Task 1: Perceptron Model for Binary Classes Classification

We have implemented perceptron model for binary classes classification. This perceptron takes 2D points as input and classifies it as belonging to either class 0 or 1. Perceptron model build in these steps:

1. Parameters Initialization
2. Model Training:
   1. Multiplication of weights and input data
   2. Weight updating (gradient descent)

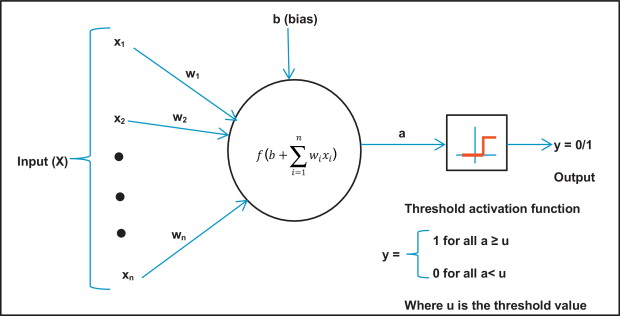


Figure: Perceptron model (google images)

## Parameters Initialization:

In perceptron model weight parameters multiply with data to get some output result, the size of weights depends on input data features. At start we have initialized using random variables.

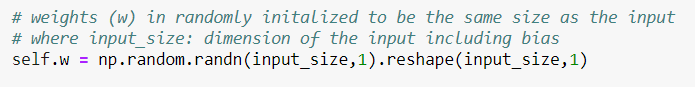


Figure: Random Weight initilization (code)

## Model Training:

We perform matrix row and weight multiplication in iterative loop, apply perceptron condition on the output and update the weights using gradient descent steps. The model use Heaviside step function as activation function and make value 1 or 0 depends on condition.

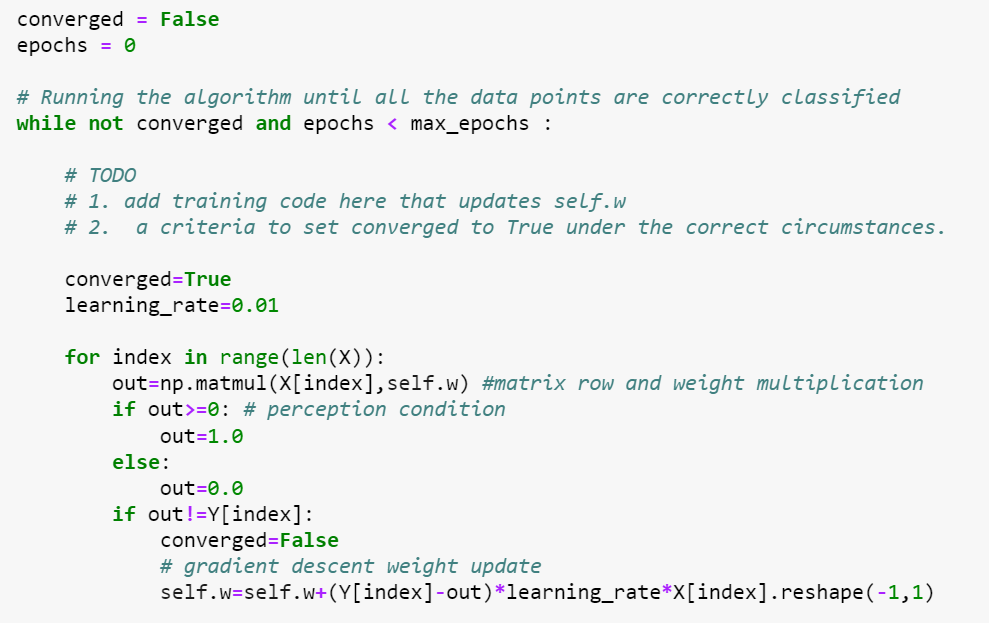


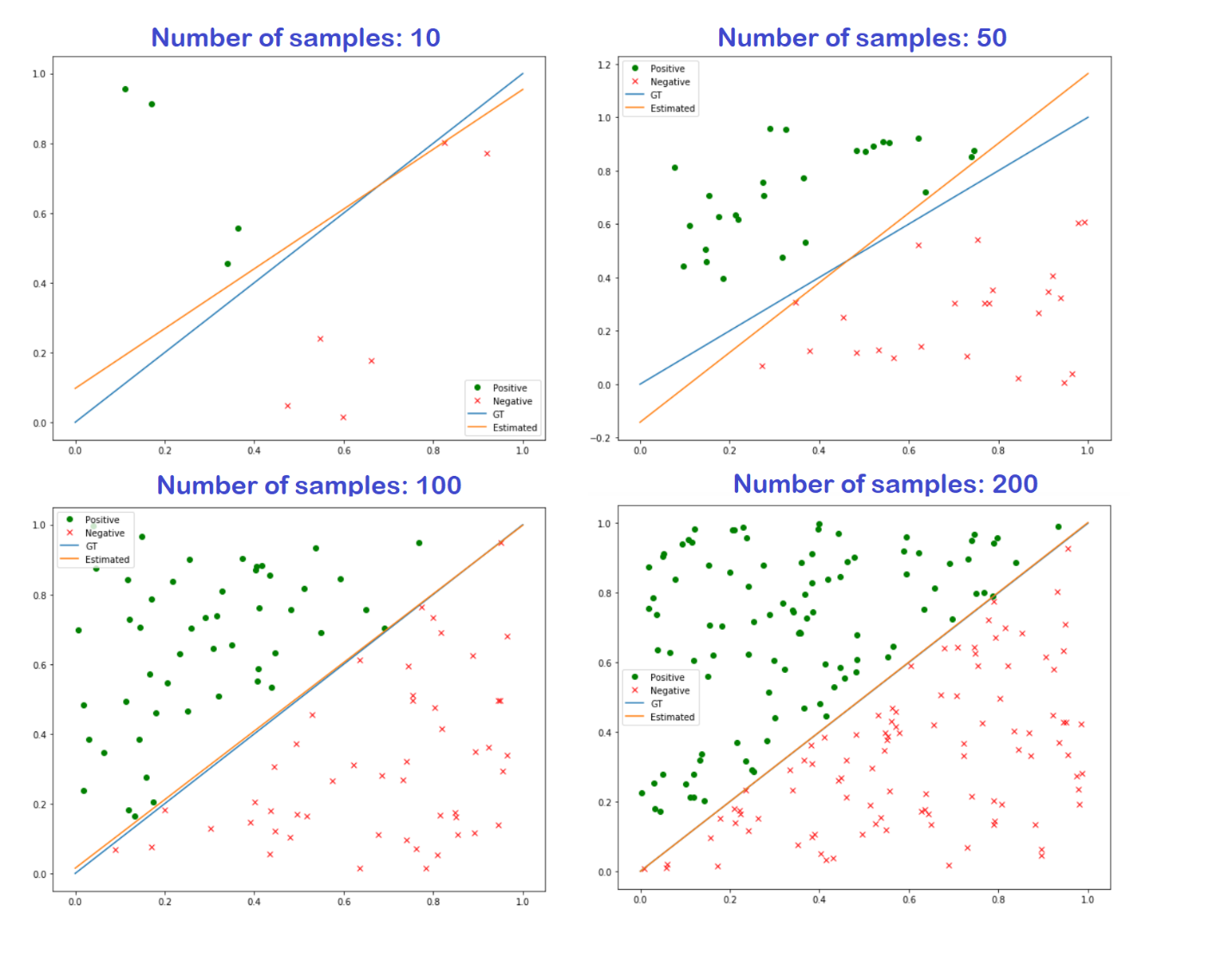
Figure: Perceptron model Training (code)

## Estimated line can correctly classify might not agree with the ground-truth boundary:

The reason behind it free space between 2 classes points, as in our algorithm we have set condition that when model converged is True then training will end. Below picture it can be seen that model converged but doesn’t meet the ground truth line if space between points is more.

Behavior depend on the number of samples:

Yes, it depends on number of samples, can be observed in the below image. As samples increase the points are coming close to each other and less space available to converge.

Figure: Estimated line and Ground Truth comparison (this figure made using changing the number of samples in the code start)

## Sometimes boundary oscillating between two solutions and not reaching 100% accuracy:

The reason of this is gradient descent step size, that defined by learning rate. When the learning rate is big gradient descent take bigger step and fluctuate between positive and negative side. And can’t able to fully converge to solution and full accuracy.

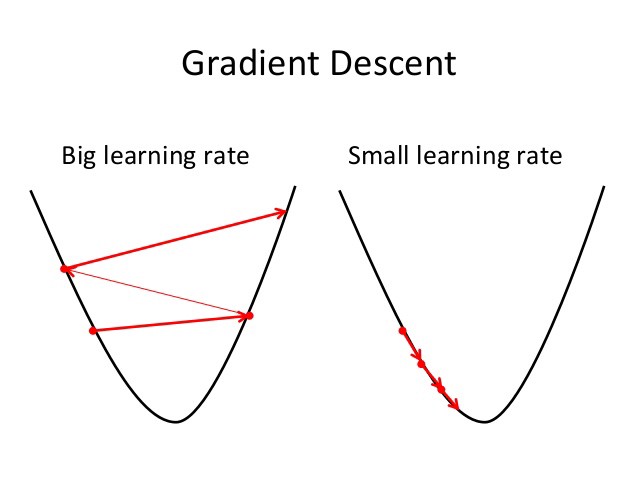


Figure: Gradient Descent steps (google images)

One example of this behavior shown below where our estimated line fluctuates between positive and negative side.

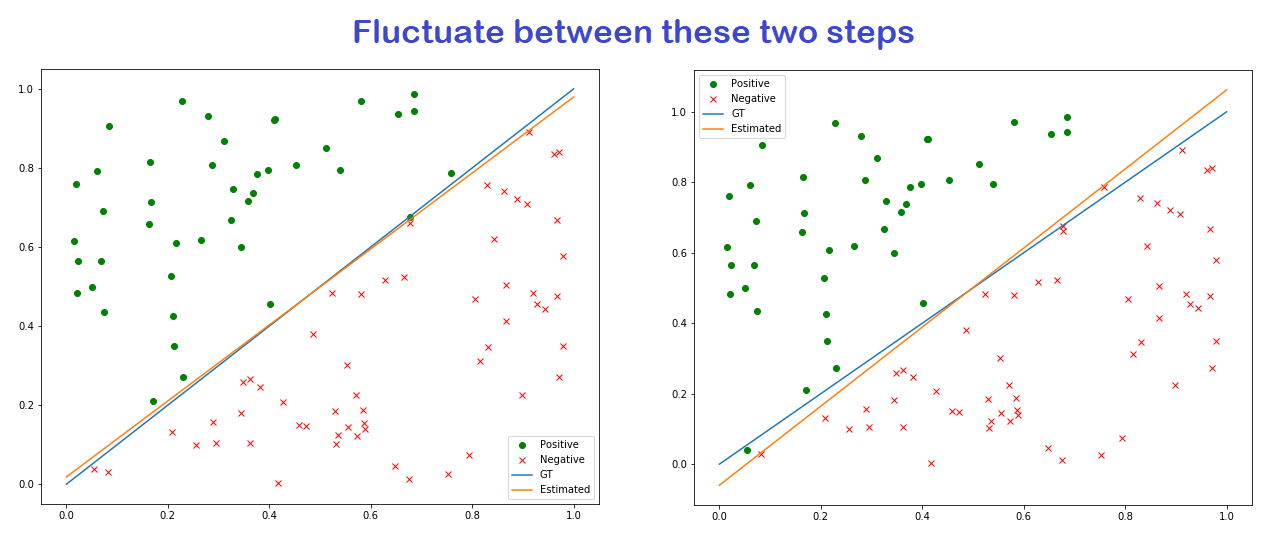


Figure: Big Learning rate problem model not converge, it fluctuate between these two state( code: generated by changing the code learning rate to bigger amount)

We change the learning\_rate=0.01 of training algorithm to learning\_rate=0.001 to make it modified algorithm.

## Number of epochs required to converge in for both the original and the modified algorithm:

Random initialization causes the algorithm to converge in different number of epochs. That’s why we take different number of measurement and showing the mean result of convergence. As a general if there are less number of point then Training algorithm better if the points increase then Modified algorithm better. As a general modified algorithm is better as it will may take more steps but model will converge. If the number of sample increase more than it we may need to choose smaller learning rate to converge.

|  |  |  |
| --- | --- | --- |
| Number of Epochs Required | | |
| Sample Size | Training Algorithm (lr=0.01) | Modified Algorithm (lr=0.001) |
| 10 | 33 | 296 |
| 100 | 28 | 246 |
| 1000 | 423 | 27 |
| 10000 | Never Converged | 20 |

Task 2: Perceptron Model for 3-Classes Classification

We implemented perceptron model on 3 classes classification, the dataset points plot shown below, where different colors represent different classes.

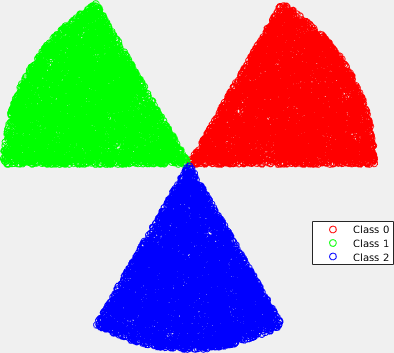
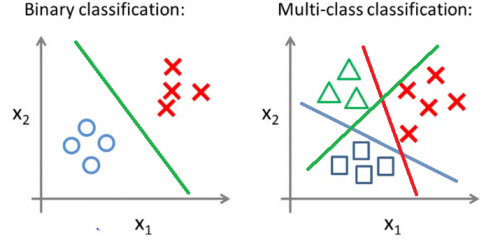


Figure 1: Dataset containing three classes

We chose one vs All classification approach to solve this problem as in perceptron model can distinguish between 2 classes at one time, so we have set one class true and other 2 classes false at given instant. The difference between binary classification and one vs all shown below:



By this method we have 3 different cases appear that we have trained 3 models weight for this. The cases are shown below:

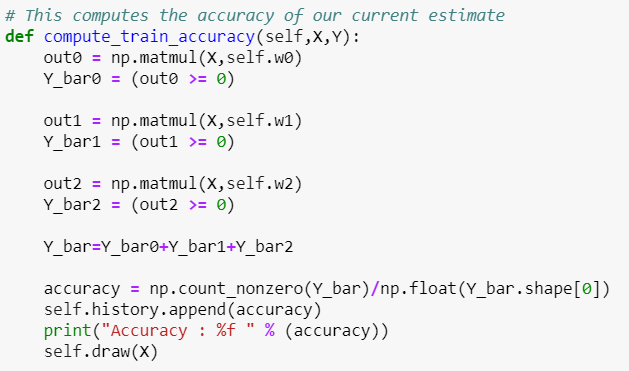
|  |  |  |
| --- | --- | --- |
| Cases | True | False |
| 1 | Class 0 | Class 1, Class 2 |
| 2 | Class 1 | Class 0, Class 3 |
| 2 | Class 2 | Class 0, Class 1 |

In this scenario I have turned the false both example values to zero and true class values to 1 and trained it as positive side is true for specific case, the below code shown the implementation of 3 cases.



## Modified Accuracy over time:

We have modified the accuracy function, now we are measuring the accuracy of each class predicted right value, then sum these values and divide over total number of values.



The accuracy plot is shown below:

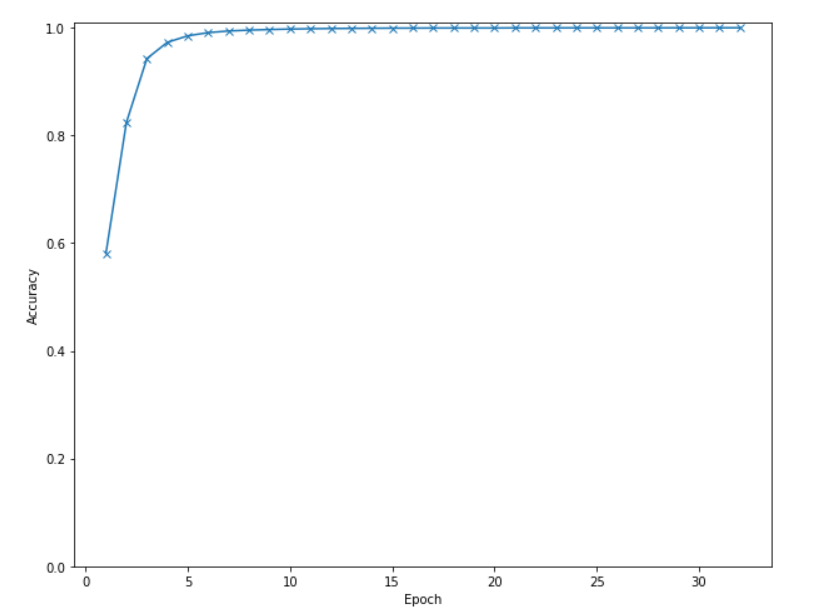
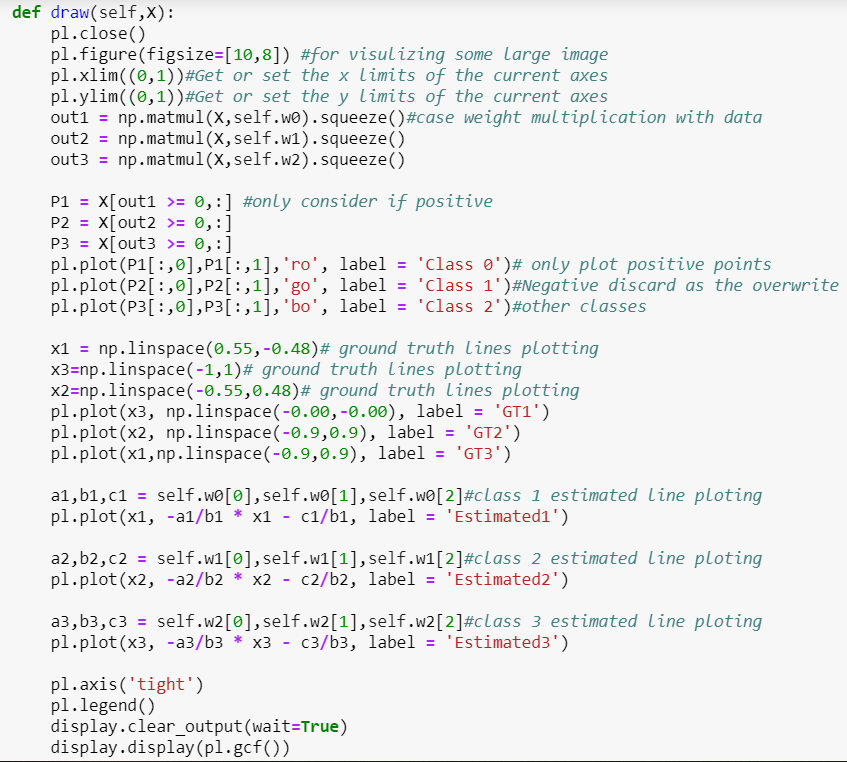


Figure: Accuracy of model (code)

## Visualize the decision boundaries:

The modified draw function implementation shown below, we are only showing positive side of class as negative side over write the other class color.



The plots on different epoch steps shown below:

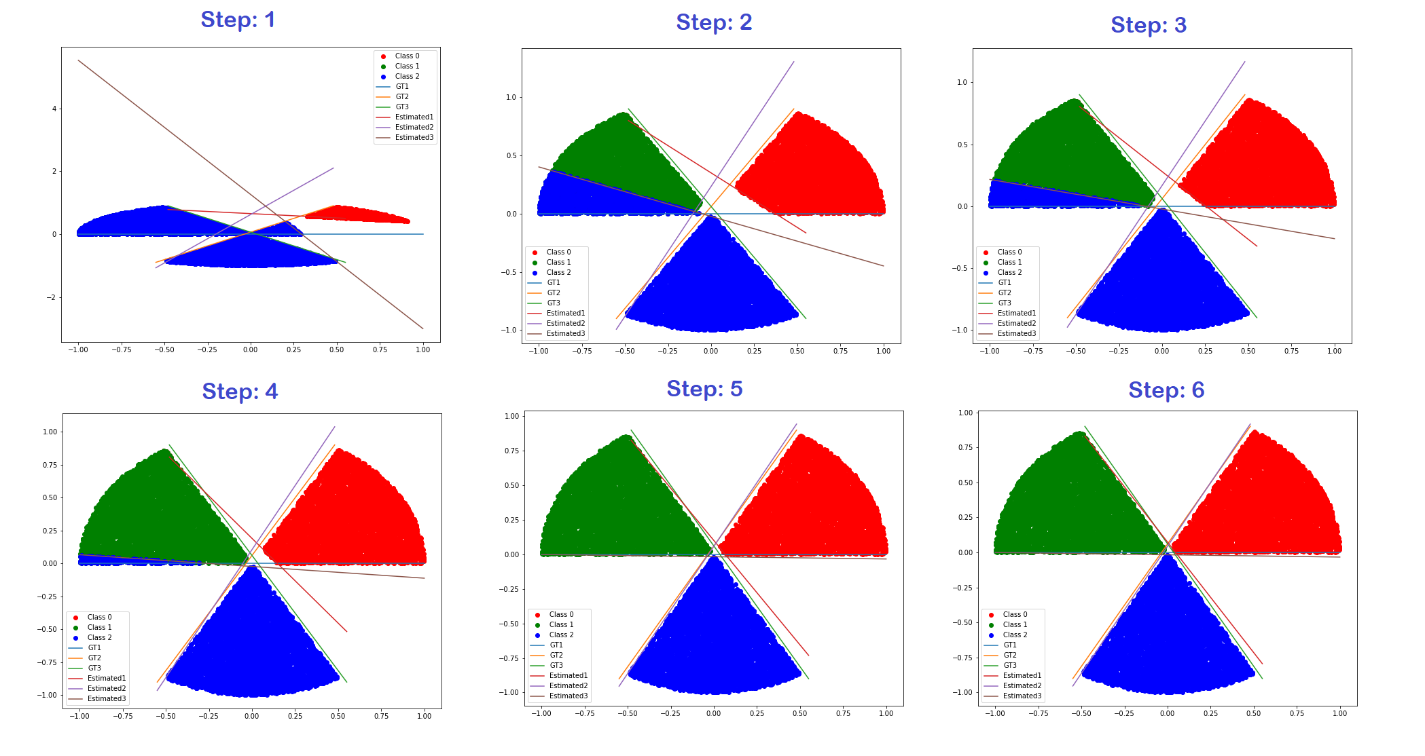


Figure: Steps of convergence (code)

### Linear classifier is still a good choice for this dataset:

As these classes can be separated by each other with linear classifier based on line. Linear classifier based on 1st degree so they don’t have any curve or bend in plotting, as these classes can be separated by line from each other that why linear classifier is good choice and we can achieve maximum accuracy with it.

Task 3: 3-Layer Neural Network like XOR Gate

We have made a 3-Layer Neural Network to classify the non-linearly separable outputs like XOR gate. Neural Network build in these steps:

1. Define the neural network structure ( # of input units, # of hidden units, etc).
2. Initialize the model's parameters
3. Loop:
   1. Implement forward propagation
   2. Compute loss
   3. Implement backward propagation to get the gradients
   4. Update parameters (gradient descent)

## Neural Network Structure:

We have built 3-layers Neural Network

* Input Layer (Taking two input A, B and a bias): This has a size of 3
* Hidden Layer (different number of neurons in this layer)
* Output Layer: This has a size of 1 as we are generating a single number for every input

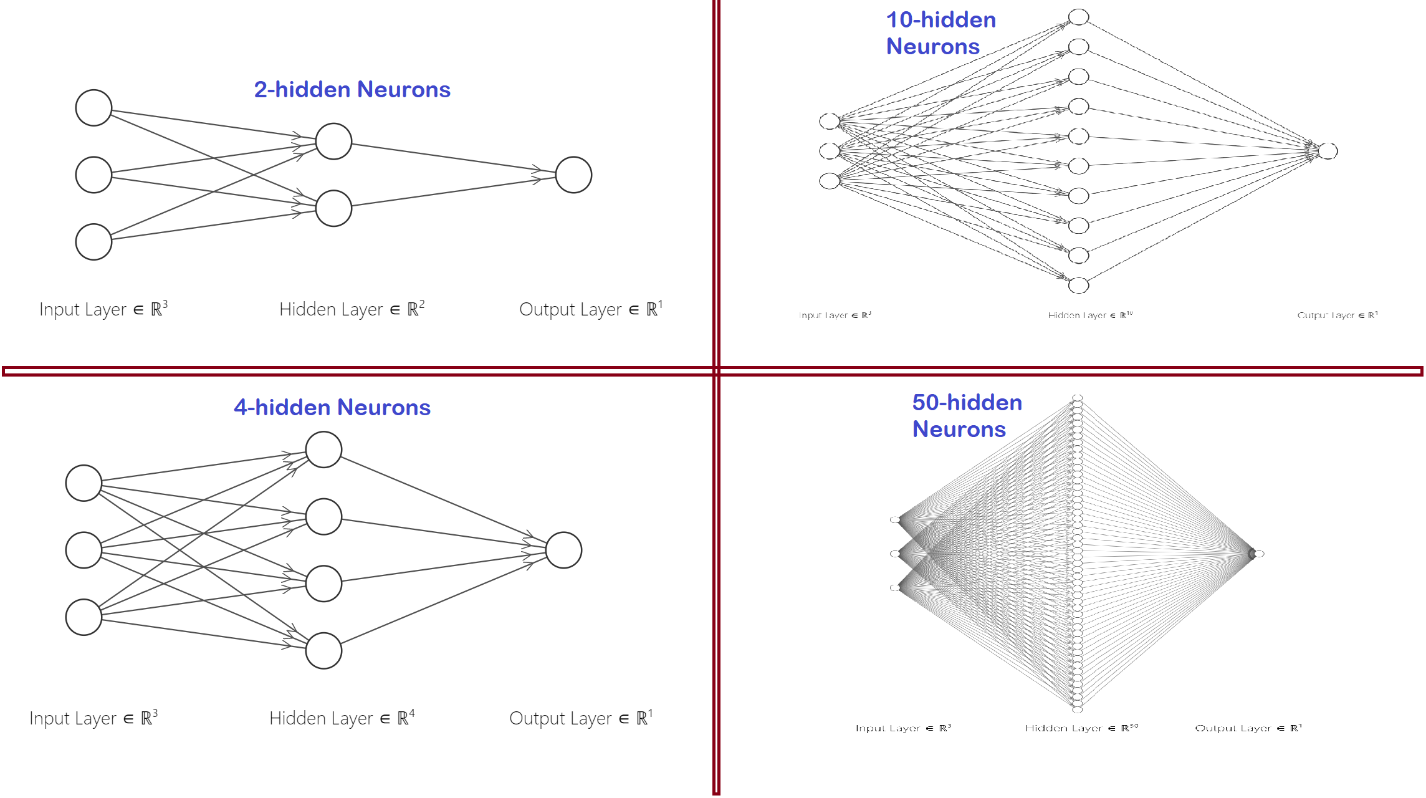


Figure: Example of different hidden layers neurons (<http://alexlenail.me/NN-SVG/index.html>)

## Initialize the model's parameters:

In our model there 2 weights parameters one for hidden and one for output layer calculation, the size of weights depends on input data features and number of neurons in hidden layer. At start we have initialized using random variables.

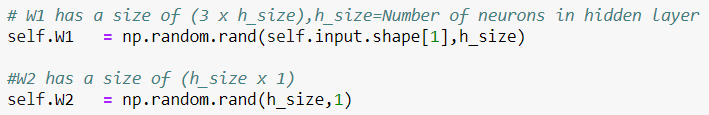
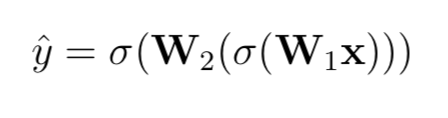


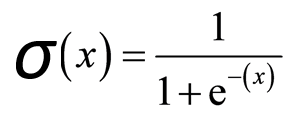
Figure: Random Initialization of weights(code)

## Forward Propagation:

We have performed forward propagation to multiply the weights with our data at both layers and applied non linearity activation function sigmoid. The forward function takes the input x and does the following operations:



Where sigmoid is defined as:



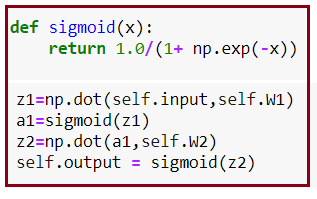
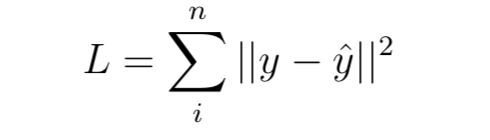


Figure: Forward Propagation(code)

## Compute loss:

We have used squared error as the loss which is defined by



where yˆ is our estimated output and y is the ground truth value. The loss function graph respect to number of epochs are shown below. One can notice that different Neurons numbers in hidden layer takes different time to converge to lowest loss.

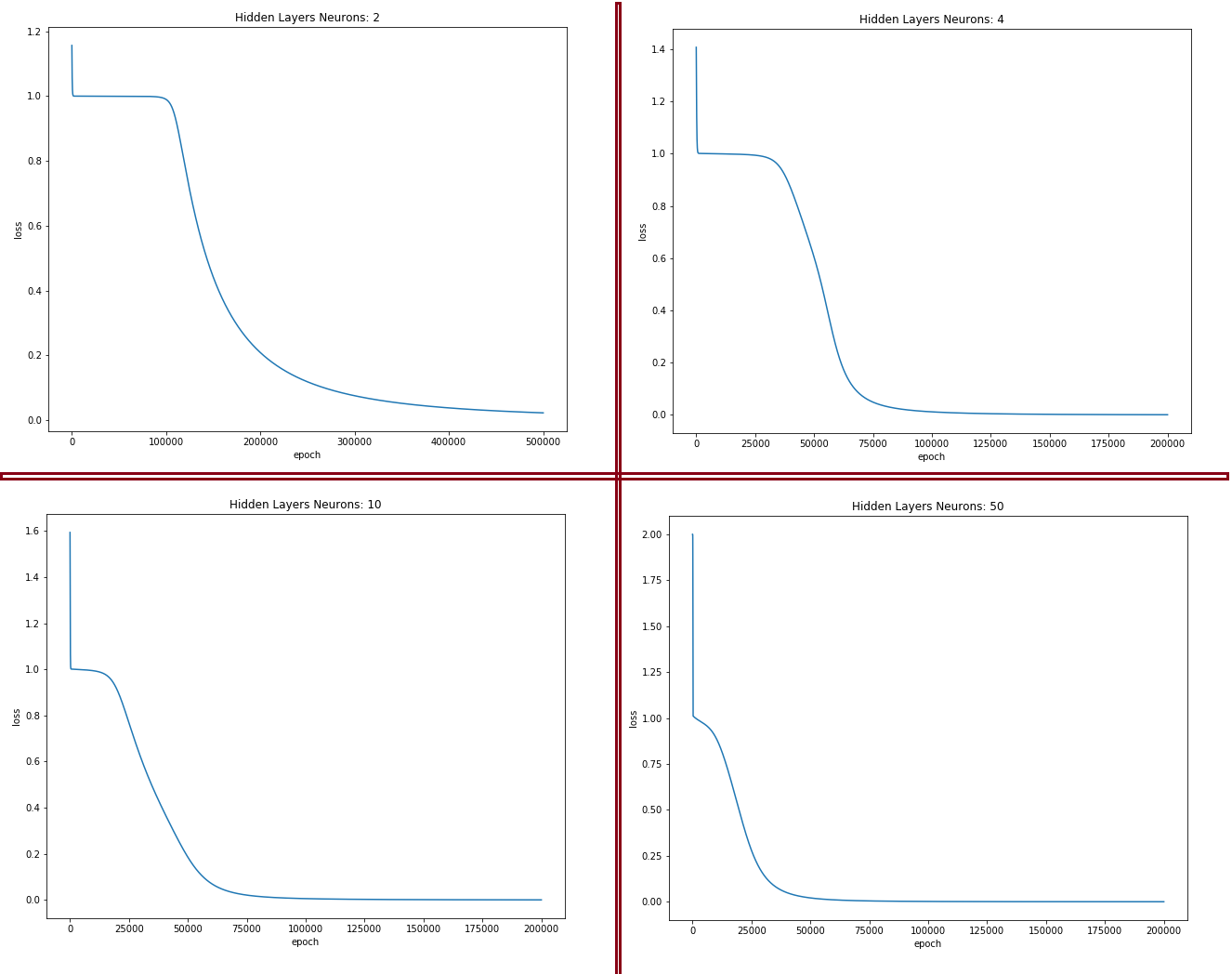
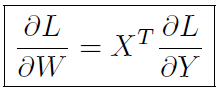


Figure: Different Hidden Layers Neuron Number Loss/epochs plot(code)

## Backward Propagation:

We performed backward propagation to update the weights parameters using gradient descent method. We chose chain rule to find the derivative of loss function.



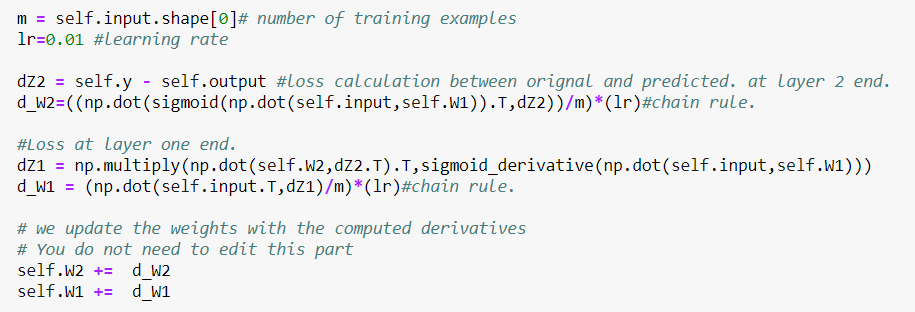


Figure: Backward Propagation (code)

## Performance of the network with different numbers of hidden neuron:

As the number of neurons increase the model try to over fit on the solution, its loss decrease with respect to less number of neurons, model converge fast. But as the number of neurons increase the time and computation resources also increased. Check the below tables.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Loss vs Number of Neurons and Iteratons | | | | |
| Iterations | Neurons: 2 | Neurons: 4 | Neurons: 10 | Neurons: 50 |
| 1 | 1.11761625 | 1.1925961 | 1.7509841 | 1.99999647 |
| 1000 | 1.00343358 | 1.00067692 | 1.0027286 | 1.00808088 |
| 10000 | 0.99905261 | 0.9646847 | 0.98702245 | 0.90947934 |
| 50000 | 0.77851156 | 0.05089196 | 0.06889183 | 0.01881463 |
| 100000 | 0.32794832 | 0.00618477 | 0.00262134 | 0.00149503 |
| 150000 | 0.16730678 | 0.00239602 | 0.00071091 | 0.00041009 |
| 200000 | 0.10049194 | 0.00129817 | 0.00032218 | 0.00017822 |

|  |  |
| --- | --- |
| Time for 2,00,000 iterations | |
| Number of Neuorns | Time (sec) |
| Neurons: 2 | 18.6 |
| Neurons: 4 | 18.2 |
| Neurons: 10 | 19.7 |
| Neurons: 50 | 22.2 |

## Minimum number of neurons needed in the hidden layer for successful training:

Minimum 1 neurons needed for succesful training, the all the input will attach on it. You can see one example of it below where 4 input neurons, 1 hidden and 1 output neurons.

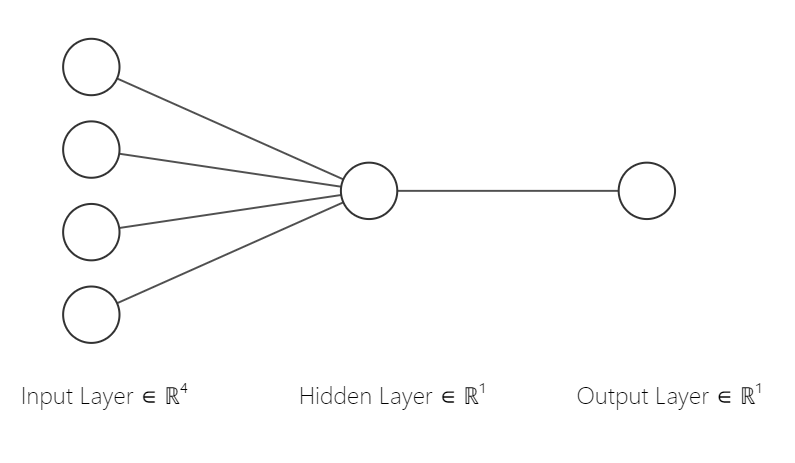


Figure: 1 hidden Neuron(<http://alexlenail.me/NN-SVG/index.html>)

## When there are too many neurons in the hidden layer:

1. Computation Resources increase
2. More time for training
3. Model Converge fast on training data.
4. Learning model from under fit to over fit.

There are some empirically-derived rules-of-thumb for chosing hidden neurons number, of these, the most commonly relied on is 'the optimal size of the hidden layer is usually between the size of the input and size of the output layers'. Jeff Heaton