] = (x, y)
                elif X.shape[1] == 3:
                    x = np.linspace(-5,20,100)
                    y = np.linspace(-5,20,100)
                    x , y = np.meshgrid(x,y)
                    z = (-
self.weights[0] * x - self.weights[1] * y - self.bias)/self.weights[2]
                    self.showLearning[_] = (x, y, z)
                else:
                    assert "Dimension are more than three and learning is True
"
            #Going for each datapoint
            for datapoint, y in zip(X, label):
                a = self.predict(datapoint)
                update = self.eta * (y-a)
                #updating parameters if there is misclassification
                self.weights = self.weights + update * datapoint
                self.bias += update
                errors += int(update != 0.0)
            print("Epoch : ", _ + 1)
            print("Weights : ", self.weights)
            print("Bias : ", self.bias)



            if errors == 0: #Stopping the iteration, if there is no mis classi
fication for one epoch
                break

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.weights) + self.bias

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)

def selfCreated2D():
```

```
    #Creating the 2D dataset
    n = int(input("Enter the number of datapoints for which you want to genera
te the data : "))
    # np.random.seed(10)
    x = np.random.multivariate_normal((2,2), [[1,0],[0,1]], int(n/2))
    x1 = np.c_[x,np.ones(int(n/2))]
    y = np.random.multivariate_normal((10,5), [[1,0],[0,1]], int(n/2))
    y1 = np.c_[y,np.ones(int(n/2))*-1]
    whole_dataset = np.r_[x1,y1]
    np.random.shuffle(whole_dataset)


    while True:
        showLearning = input("Press 1 if you want to visualise the learning pr
ocess otherwise press 0 : ")
        if showLearning == "1":
            showLearning = True
            break
        elif showLearning == "0":
            showLearning = False
            break
        else:
            print("Enter correct input : ")

    #creating theb perceptron instance and training it
    pp1 = Perceptron(0.01, 100)
    pp1.train(whole_dataset, showLearning = showLearning)

    #Plotting the boundry during the learning
    if showLearning == True:
        for k in range(len(pp1.showLearning)):

            plt.scatter(x[:,0], x[:,1], color = "r", marker = "x")
            plt.scatter(y[:,0], y[:,1], color = "b", marker = "o")
            plt.plot(pp1.showLearning[k][0], pp1.showLearning[k][1])
            plt.xlabel("Feature 1")
            plt.ylabel("Feature 2")
            plt.title("Training")
            plt.xlim(-5,15)
            plt.ylim(-5,10)
            plt.show()
    else:
        x1 = np.linspace(-100,100,10)
        y1 = (x1 * pp1.weights[0] + pp1.bias)/(-1*pp1.weights[1])
        plt.scatter(x[:,0], x[:,1], color = "r", marker = "x")
        plt.scatter(y[:,0], y[:,1], color = "b", marker = "o")
        plt.xlim(-5,15)
        plt.ylim(-5,10)
```

```
        plt.xlabel("Feature 1")
        plt.ylabel("Feature 2")
        plt.plot(x1, y1)
        plt.show()




def selfCreated3D():
    #Creating the 2D dataset
    n = int(input("Enter the number of datapoints for which you want to genera
te the data : "))
    np.random.seed(10)
    x = np.random.multivariate_normal((2,7,2), [[1,0,0],[0,1,0],[0,0,1]], int(
n/2))
    x1 = np.c_[x,np.ones(int(n/2))]
    y = np.random.multivariate_normal((9,12,15), [[1,0,0],[0,1,0],[0,0,1]], in
t(n/2))
    y1 = np.c_[y,np.ones(int(n/2))*-1]
    whole_dataset = np.r_[x1,y1]
    np.random.shuffle(whole_dataset)


    showLearning = input("Press 1 if you want to visualise the learning proces
s otherwise press 0: ")
    while True:
        if showLearning == "1":
            showLearning = True
            break
        elif showLearning == "0":
            showLearning = False
            break
        else:
            print("Enter correct input : ")

    #creating theb perceptron instance and training it
    pp1 = Perceptron(0.01, 100)
    pp1.train(whole_dataset, showLearning = showLearning)

    #Plotting the boundry during the learning
    if showLearning == True:
        for k in range(len(pp1.showLearning)):
            ax = plt.axes(projection = "3d")
            ax.scatter3D(x[:,0], x[:,1], x[:,2], color = "r", marker = "x")
            ax.scatter3D(y[:,0], y[:,1], y[:,2],color = "b", marker = "o")
            ax.plot_surface(pp1.showLearning[k][0], pp1.showLearning[k][1], pp
1.showLearning[k][2])
            ax.set_xlabel('Feature 1')
```

```
            ax.set_ylabel('Feature 2')
            ax.set_zlabel('Feature 3')
            ax.set_title("Training ")
            plt.show()
    else:
        ax = plt.axes(projection = "3d")
        ax.scatter3D(x[:,0], x[:,1], x[:,2], color = "r", marker = "x")
        ax.scatter3D(y[:,0], y[:,1], y[:,2],color = "b", marker = "o")
        x = np.linspace(-5,20,100)
        y = np.linspace(-5,20,100)
        x , y = np.meshgrid(x,y)
        z = (-
pp1.weights[0] * x - pp1.weights[1] * y - pp1.bias)/pp1.weights[2]
        ax.plot_surface(x, y, z)
        ax.set_xlabel('Feature 1')
        ax.set_ylabel('Feature 2')
        ax.set_zlabel('Feature 3')
        plt.show()




def setosaVsVersicular():
    """
    This method loads the Iris dataset, extract the setosa and versicular obje
cts, and trains a perceptron on the that
    """
    #Loading the Iris dataset
    df = pd.read_csv('https://archive.ics.uci.edu/ml/'
            'machine-learning-databases/iris/iris.data', header=None)
    #Extracting the relevant data
    df = df[:100]
    df[4].iloc[:50] = 1
    df[4].iloc[50:] = -1
    df = np.array(df)
    np.random.seed(3284)
    np.random.shuffle(df)
    #creating theb perceptron instance and training it
    pp1 = Perceptron(0.01, 100)
    pp1.train(df, showLearning=True)


def main():
    """This method ask the user about the dataset on which you want to train t
he perceptron and direct
    the program to that dataset"""
    while True:
```

```
        menu = input("\nEnter from the following choices :\n1. Press 1 train f
or self created 2D dataset\n2. Press 2 train for self created 3D dataset\n3. P
ress 3 train for self setosa vs versicular dataset\n4. Press Q to quit : ")
        if menu == "1":
            selfCreated2D()
        if menu == "2":
            selfCreated3D()
        if menu == "3":
            setosaVsVersicular()
        elif menu == "q" or menu == "Q":
            break

if __name__ == "__main__":
    main()
```

Neural Network(Backpropagation) :

```
from matplotlib import colors
import numpy as np
from numpy.core.fromnumeric import argmax
from tqdm import tqdm
import pandas as pd
import matplotlib.pyplot as plt

class NeuralNetwork:
    """
    Feedforward neural network / Multi-
layer perceptron classifier. Uses back propagation
    to optimise the error function

    Parameters:
        iterations : Number of Epochs
        hiddenUnits : Number of neurons in the hidden layer
        eta : Learning rate
    """
    def __init__(self, iteration = 1000, eta= 0.01, hiddenUnits = 15):
        self.iteration = iteration
        self.eta = eta
        self.hiddenUnits = hiddenUnits

    def sigmoid(self, v):
        """Compute the logistic function()sigmoid"""
        return 1 / (1 + np.exp(-np.clip(v, -250, 250)))

    def forward(self, X):
        """
```

```
        This method takes a datapoint, pass it through the forward path of neu
ral network and
        returns the induced local field and activation scores of the output la
yer
        """
        netInputHidden = np.dot(self.weight_h.T, X) + self.bias_h
        netOutHidden = self.sigmoid(netInputHidden)
        self.netInputOuterLayer = np.dot(self.weight_o.T, netOutHidden) + self
.bias_o
        netOutOuterLayer = self.sigmoid(self.netInputOuterLayer)

        return netOutHidden, netOutOuterLayer

    def derivativeSigmoid(self, v):
        """Compute the derivative of logistic function()sigmoid"""

        return v * (1 - v)

    def fittingWithBackPropagation(self, dataset, label, validationX, validati
onLabel):
        """
        This method takes training and testing dataset, trains the model on th
e given dataset by
        using the backpropagation and plots the loss vs epoch and accuracy vs
epoch curve. It also
        prints out the accuracies on the training as well as testing set
        """
        #Initialisation of parameters
        self.weight_h = np.random.normal(0, 0.1, (dataset.shape[1], self.hidde
nUnits))
        self.weight_o = np.random.normal(0, 0.1, (self.hiddenUnits, label.shap
e[1]))
        self.bias_h = np.random.normal(0, 0.1, (self.hiddenUnits))
        self.bias_o = np.random.normal(0, 0.1, (label.shape[1]))
        validationAccuracies = []
        trainingAccuracies = []
        trainingError = []
        validationError = []
        for _ in tqdm(range(self.iteration)): #Going through each epoch
            averageTrainingError = 0
            averageValidationError = 0
            for datapoint, d in zip(dataset, label): #Going through each datap
oint
                #Going through foward path
                y_hidden, y_out = self.forward(datapoint)

                #Doing back propagation
                e = d - y_out
```

```
                averageTrainingError += 0.5 * np.sum((d - y_out) ** 2)
                deltaOut = e * self.derivativeSigmoid(y_out)
                self.weight_o = self.weight_o + self.eta * np.outer(y_hidden,
deltaOut)
                self.bias_o = self.bias_o + self.eta * deltaOut
                deltaHidden = self.derivativeSigmoid(y_hidden) * np.sum(np.dot
(self.weight_o, deltaOut))
                self.weight_h = self.weight_h + self.eta * np.outer(datapoint,
 deltaHidden)
                self.bias_h = self.bias_h + self.eta * deltaHidden

            # Going through validation data and tracking the error and accurac
ies
            for datapoint, d in zip(validationX, validationLabel):
                y_hidden, y_out = self.forward(datapoint)


                averageValidationError += 0.5 * np.sum((d - y_out) ** 2)
            averageTrainingError /= dataset.shape[0]
            averageValidationError /= validationX.shape[0]

            validationError.append(averageValidationError)
            trainingError.append(averageTrainingError)

            trainingAccuracies.append(accuracy(self, dataset, label))
            validationAccuracies.append(accuracy(self, validationX, validation
Label))

        #Plotting the accuracy vs epoch and accuracy vs loss curves
        self.plot(trainingAccuracies, validationAccuracies, trainingError, val
idationError)

    def plot(self, trainigAccuracy, validationAccuracy, trainingError, validat
ionError):
        """
        This method takes the error function and accuracy of training and vali
dation set and
        plot their curves
        """

        movingAverage = 10
        smoothenedTrainigAccuracy = []
        smoothenedValidationAccuracy = []
        smoothenedTrainigError = []
        smoothenedValidationError = []

        #Smoothening the loss and accuracy curves over the moving average of 1
0
        for i in range(0, self.iteration, movingAverage):
```

```
            smoothenedTrainigAccuracy.append(sum(trainigAccuracy[i:i+movingAve
rage])/movingAverage)
            smoothenedValidationAccuracy.append(sum(validationAccuracy[i:i+mov
ingAverage])/movingAverage)
            smoothenedTrainigError.append(sum(trainingError[i:i+movingAverage]
)/movingAverage)
            smoothenedValidationError.append(sum(validationError[i:i+movingAve
rage])/movingAverage)
        f, (ax1,ax2) = plt.subplots(2, 1, sharex=True)
        ax1.plot(range(0, self.iteration, movingAverage), smoothenedTrainigErr
or, color = "r", label = "Training")
        ax1.plot(range(0, self.iteration, movingAverage), smoothenedValidation
Error, color = "g", label = "Validation")
        ax1.legend()
        ax1.set_title("Error VS Iterations")
        ax1.set_ylabel("Error Function")
        ax2.plot(range(0, self.iteration, movingAverage), smoothenedTrainigAcc
uracy, color = "r", label = "Training")
        ax2.plot(range(0, self.iteration, movingAverage), smoothenedValidation
Accuracy, color = "g", label = "Validation")
        ax2.set_title("Accuracy VS Iterations")
        ax2.set_xlabel("Iterations")
        ax2.set_ylabel("Accuracy(%)")
        ax2.legend()
        plt.suptitle('Hidden Units "' + str(self.hiddenUnits) + '"' + ' and Le
arning rate "' + str(self.eta) + '"')
        plt.show()

def takeHyperParameters():
    '''This function takes the hyperparameters from the user and return these
hyoer parameters'''

    print("\nEnter the following hyperparameters\n")
    iterations = int(input("Enter the number of Epochs for which you want to t
rain the model : "))
    eta = float(input("Enter the value of learning rate : "))
    hiddenUnits = int(input("Enter the number of hidden units : "))
    return iterations, eta, hiddenUnits


def accuracy(nn, X, y):
    """
        This method takes the dataset and runs it through the forward path of
the network, gets the
        activation score of the output layer, check it against the given label
 and give a prediction
        about which class it belongs to. By taking the account of true predict
ions, at the end it returns
        the accuracy of the model on the dataset
```

```
    """
    tp = 0
    for data, d in zip(X, y):
        a,b = nn.forward(data)
        if argmax(b) == argmax(d):
            tp += 1
        accuracy = round(tp/y.shape[0] * 100, 2)
    return accuracy

def crab():
    """
    This method loads the crab dataset and train a neural network model on the
 dataset
    """
    #Loading the crab dataset
    with open("crabData.csv") as file:
        data = []
        for line in file:
            data.append(line.strip().split(","))
        file.close()

    np.random.seed(4738)
    #Cleaning and splitting the data in training and testing data
    data = np.array(data)
    data = data[1:,:]
    np.random.shuffle(data)
    dataDivision = int(data.shape[0]*0.8)
    X = data[:dataDivision, 3:].astype(np.float)
    validationX = data[dataDivision:, 3:].astype(np.float)
    #One hot encoding the labels
    Y = np.array(pd.get_dummies(data[:, 1]))
    trainingLabel = Y[:dataDivision,:]
    validationLabel = Y[dataDivision:,:]
    iterations, eta, hiddenUnits = takeHyperParameters()

    #initialising and training the network
    network = NeuralNetwork(iteration = iterations,hiddenUnits=hiddenUnits, et
a= eta)
    network.fittingWithBackPropagation(X, trainingLabel, validationX, validati
onLabel)
    print("Training Accuracy : " + str(accuracy(network, X, trainingLabel)) +
"%")
    print("Testing Accuracy : " + str(accuracy(network, validationX, validatio
nLabel)) + "%")


def Iris():
    """
```

```
    This method loads the Iris dataset and train a neural network model on the
 dataset
    """
    #Loading the Iris dataset
    df = pd.read_csv('https://archive.ics.uci.edu/ml/'
            'machine-learning-databases/iris/iris.data', header=None)

    np.random.seed(1210)
    #Cleaning and splitting the data in training and testing data
    permutaion = np.random.permutation(150)
    df = df.iloc[permutaion]
    #One hot encoding the labels
    Y = np.array(pd.get_dummies(df[4]))
    data = np.array(df)
    data = data[:,:4]
    dataDivision = int(data.shape[0]*0.8)
    X = data[:dataDivision,:].astype(np.float)
    trainingLabel = Y[:dataDivision,:]
    validationLabel = Y[dataDivision:,:]
    validationX = data[dataDivision:, :].astype(np.float)
    iterations, eta, hiddenUnits = takeHyperParameters()

    #initialising and training the network
    network = NeuralNetwork(iteration = iterations,hiddenUnits=hiddenUnits, et
a= eta)
    network.fittingWithBackPropagation(X, trainingLabel, validationX, validati
onLabel)
    print("Training Accuracy : " + str(accuracy(network, X, trainingLabel)) +
"%")
    print("Testing Accuracy : " + str(accuracy(network, validationX, validatio
nLabel)) + "%")


def main():
    """This method ask the user about the dataset on which you want to train y
our model and direct
    the program to that dataset"""

    print("\n\t\t\tWelcome")
    while True:
        print("\nChoose from the following on which dataset you want to train
your model")
        menu = input("1. Press 1 for Iris dataset(Classification between Setos
a, Versicular and Virginka)\n2. Press 2 for Crab dataset(Classification betwee
n Male and Female)\n3. Press Q/q to quit : ")
        if menu == "1":
            Iris()
        elif menu == "2":
            crab()
```

```
        elif menu == "Q" or menu == "q":
            break
        else:
            print("\nEnter Valid Entry\n")


if __name__ == "__main__":
    main()
```

Neural Network (Backpropagattion, Implementation with momentum term):

```
from matplotlib import colors
import numpy as np
from numpy.core.fromnumeric import argmax
from tqdm import tqdm
import pandas as pd
import matplotlib.pyplot as plt

class NeuralNetwork:
    """
    Feedforward neural network / Multi-
layer perceptron classifier. Uses back propagation
    to optimise the error function

    Parameters:
        iterations : Number of Epochs
        hiddenUnits : Number of neurons in the hidden layer
        eta : Learning rate
        mu : Momentum Co efficient
    """
    def __init__(self, iteration = 1000, eta= 0.01, hiddenUnits = 15, mu = 0.1
):
        self.iteration = iteration
        self.eta = eta
        self.hiddenUnits = hiddenUnits
        self.mu = mu

    def sigmoid(self, v):
        """Compute the logistic function()sigmoid"""
        return 1 / (1 + np.exp(-np.clip(v, -250, 250)))

    def forward(self, X):
        """
        This method takes a datapoint, pass it through the forward path of neu
ral network and
        returns the induced local field and activation scores of the output la
yer
        """
```

```
        netInputHidden = np.dot(self.weight_h.T, X) + self.bias_h
        netOutHidden = self.sigmoid(netInputHidden)
        netInputOuterLayer = np.dot(self.weight_o.T, netOutHidden) + self.bias
_o
        netOutOuterLayer = self.sigmoid(netInputOuterLayer)

        return netOutHidden, netOutOuterLayer

    def derivativeSigmoid(self, v):
        """Compute the derivative of logistic function()sigmoid"""
        return v * (1 - v)

    def fittingWithBackPropagation(self, dataset, label, validationX, validati
onLabel):
        """
        This method takes training and testing dataset, trains the model on th
e given dataset by
        using the backpropagation and plots the loss vs epoch and accuracy vs
epoch curve. It also
        prints out the accuracies on the training as well as testing set
        """

        np.random.seed(100)
        #Initialisation of parameters
        self.weight_h = np.random.normal(0, 0.1, (dataset.shape[1], self.hidde
nUnits))
        self.weight_o = np.random.normal(0, 0.1, (self.hiddenUnits, label.shap
e[1]))
        self.bias_h = np.random.normal(0, 0.1, (self.hiddenUnits))
        self.bias_o = np.random.normal(0, 0.1, (label.shape[1]))
        update_o = np.zeros(self.weight_o.shape)
        update_h = np.zeros(self.weight_h.shape)
        validationAccuracies = []
        trainingAccuracies = []
        trainingError = []
        validationError = []
        for _ in tqdm(range(self.iteration)): #Going through each epoch
            averageTrainingError = 0
            averageValidationError = 0
            for datapoint, d in zip(dataset, label): #Going through each datap
oint

                #Going through foward path
                y_hidden, y_out = self.forward(datapoint)

                #Doing back propagation
                e = d - y_out
                averageTrainingError += 0.5 * np.sum((d - y_out) ** 2)
```

```
                deltaOut = e * self.derivativeSigmoid(y_out)
                update_o = self.eta * np.outer(y_hidden, deltaOut) + self.mu *
 update_o
                self.weight_o = self.weight_o + update_o
                self.bias_o = self.bias_o + self.eta * deltaOut
                deltaHidden = self.derivativeSigmoid(y_hidden) * np.sum(np.dot
(self.weight_o, deltaOut))
                update_h = self.eta * np.outer(datapoint, deltaHidden) + self.
mu * update_h
                self.weight_h = self.weight_h + update_h
                self.bias_h = self.bias_h + self.eta * deltaHidden


            # Going through validation data and tracking the error and accurac
ies
            for datapoint, d in zip(validationX, validationLabel):
                y_hidden, y_out = self.forward(datapoint)
                averageValidationError += 0.5 * np.sum((d - y_out) ** 2)
            averageTrainingError /= dataset.shape[0]
            averageValidationError /= validationX.shape[0]


            validationError.append(averageValidationError)
            trainingError.append(averageTrainingError)


            trainingAccuracies.append(accuracy(self, dataset, label))
            validationAccuracies.append(accuracy(self, validationX, validation
Label))

        #Plotting the accuracy vs epoch and accuracy vs loss curves
        self.plot(trainingAccuracies, validationAccuracies, trainingError, val
idationError)

    def plot(self, trainigAccuracy, validationAccuracy, trainingError, validat
ionError):
        """
        This method takes the error function and accuracy of training and vali
dation set and
        plot their curves
        """

        movingAverage = 10
        smoothenedTrainigAccuracy = []
        smoothenedValidationAccuracy = []
        smoothenedTrainigError = []
        smoothenedValidationError = []
```

```
        #Smoothening the loss and accuracy curves over the moving average of 1
0
        for i in range(0, self.iteration, movingAverage):
            smoothenedTrainigAccuracy.append(sum(trainigAccuracy[i:i+movingAve
rage])/movingAverage)
            smoothenedValidationAccuracy.append(sum(validationAccuracy[i:i+mov
ingAverage])/movingAverage)
            smoothenedTrainigError.append(sum(trainingError[i:i+movingAverage]
)/movingAverage)
            smoothenedValidationError.append(sum(validationError[i:i+movingAve
rage])/movingAverage)
        f, (ax1,ax2) = plt.subplots(2, 1, sharex=True)
        ax1.plot(range(0, self.iteration, movingAverage), smoothenedTrainigErr
or, color = "r", label = "Training")
        ax1.plot(range(0, self.iteration, movingAverage), smoothenedValidation
Error, color = "g", label = "Validation")
        ax1.legend()
        ax1.set_title("Error VS Iterations")
        ax1.set_ylabel("Error Function")
        ax2.plot(range(0, self.iteration, movingAverage), smoothenedTrainigAcc
uracy, color = "r", label = "Training")
        ax2.plot(range(0, self.iteration, movingAverage), smoothenedValidation
Accuracy, color = "g", label = "Validation")
        ax2.set_title("Accuracy VS Iterations")
        ax2.set_xlabel("Iterations")
        ax2.set_ylabel("Accuracy(%)")
        ax2.legend()
        plt.suptitle('Hidden Units "' + str(self.hiddenUnits) + '"' + ', Learn
ing rate "' + str(round(self.eta, 2)) + '"' + ' and Momentum coeficient "' + s
tr(round(self.mu, 2)) + '"')
        plt.show()

def takeHyperParameters():
    '''This function takes the hyperparameters from the user and return these
hyoer parameters'''

    print("\nEnter the following hyperparameters\n")
    iterations = int(input("Enter the number of Epochs for which you want to t
rain the model : "))
    eta = float(input("Enter the value of learning rate : "))
    hiddenUnits = int(input("Enter the number of hidden units : "))
    mu = float(input("Enter the momentum coefficient : "))
    return iterations, eta, hiddenUnits, mu


def accuracy(nn, X, y):
    """
```

```
        This method takes the dataset and runs it through the forward path of
the network, gets the
        activation score of the output layer, check it against the given label
 and give a prediction
        about which class it belongs to. By taking the account of true predict
ions, at the end it returns
        the accuracy of the model on the dataset
    """
    tp = 0
    for data, d in zip(X, y):
        a,b = nn.forward(data)
        if argmax(b) == argmax(d):
            tp += 1
        accuracy = round(tp/y.shape[0] * 100, 2)
    return accuracy

def crab():
    """
    This method loads the crab dataset and train a neural network model on the
 dataset
    """
    #Loading the crab dataset
    with open("crabData.csv") as file:
        data = []
        for line in file:
            data.append(line.strip().split(","))
        file.close()

    np.random.seed(4738)
    #Cleaning and splitting the data in training and testing data
    data = np.array(data)
    data = data[1:,:]
    np.random.shuffle(data)
    dataDivision = int(data.shape[0]*0.8)
    X = data[:dataDivision, 3:].astype(np.float)
    validationX = data[dataDivision:, 3:].astype(np.float)
    #One hot encoding the labels
    Y = np.array(pd.get_dummies(data[:, 1]))
    trainingLabel = Y[:dataDivision,:]
    validationLabel = Y[dataDivision:,:]
    iterations, eta, hiddenUnits, mu = takeHyperParameters()

    #initialising and training the network
    network = NeuralNetwork(iteration = iterations,hiddenUnits=hiddenUnits, et
a= eta, mu = mu)
    network.fittingWithBackPropagation(X, trainingLabel, validationX, validati
onLabel)
```

```
    print("Training Accuracy : " + str(accuracy(network, X, trainingLabel)) +
"%")
    print("Testing Accuracy : " + str(accuracy(network, validationX, validatio
nLabel)) + "%")


def Iris():
    """
    This method loads the Iris dataset and train a neural network model on the
 dataset
    """
    #Loading the Iris Dataset
    df = pd.read_csv('https://archive.ics.uci.edu/ml/'
            'machine-learning-databases/iris/iris.data', header=None)

    np.random.seed(1210)
    #Cleaning and splitting the data in training and testing data
    permutaion = np.random.permutation(150)
    df = df.iloc[permutaion]
    #One hot encoding the labels
    Y = np.array(pd.get_dummies(df[4]))
    data = np.array(df)
    data = data[:,:4]
    dataDivisior = int(data.shape[0]*0.8)
    X = data[:dataDivision,:].astype(np.float)
    trainingLabel = Y[:dataDivision,:]
    validationLabel = Y[dataDivision:,:]
    validationX = data[dataDivision:, :].astype(np.float)
    iterations, eta, hiddenUnits, mu = takeHyperParameters()

    #initialising and training the network
    network = NeuralNetwork(iteration = iterations,hiddenUnits=hiddenUnits, et
a= eta, mu = mu)
    network.fittingWithBackPropagation(X, trainingLabel, validationX, validati
onLabel)
    print("Training Accuracy : " + str(accuracy(network, X, trainingLabel)) +
"%")
    print("Testing Accuracy : " + str(accuracy(network, validationX, validatio
nLabel)) + "%")


def main():
    """This method ask the user about the dataset on which you want to train y
our model and direct
    the program to that dataset"""

    print("\n\t\t\tWelcome")
    while True:
```

```
        print("\nChoose from the following on which dataset you want to train
your model")
        menu = input("1. Press 1 for Iris dataset(Classification between Setos
a, Versicular and Virginka)\n2. Press 2 for Crab dataset(Classification betwee
n Male and Female)1\n3. Press Q/q to quit : ")
        if menu == "1":
            Iris()
        elif menu == "2":
            crab()
        elif menu == "Q" or menu == "q":
            break
        else:
            print("\nEnter Valid Entry\n")


if __name__ == "__main__":
    main()
```

Neural Network (Optimising by GA):

```
import numpy as np
from numpy.core.fromnumeric import argmax
from geneticalgorithm import geneticalgorithm as ga
import pandas as pd

class NeuralNetwork:
    """
    Feedforward neural network / Multi-
layer perceptron classifier. Uses Genetic Algorithm
    to optimise the error function

    Parameters:
        iterations : Number of Epochs
        hiddenUnits : Number of neurons in the hidden layer
        populationSize : Number of individuals in the population
        mutationProbability : Mutation rate
        eliteRatio : Proportion of elites in the population
        crossoverProbability : Crossover Rate
        parentsPortion : The portion of population filled by the members of th
e previous generation
        crossoverType : Type of crossover (uniform, one_point, two_point)
        variableType : Type of variable (Real or Bool)
        lb : Lower bound of the variable
        ub : upper bound of the variable
    """
    def __init__(self, hiddenUnits = 32, iterations  = 100, populationSize = 1
50, mutationProbability = 0.2, elitRatio = 0.1, crossoverProbability = 0.5, pa
```

```
rentsPortion = 0.2, crossoverType = "uniform", variableType = "real", lb = -
15, ub = 15):
        self.hiddenUnits = hiddenUnits
        self.iterations = iterations
        self.populationSize = populationSize
        self.mutationProbability = mutationProbability
        self.elitRatio = elitRatio
        self.crossoverProbability = crossoverProbability
        self.parentsPortion = parentsPortion
        self.crossoverType = crossoverType
        self.variableType = variableType
        self.lb = lb
        self.ub = ub

    def forward(self, X):
        """
        This method takes a datapoint, pass it through the forward path of neu
ral network and
        returns the activation scores of the output layer
        """

        netInputHidden = np.dot(X, self.w_hidden) + self.b_hidden
        netOutHidden = self.sigmoid(netInputHidden)
        netInputOuterLayer = np.dot(netOutHidden, self.w_out) + self.b_out
        netOutOuterLayer = self.sigmoid(netInputOuterLayer)

        return netOutOuterLayer

    def sigmoid(self, v):
        """Compute the logistic function()sigmoid"""
        return 1 / (1 + np.exp(-np.clip(v, -250, 250)))

    def errorFunction(self,X):
        """
        This method is the error function that we want to otimise through GA
        """

        #Splitting falttened vector of parameters in respective sets of matrix
(weights) and vector form(bias)
        self.w_hidden = X[:int(self.dataset.shape[1] * self.hiddenUnits)].copy
().reshape((self.dataset.shape[1],self.hiddenUnits))
        self.b_hidden = X[int(self.dataset.shape[1] * self.hiddenUnits):int((s
elf.dataset.shape[1] * self.hiddenUnits) + self.hiddenUnits)].copy()
        self.w_out = X[int((self.dataset.shape[1] * self.hiddenUnits) + self.h
iddenUnits):int((self.dataset.shape[1] * self.hiddenUnits) + self.hiddenUnits
+ (self.hiddenUnits*self.label.shape[1]))].copy().reshape((self.hiddenUnits,se
lf.label.shape[1]))
```

```
        self.b_out = X[int((self.dataset.shape[1] * self.hiddenUnits) + self.h
iddenUnits + (self.hiddenUnits*self.label.shape[1])):].copy()

        netOutOuterLayer = self.forward(self.dataset)
        #mean squared error
        e = np.sum(0.5*(self.label - netOutOuterLayer)**2)

        return e

    def fittingWithGA(self, dataset, label):
        """
        This method takes training and testing dataset, trains the model on th
e given dataset by
        using the GA and plots the loss vs epoch curve.
        """

        self.dataset = dataset
        self.label = label
        #Getting the dimensions to Flatten the matrix of weights and biases to
 a single  vector
        d = (dataset.shape[1] * self.hiddenUnits) + self.hiddenUnits + (self.h
iddenUnits * label.shape[1]) + self.label.shape[1]
        #Passing the hyperparameters
        boundary = np.c_[np.zeros(d) + self.lb, np.zeros(d)+ self. ub]
        algorithm_param = {'max_num_iteration': self.iterations,\
            'population_size':self.populationSize,\
                'mutation_probability':self.mutationProbability,\
            'elit_ratio':self.elitRatio,\
                'crossover_probability': self.crossoverProbability,\
                'parents_portion': self.parentsPortion,\
                'crossover_type':self.crossoverType,\
                'max_iteration_without_improv':None}
        #calling instance of GA
        model= ga(function=self.errorFunction, dimension=d,variable_boundaries
= boundary , variable_type=self.variableType, algorithm_parameters=algorithm_p
aram)
        model.run()
        solution=model.output_dict["variable"]
        #converting the optimised flattened vector back to matrix and vector f
orm(for weights and biases respectively)
        self.w_hidden = solution[:int(self.dataset.shape[1] * self.hiddenUnits
)].copy().reshape((self.dataset.shape[1],self.hiddenUnits))
        self.b_hidden = solution[int(self.dataset.shape[1] * self.hiddenUnits)
:int((self.dataset.shape[1] * self.hiddenUnits) + self.hiddenUnits)].copy()
        self.w_out = solution[int((self.dataset.shape[1] * self.hiddenUnits) +
 self.hiddenUnits):int((self.dataset.shape[1] * self.hiddenUnits) + self.hidde
nUnits + (self.hiddenUnits*self.label.shape[1]))].copy().reshape((self.hiddenU
nits,self.label.shape[1]))
```

```
        self.b_out = solution[int((self.dataset.shape[1] * self.hiddenUnits) +
 self.hiddenUnits + (self.hiddenUnits*self.label.shape[1])):].copy()

    def evaluation(self,dataset, label, accuracyType):
        """
        This method takes the dataset and runs it through the forward path of
the network, gets the
        activation score of the output layer, check it against the given label
 and give a prediction
        about which class it belongs to. By taking the account of true predict
ions, at the end it prints
        out the accuracy of the model on the dataset
        """
        Tp = 0
        for data, d in zip(dataset, label):
            b = self.forward(data)
            if argmax(b) == argmax(d):
                Tp += 1
        print(accuracyType + " : " + str(round(Tp/label.shape[0] * 100, 2)) +
"%")


def takeHyperParameters():
    '''This function takes the hyperparameters from the user and return these
hyoer parameters'''

    print("\nEnter the following hyperparameters\n")
    hiddenUnits = int(input("Enter the number of hidden units : "))
    iterations = int(input("Enter the number of Epochs for which you want to t
rain the model : "))
    populationSize = int(input("Enter the Population Size : "))
    mutationProbability = float(input("Enter the mutation probability : "))
    crossoverProbability = float(input("Enter the Crossover Probability : "))
    crossoverType = input("Enter the crossover type(uniform, one_point, two_po
int) : ")
    variableType = input("Enter the variable type(real or bool) : ")
    lb = float(input("Enter the lower bound of the variable : "))
    ub = float(input("Enter the upper bound of the variable: "))

    return iterations, populationSize, hiddenUnits, mutationProbability, cross
overProbability, crossoverType, variableType, lb, ub

def crab():
    """
    This method loads the crab dataset and train a neural network model on the
 dataset
    """
    #Loading the crab dataset
```

```
    with open("crabData.csv") as file:
        data = []
        for line in file:
            data.append(line.strip().split(","))
        file.close()

    np.random.seed(5436)
    #Cleaning and splitting the data in training and testing data
    data = np.array(data)
    data = data[1:,:]
    np.random.shuffle(data)
    dataDivision = int(data.shape[0]*0.8)
    X = data[:dataDivision, 3:].astype(np.float)
    testX = data[dataDivision:, 3:].astype(np.float)
    #One hot encoding the labels
    Y = np.array(pd.get_dummies(data[:, 1]))
    trainingLabel = Y[:dataDivision,:]
    testingLabel = Y[dataDivision:,:]
    iterations, populationSize, hiddenUnits, mutationProbability, crossoverPro
bability, crossoverType, variableType, lb, ub = takeHyperParameters()
    #initialising and training the network
    network = NeuralNetwork(iterations = iterations, populationSize = populati
onSize, hiddenUnits = hiddenUnits, mutationProbability = mutationProbability,
crossoverProbability = crossoverProbability, crossoverType = crossoverType, va
riableType = variableType, lb =lb, ub = ub)
    network.fittingWithGA(X, trainingLabel)

    #evaluating the model on training and testing data
    network.evaluation(X,trainingLabel, "Traniing Accuracy")
    network.evaluation(testX, testingLabel, "Testing Accuracy")

def Iris():
    """
    This method loads the Iris dataset and train a neural network model on the
 dataset
    """
    #Loading the Iris dataset
    df = pd.read_csv('https://archive.ics.uci.edu/ml/'
            'machine-learning-databases/iris/iris.data', header=None)
    np.random.seed(100)
    permutaion = np.random.permutation(150)
    df = df.iloc[permutaion]
    #One hot encoding the labels
    Y = np.array(pd.get_dummies(df[4]))
    data = np.array(df)
    data = data[:,:4]
    #Cleaning and splitting the data in training and testing data
    dataDivision = int(data.shape[0]*0.8)
```

```
    X = data[:dataDivision,:].astype(np.float)
    trainingLabel = Y[:dataDivision,:]
    testingLabel = Y[dataDivision:,:]
    testX = data[dataDivision:, :].astype(np.float)
    iterations, populationSize, hiddenUnits, mutationProbability, crossoverPro
bability, crossoverType, variableType, lb, ub = takeHyperParameters()
    #initialising and training the network
    network = NeuralNetwork(iterations = iterations, populationSize = populati
onSize, hiddenUnits = hiddenUnits, mutationProbability = mutationProbability,
crossoverProbability = crossoverProbability, crossoverType = crossoverType, va
riableType = variableType, lb =lb, ub = ub)
    network.fittingWithGA(X, trainingLabel)

    #evaluating the model on training and testing data
    network.evaluation(X,trainingLabel, "Training Accuracy")
    network.evaluation(testX, testingLabel, "Testing Accuracy")

def main():
    """This method ask the user about the dataset on which you want to train y
our model and direct
    the program to that dataset"""
    print("\n\t\t\tWelcome")
    while True:
        print("\nChoose from the following on which dataset you want to train
your model")
        menu = input("1. Press 1 for Iris dataset(Classification between Setos
a, Versicular and Virginka)\n2. Press 2 for Crab dataset(Classification betwee
n Male and Female)\n3. Press Q/q to quit : ")
        if menu == "1":
            Iris()
        elif menu == "2":
            crab()
        elif menu == "Q" or menu == "q":
            break
        else:
            print("\nEnter Valid Entry\n")


if __name__ == "__main__":
    main()
```

Neural Network(Optimisation by PSO):

```
import numpy as np
from numpy.core.fromnumeric import argmax
import pandas as pd
import pyswarms as ps
import matplotlib.pyplot as plt
```

```
class NeuralNetwork:
    """
    Feedforward neural network / Multi-
layer perceptron classifier. Uses Particle swarm optimisation
    to optimise the error function

    Parameters:
        iterations : Number of Epochs
        hiddenUnits : Number of neurons in the hidden layer
        particles : Number of particles in the swarm
        c1 : Cognitive component coefficient
        c2 : Social component coefficient
        w : weight inertia
    """

    def __init__(self, iterations = 100, hiddenUnits = 32, particles =  1295,
c1 = 0.5, c2 = 0.7, w = 0.9, lb = -15, ub = 15):
        self.hiddenUnits = hiddenUnits
        self.iterations = iterations
        self.particles = particles
        self.c1 = c1
        self.c2 = c2
        self.w = w


    def sigmoid(self, v):
        """Compute the logistic function()sigmoid"""
        return 1 / (1 + np.exp(-np.clip(v, -250, 250)))

    def forward(self, X):
        """
        This method takes a datapoint, pass it through the forward path of neu
ral network and
        returns the activation scores of the output layer
        """
        netInputHidden = np.dot(self.weight_h.T, X) + self.bias_h
        netOutHidden = self.sigmoid(netInputHidden)
        netInputOuterLayer = np.dot(self.weight_o.T, netOutHidden) + self.bias
_o
        netOutOuterLayer = self.sigmoid(netInputOuterLayer)

        return netOutOuterLayer

    def errorFunction(self,X):
        """
        This method is the error function that we want to otimise through PSO
        """
```

```
        particles = []
        for i in range(X.shape[0]): #Iterating through every particle

            #Splitting falttened vector of parameters in respective sets of ma
trix(weights) and vector form(bias)
            w_hidden = X[i,:int(self.dataset.shape[1] * self.hiddenUnits)].cop
y().reshape((self.dataset.shape[1],self.hiddenUnits))
            b_hidden = X[i,int(self.dataset.shape[1] * self.hiddenUnits):int((
self.dataset.shape[1] * self.hiddenUnits) + self.hiddenUnits)].copy()
            w_out = X[i,int((self.dataset.shape[1] * self.hiddenUnits) + self.
hiddenUnits):int((self.dataset.shape[1] * self.hiddenUnits) + self.hiddenUnits
 + (self.hiddenUnits*self.label.shape[1]))].copy().reshape((self.hiddenUnits,s
elf.label.shape[1]))
            b_out = X[i,int((self.dataset.shape[1] * self.hiddenUnits) + self.
hiddenUnits + (self.hiddenUnits*self.label.shape[1])):].copy()

            #forward path to  get the activation score
            netInputHidden = np.dot(self.dataset, w_hidden) + b_hidden
            netOutHidden = self.sigmoid(netInputHidden)
            netInputOuterLayer = np.dot(netOutHidden, w_out) + b_out
            netOutOuterLayer = self.sigmoid(netInputOuterLayer)
            #mean squared error function
            e = np.sum(0.5*(self.label - netOutOuterLayer)**2)
            particles.append(e)

        return particles


    def fittingWithPSO(self, dataset, label, testX, testingLabel):
        """
        This method takes training and testing dataset, trains the model on th
e given dataset by
        using the PSO and plots the loss vs epoch curve. It also prints out th
e accuracies on the
        training as well as testing set
        """

        self.dataset = dataset
        self.label = label

        #Getting the dimensions to Flatten the matrix of weights and biases to
 a single  vector
        d = (dataset.shape[1] * self.hiddenUnits) + self.hiddenUnits + (self.h
iddenUnits * label.shape[1]) + self.label.shape[1]


        # passing the hyperparameters
        options = {'c1': self.c1, 'c2': self.c2, 'w': self.w}
```

```
        # Calling instance of PSO
        optimizer = ps.single.GlobalBestPSO(n_particles=self.particles, dimens
ions=d, options=options)
        cost, pos = optimizer.optimize(self.errorFunction, iters=self.iteratio
ns)

        #converting the optimised flattened vector back to matrix and vector f
orm(for weights and biases respectively)
        self.weight_h = pos[:int(self.dataset.shape[1] * self.hiddenUnits)].co
py().reshape((self.dataset.shape[1],self.hiddenUnits))
        self.bias_h = pos[int(self.dataset.shape[1] * self.hiddenUnits):int((s
elf.dataset.shape[1] * self.hiddenUnits) + self.hiddenUnits)].copy()
        self.weight_o = pos[int((self.dataset.shape[1] * self.hiddenUnits) + s
elf.hiddenUnits):int((self.dataset.shape[1] * self.hiddenUnits) + self.hiddenU
nits + (self.hiddenUnits*self.label.shape[1]))].copy().reshape((self.hiddenUni
ts,self.label.shape[1]))
        self.bias_o = pos[int((self.dataset.shape[1] * self.hiddenUnits) + sel
f.hiddenUnits + (self.hiddenUnits*self.label.shape[1])):].copy()

        #Plotting the Error vs epoch curve
        plt.plot(optimizer.cost_history)
        plt.xlabel("Epochs")
        plt.ylabel("Error Function")
        plt.title("c1="+str(self.c1)+", c2="+str(self.c2)+", w="+str(self.w)+"
, particles="+str(self.particles))
        plt.show()


        #Printing the accuracies of training and testing set
        self.evaluation(self.dataset,self.label, "Training Accuracy")
        self.evaluation(testX, testingLabel, "Testing Accuracy")


    def evaluation(self,dataset, label, accuracyType):
        """
        This method takes the dataset and runs it through the forward path of
the network, gets the
        activation score of the output layer, check it against the given label
 and give a prediction
        about which class it belongs to. By taking the account of true predict
ions, at the end it prints
        out the accuracy of the model on the dataset
        """

        tp = 0
        for data, d in zip(dataset, label):
            v = self.forward(data)
```

```
            if argmax(v) == argmax(d):
                tp += 1
        print(accuracyType +" : " + str(round(tp/label.shape[0] * 100, 2)) + "
%")

def takeHyperParameters():
    '''This function takes the hyperparameters from the user and return these
hyoer parameters'''

    print("\nEnter the following hyperparameters\n")
    hiddenUnits = int(input("Enter the number of hidden units : "))
    iterations = int(input("Enter the number of Epochs for which you want to t
rain the model : "))
    particles = int(input("Enter the number of particles : "))
    c1 = float(input("Enter the conginitive component coefficient c1 : "))
    c2 = float(input("Enter the social component coefficient c2 : "))
    w = float(input("Enter the Intertia weight w : "))


    return hiddenUnits, iterations, particles, c1, c2, w



def crab():
    """
    This method loads the crab dataset and train a neural network model on the
 dataset
    """
    #Loading the crab dataset
    with open("crabData.csv") as file:
        data = []
        for line in file:
            data.append(line.strip().split(","))
        file.close()

    np.random.seed(5436)
    #Cleaning and splitting the data in training and testing data
    data = np.array(data)
    data = data[1:,:]
    np.random.shuffle(data)
    dataDivisior = int(data.shape[0]*0.8)
    X = data[:dataDivisior, 3:].astype(np.float)
    testX = data[dataDivisior:, 3:].astype(np.float)
    #One hot encoding the labels
    Y = np.array(pd.get_dummies(data[:, 1]))
    trainingLabel = Y[:dataDivisior,:]
    testingLabel = Y[dataDivisior:,:]

    hiddenUnits, iterations, particles, c1, c2, w = takeHyperParameters()
```

```
    #initialising and training the network
    network = NeuralNetwork(hiddenUnits = hiddenUnits, iterations = iterations
, particles = particles, c1 = c1, c2 = c2, w = w)
    network.fittingWithPSO(X, trainingLabel, testX, testingLabel)

def Iris():
    """
    This method loads the Iris dataset and train a neural network model on the
 dataset
    """
    #Loading the Iris dataset
    df = pd.read_csv('https://archive.ics.uci.edu/ml/'
            'machine-learning-databases/iris/iris.data', header=None)
    np.random.seed(100)
    #Cleaning and splitting the data in training and testing data
    permutaion = np.random.permutation(150)
    df = df.iloc[permutaion]
    #One hot encoding the labels
    Y = np.array(pd.get_dummies(df[4]))
    data = np.array(df)
    data = data[:,:4]
    #Cleaning and splitting the data in training and testing data
    dataDivision = int(data.shape[0]*0.8)
    X = data[:dataDivision,:].astype(np.float)
    trainingLabel = Y[:dataDivision,:]
    testingLabel = Y[dataDivision:,:]
    testX = data[dataDivision:, :].astype(np.float)
    hiddenUnits, iterations, particles, c1, c2, w = takeHyperParameters()
    #initialising and training the network
    network = NeuralNetwork(hiddenUnits = hiddenUnits, iterations = iterations
, particles = particles, c1 = c1, c2 = c2, w = w)
    network.fittingWithPSO(X, trainingLabel, testX, testingLabel)

def main():
    """This method ask the user about the dataset on which you want to train y
our model and direct
    the program to that dataset"""

    print("\n\t\t\tWelcome")
    while True:
        print("\nChoose from the following on which dataset you want to train
your model")
        menu = input("1. Press 1 for Iris dataset(Classification between Setos
a, Versicular and Virginka)\n2. Press 2 for Crab dataset(Classification betwee
n Male and Female)\n3. Press Q/q to quit : ")
        if menu == "1":
            Iris()
        elif menu == "2":
```

```
            crab()
        elif menu == "Q" or menu == "q":
            break
        else:
            print("\nEnter Valid Entry\n")


if __name__ == "__main__":
    main()
```