Neural Network Implementation from Scratch

Rohan Seth (927001590)& Atreye Ghosh(827009108)
`rohanseth@tamu.edu, aghosh32@tamu.edu`

Texas A&M University — December 11, 2018

Introduction

An artificial neural network is a model of computation inspired by the structure of neural networks in the brain. In simplified models of the brain, it consists of a large number of basic computing device (neurons) that are connected to each other in a complex communication network, through which the brain is able to carry out highly complex computations. Artificial neural networks are formal computation constructs that are modeled after this computation paradigm. A neural network can be described as a directed graph whose nodes correspond to neurons and edges correspond to links between them. Each neuron receives as input a weighted sum of the outputs of the neurons connected to its incoming edges. We focus on feedforward networks in which the underlying graph does not contain cycles. In the context of learning, we can define a hypothesis class consisting of neural network predictors, where all the hypotheses share the underlying graph structure of the network and differ in the weights over edges. As we shall see, neural network are very expressive: they can approximate almost any functions assuming that the size of the network is sufficiently large. The caveat is that the problem of training such hypothesis classes of neural network predictors is computationally hard. A widely used heuristic for training neural networks relies on a stochastic version of the subgradient descent framework that we studied before. There, we have shown that subgradient descent is a successful learner if the loss function is convex. In neural networks, the loss function is highly nonconvex. Nevertheless, we can still implement the subgradient descent algorithm and hope that it will find a reasonable solution (as happens to be the case in many practical tasks). In particular, the most complicated operation is the calculation of the gradient of the loss function with respect to the parameter of the network. We present backpropagation algorithm that efficiently calculates the gradient.

Info: Neural Networks consist of the following components

- One entry in the list
- Another entry in the list
- An input layer, x
- An arbitrary amount of hidden layers
- An output layer, $\hat{y}$
- A set of weights and biases between each layer, W and b
- A choice of activation function for each hidden layer. In this implementation we are using a Sigmoid activation function.

1 Problem Statement

Implementation of **two layer Neural Network from Scratch** without predefined library functions. Also, to conduct a accuracy comparison with other algorithms like logistic regression.

- We train our data on 70% of the IRIS sampleset and use the rest 30% as our test data.

- We play around with hidden layer size, combination of features, Number of trainings, activation function
- We plot the separation boundary with three variables
- Conduct a quantatitive comparison with Logistic Regression.

2 Implementation

We describe the key sets of our Implementation of single Hidden Layer Neural Network.

2.1 Training the Neural Network

Naturally, the right values for the weights and biases determines the strength of the predictions. The process of fine-tuning the weights and biases from the input data is known as training the Neural Network. Each iteration of the training process consists of the following steps:

(a) Calculating the predicted output ŷ, known as feedforward

(b) Updating the weights and biases, known as backpropagation

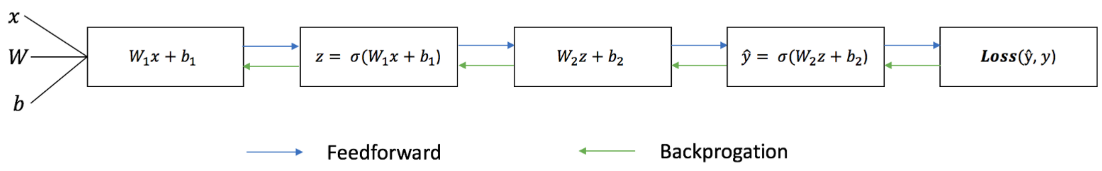


Figure 1: Impact of Hidden Layer Size on our Neural Network

Command Line
```
W1 += -learn_rate * dW1
b1 += -learn_rate * db1
W2 += -learn_rate * dW2
b2 += -learn_rate * db2
```

2.2 Feed Forward

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

Figure 2: FeedForward on our Neural Network

Command Line
```
z1 = X.dot(W1) + b1
a1 = np.tanh(z1)
z2 = a1.dot(W2) + b2
exp_scores = np.exp(z2)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
```

2.3 Backward Propogation

$$Loss(y,\hat{y}) = \sum_{i=1}^{n}(y - \hat{y})^2$$

$$\frac{\partial\, Loss(y,\hat{y})}{\partial W} = \frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \quad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * \text{z(1-z)} * x$$

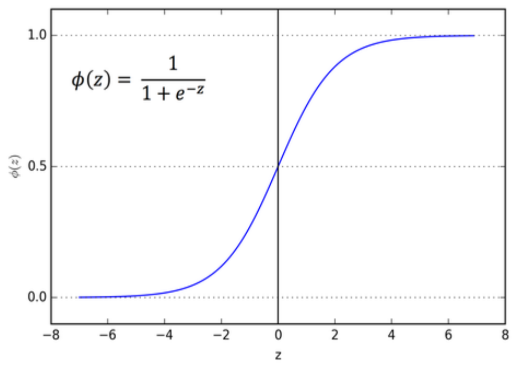
Figure 3: Chain rule for calculating derivative of the loss function with respect to the weights. Note that for simplicity, we have only displayed the partial derivative assuming a 1-layer Neural Network.

Command Line

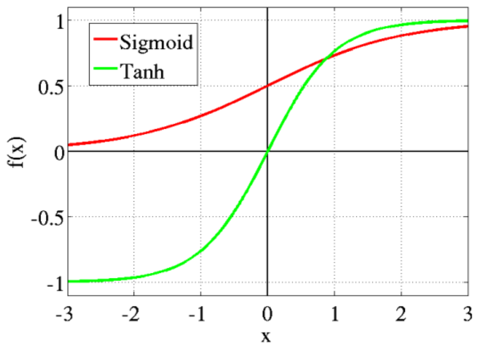
```
delta3 = probs
ab=train_size
delta3[range(train_size), (y)] -= 1
dW2 = (a1.T).dot(delta3)
db2 = np.sum(delta3, axis=0, keepdims=True)
delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
dW1 = np.dot(X.T, delta2)
db1 = np.sum(delta2, axis=0)
```

2.4 Activation Function

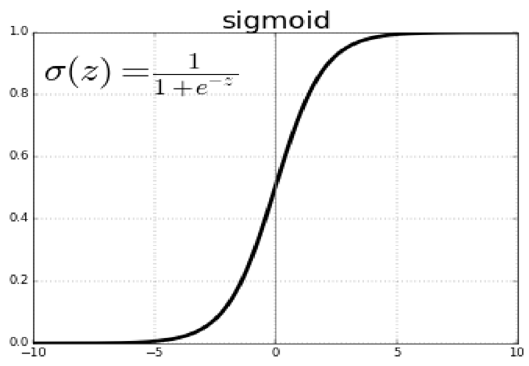
We use here Sigmoid as our first activation function and Tanh as second. We also run our simulations with Relu function to see almost similar response.

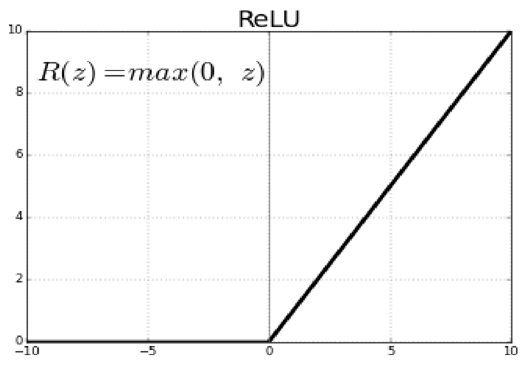


(a) Sigmoid



(b) tanh





Command Line

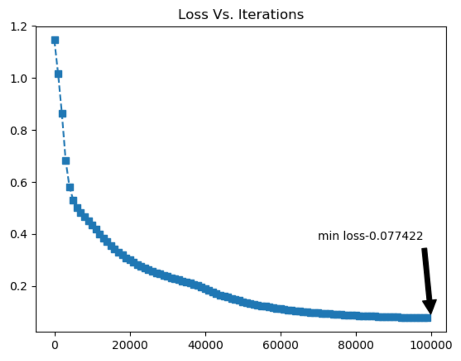
```
def cal_loss(model):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # Prediction calculation for Forward propagation
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    # Loss calculation
    corect_logprobs = -np.log(probs[range(train_size), y])
    data_loss = np.sum(corect_logprobs)
    # Addding regulatization term to loss
    data_loss += regl_l/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    return 1./train_size * data_loss
```

3 Decision Making

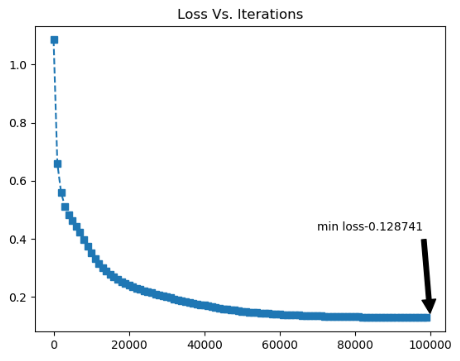
We demonstrate below the salients points we tune and train our neural network over, towards achieving our results.

3.1 Feature Set Selection

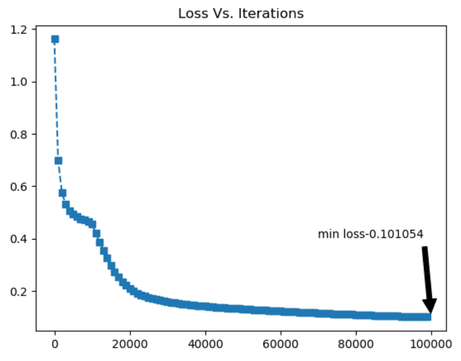
We have four features for our dataset, and we run our model across all possible combinations of pairs of features for selection and observe the Loss per iteration. Based on the graphs obtained below we proceed with the Petal Length and Petal Width as our chosen Features for Selection.



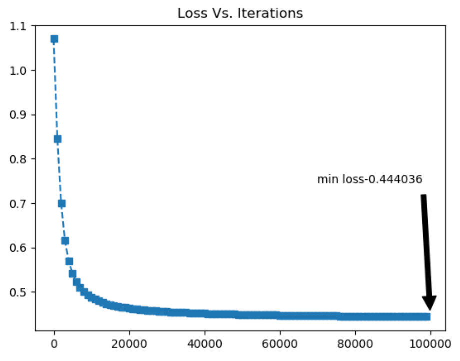
(c) Petal Length & Petal Width



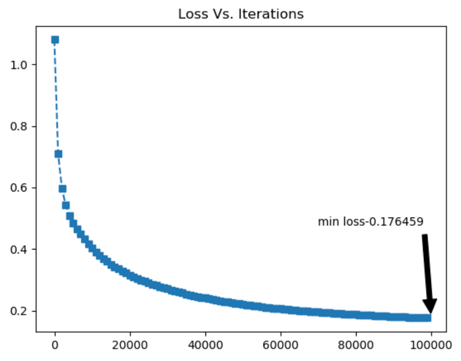
(d) Petal Length & Sepal Width



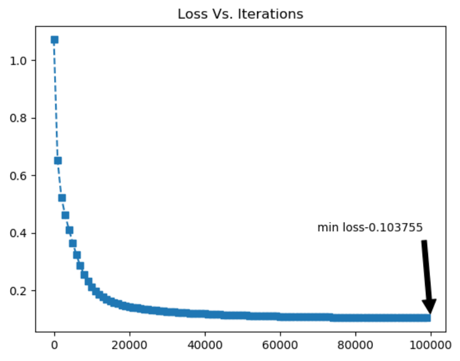
(e) Petal Length & Sepal Length



(f) Sepal Length & Sepal Width



(g) Sepal Length & Petal Width



(h) Sepal Length & Petal Width

Figure 4: Loss/Iterations for Different pairs of Features.

Info:

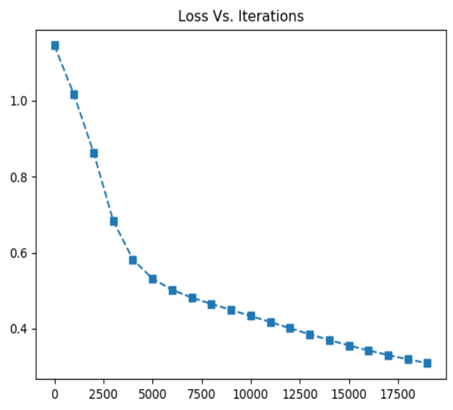
Thus of this excercise, we decide on the best features for our Neural Network that we would be proceeding with. But, this is with the following inherent assumptions

(a) Hidden Layer Size has been kept constant across the comparison

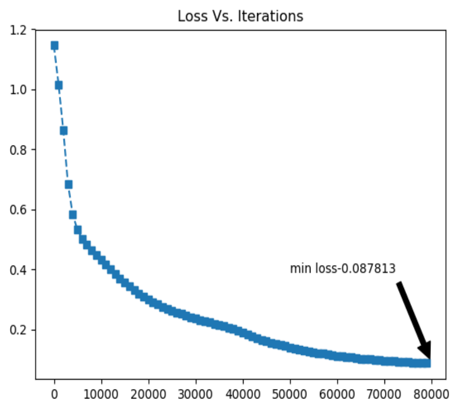
(b) Number of iterations for training has been kept constant.

3.2 Training Set

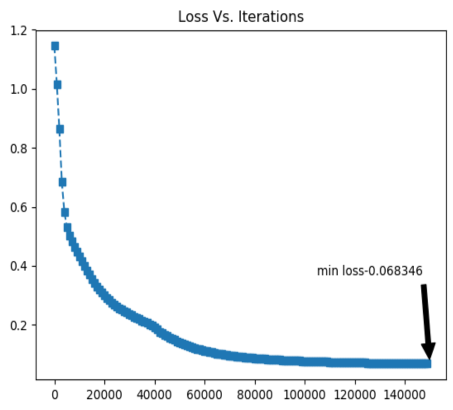
We train our neural network with different training iterations to observe the optimal size of the Training set. We observe this to be a tradeoff of Time vs. Loss.



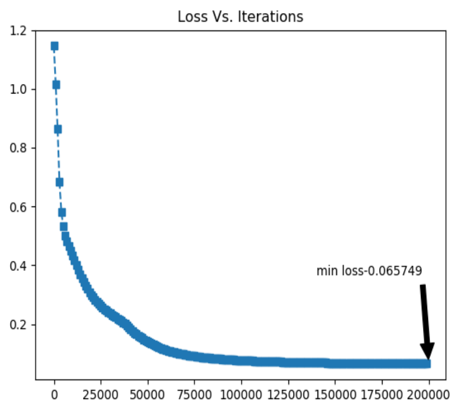
(a) 20K Iternations



(b) 80K Iterations



(c) 150K Iterations



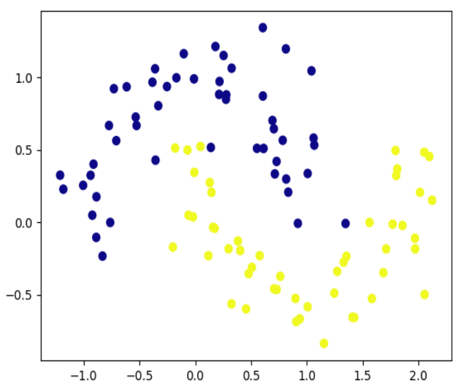
(d) 200K Iterations

Figure 5: Loss/Iterations for different number of Trainings Conducted.

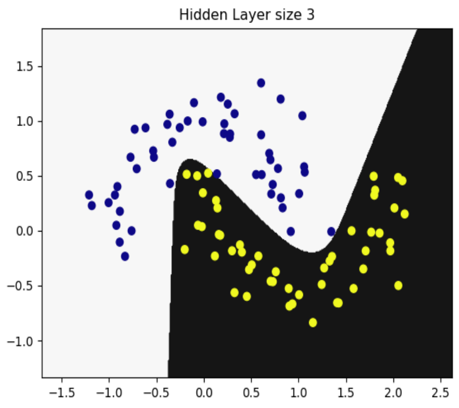
Info: We observe that the Loss saturates after 150k iterations, and there is a tradeoff between running time and the number of iterations we perform for training. Thus going forward we choose 150K iterations to train our network.

3.3 Hidden Layer Size

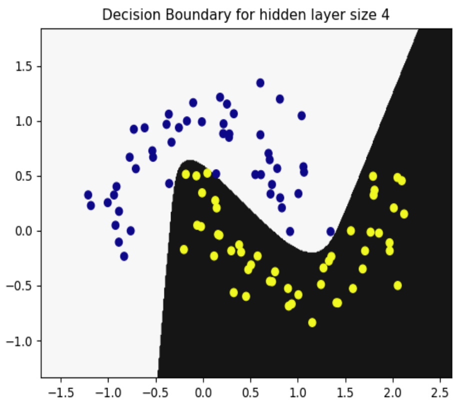
We play around with Hidden Layer size of the neural network. The underneath plots demonstrate the effect on the dataset.



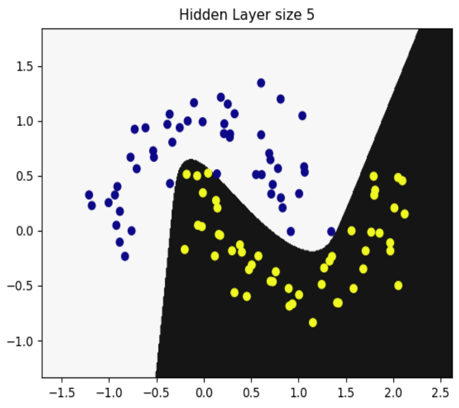
(a) Initial Data



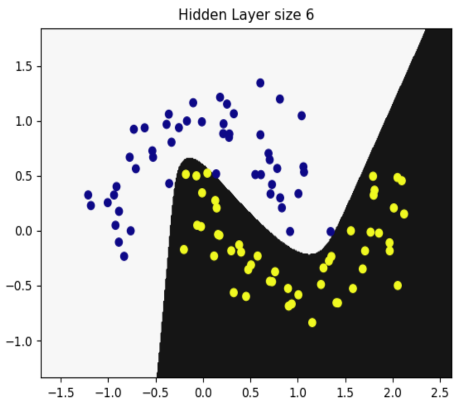
(b) Hidden Layer Size - 3



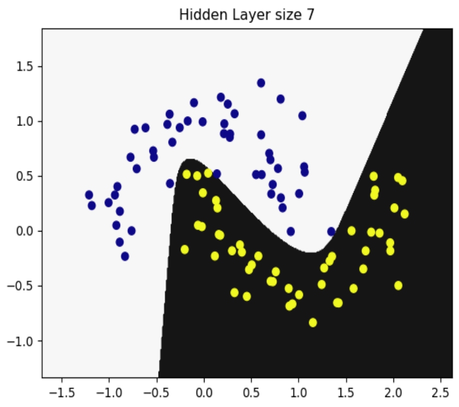
(c) Hidden Layer Size - 4



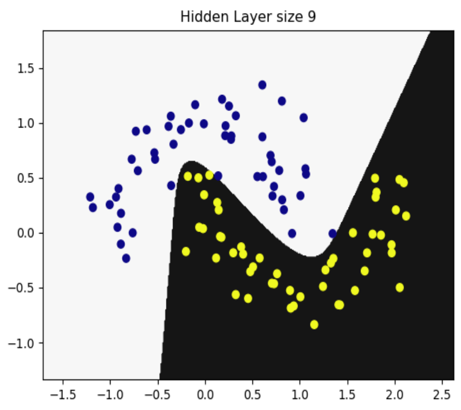
(d) Hidden Layer Size - 5



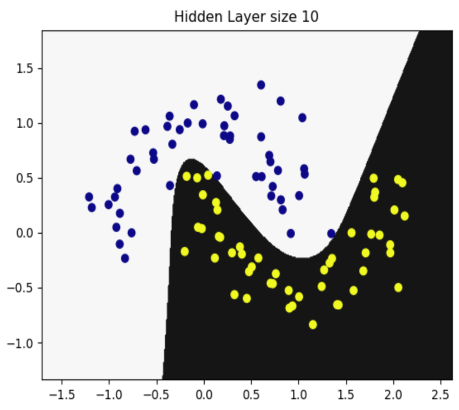
(e) Hidden Layer Size - 6



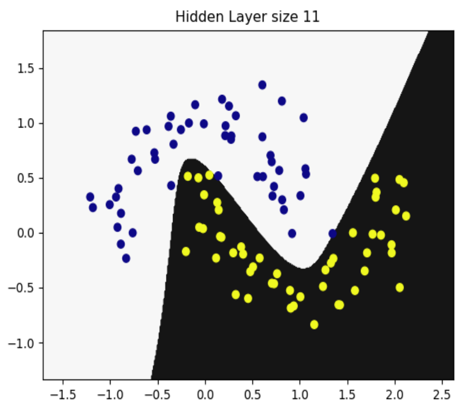
(f) Hidden Layer Size - 7



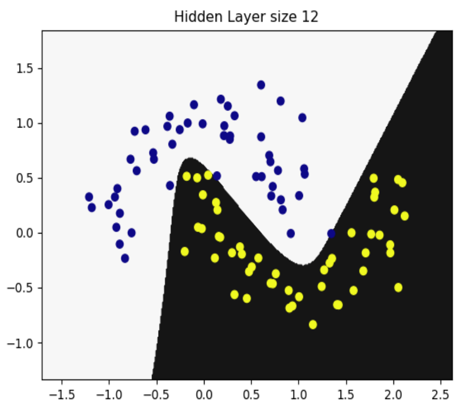
(g) Hidden Layer Size - 9



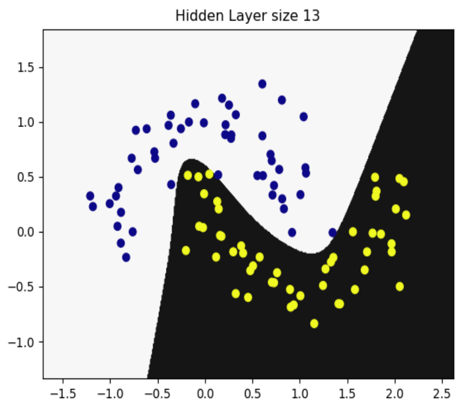
(h) Hidden Layer Size - 10



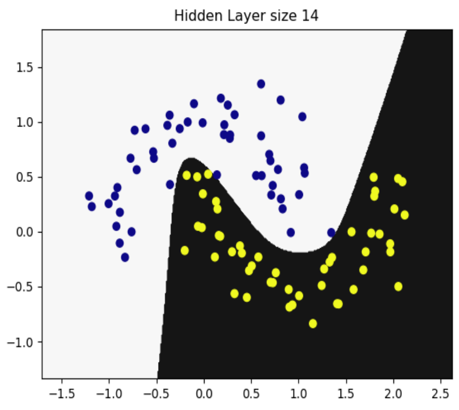
(i) Hidden Layer Size - 11



(j) Hidden Layer Size - 12



(k) Hidden Layer Size - 13



(l) Hidden Layer Size -14

Figure 6: Impact of Hidden Layer Size on our Neural Network

Info: We observe that the size of 6 is sufficiently good and then the curve overfits which doesn't bring any advantage here.

3.4 Impact of Initialization

We play around with initialization of the weight vectors to observe the rate of convergence as demonstrated below.

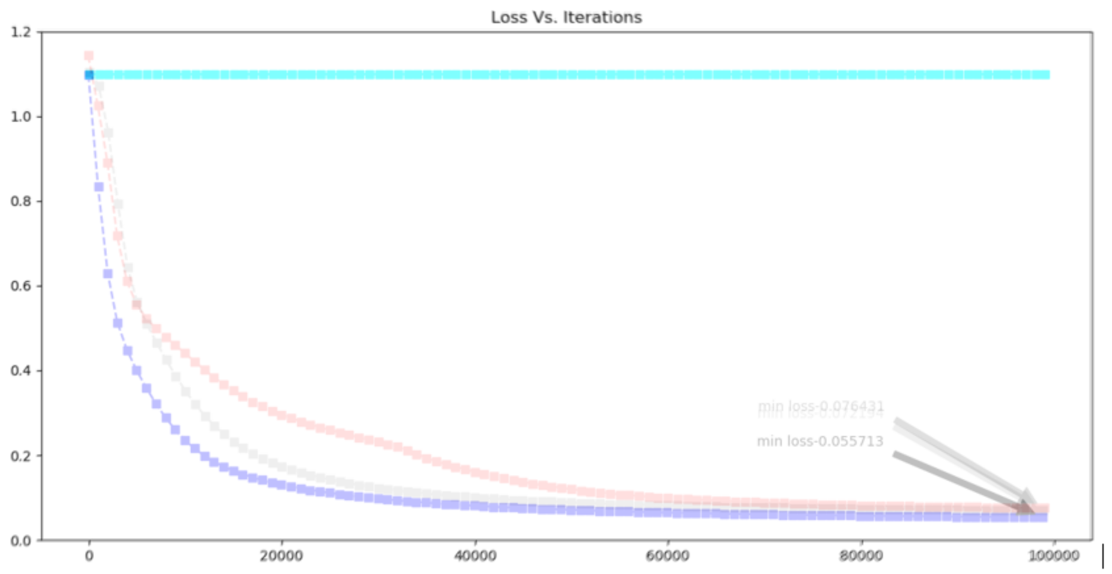


Figure 7: Impact of different Initialization of the weight vectors.

Code Part for different Initializations of weight vectors.

Command Line

```
 np.random.seed(0)
W1 = (np.random.randn(nn_input_dim, nn_hiddim)) / np.sqrt(nn_input_dim)
print(W1.shape)
b1 = np.zeros((1, nn_hiddim))
W2 = (np.random.randn(nn_hiddim, nn_output_dim))/ np.sqrt(nn_hiddim)
print(W2.shape)
b2 = np.zeros((1, nn_output_dim))
```

Info: We observe the graph in the cyan colour for all weight vectors initialized to zero, never converges. Thus with the change of the initialization point of weight vectors, we can achieve a different converegence rate as demonstrated in the figure above.

3.5 Learning Rate

We train our neural network with different training iterations to observe the optimal size of the Training set. We observe this to be a tradeoff of Time vs. Loss. In training deep networks, it is usually helpful to anneal the learning rate over time. Good intuition to have in mind is that with a high learning rate, the system contains too much kinetic energy and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function. Knowing when to decay the learning rate can be tricky: Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a long time. But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can. There are three common types of implementing the learning rate decay:

Step decay: Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. These numbers depend heavily on the type of problem and the model. One heuristic you may see in practice is to watch the validation error while training with a fixed learning rate, and reduce the learning rate by a constant (e.g. 0.5) whenever the validation error stops improving. **Exponential decay**. has the mathematical form $\alpha = \alpha_0 e^{-kt}$, where a0 ,k are hyperparameters and tt is the iteration number (but you can also use units of epochs). 1/t decay has the mathematical form $\alpha = \alpha_0/(1 + kt)$ where a0,k are hyperparameters and tt is the iteration number. In practice, we find that the step decay is slightly preferable because the hyperparameters it involves (the fraction of decay and the step timings in units of epochs) are more interpretable than the hyperparameter kk. Lastly, if you can afford the computational budget, err on the side of slower decay and train for a longer time.

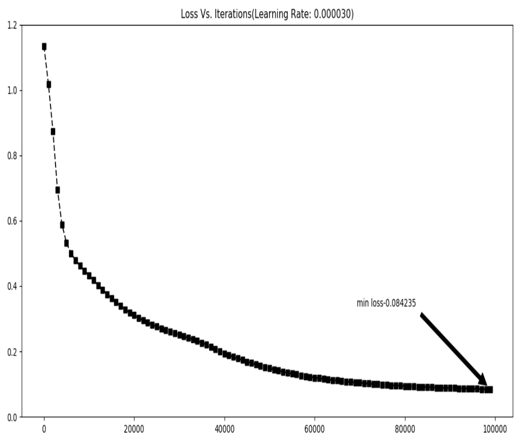
Second order methods

A second, popular group of methods for optimization in context of deep learning is based on Newton's method, which iterates the following update:

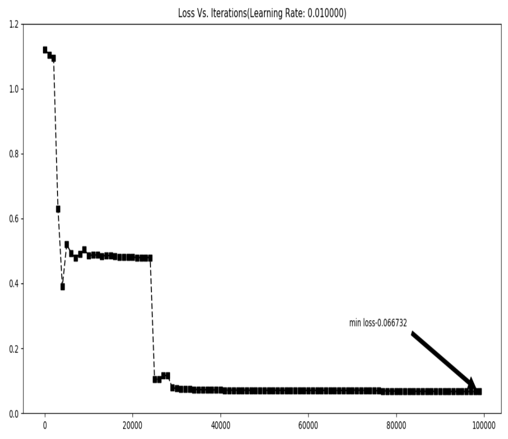
$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

Figure 8: Newton's Method

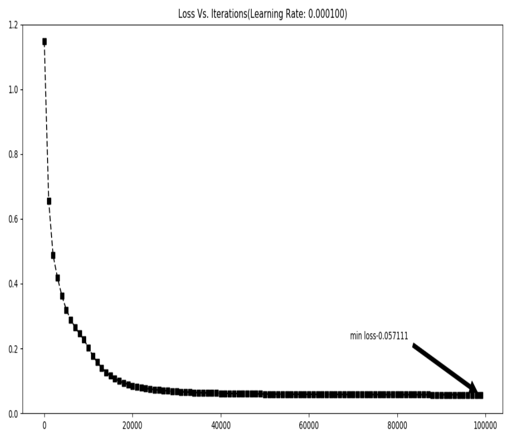
Here, Hf(x)Hf(x) is the Hessian matrix, which is a square matrix of second-order partial derivatives of the function. The term d(f(x)) is the gradient vector, as seen in Gradient Descent. Intuitively, the Hessian describes the local curvature of the loss function, which allows us to perform a more efficient update. In particular, multiplying by the inverse Hessian leads the optimization to take more aggressive steps in directions of shallow curvature and shorter steps in directions of steep curvature. Note, crucially, the absence of any learning rate hyperparameters in the update formula, which the proponents of these methods cite this as a large advantage over first-order methods.



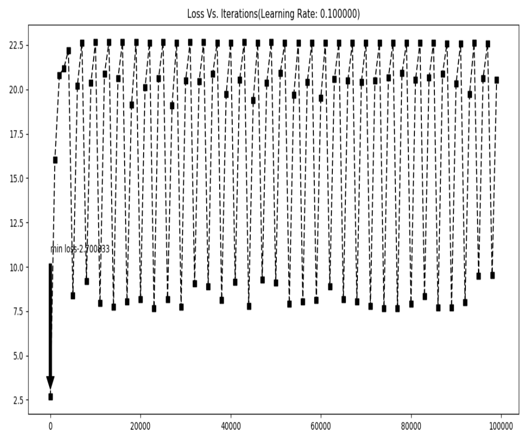
(a) for Learning Rate of 0.0003



(b) for Learning Rate of 0.01



(c) For Learning Rate of 0.001



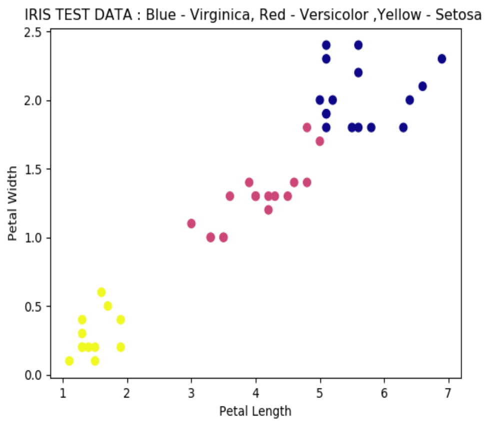
(d) For Learning Rate of 0.1

Figure 9: Loss/Iterations for Different pairs of Features varied with Learning Rates.

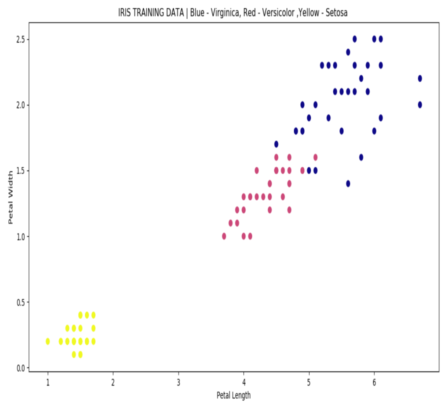
Info: The Learning rate of 0.1 is the case of typical situation when the system ping pongs between the two points and never converging to the solution. The Fig.(c) above, is the typical choice of Learning Rate that one would want in the design.

4 Accomplishments

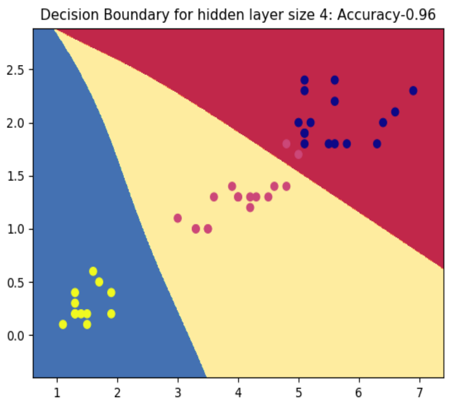
We achieve a single layer neural network that we train on our data after achieving the feature selection, optimizing the learning rate, deciding the best size of hidden layer. We then compare our result with the logistic regression on the same dataset to show our enhancements over the same.



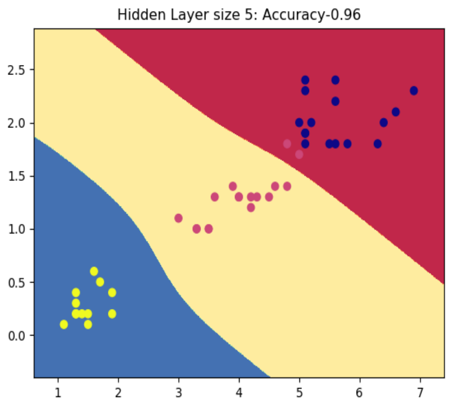
(a) Initial Test Data



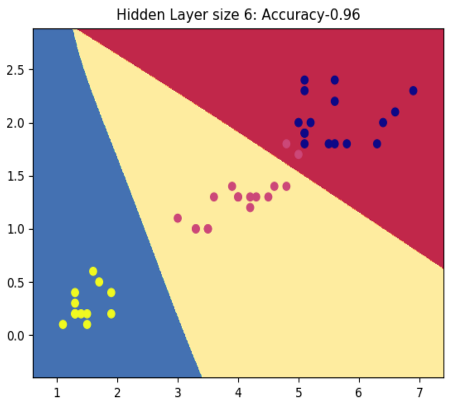
(b) Initial Train Data



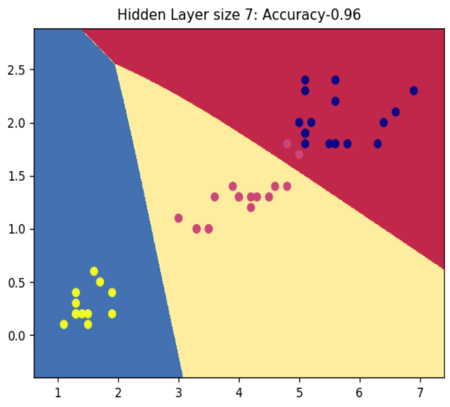
(c) Hidden Layer Size - 4



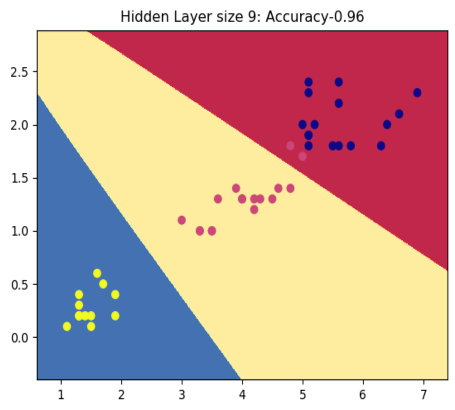
(d) Hidden Layer Size - 5



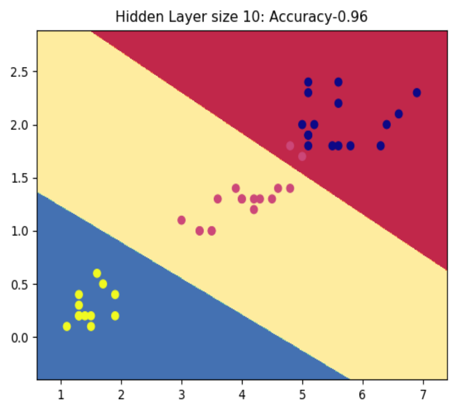
(e) Hidden Layer Size - 6



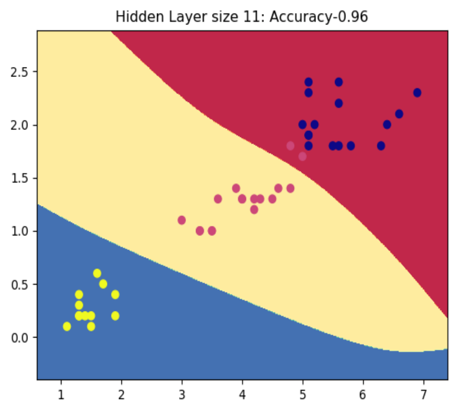
(f) Hidden Layer Size - 7



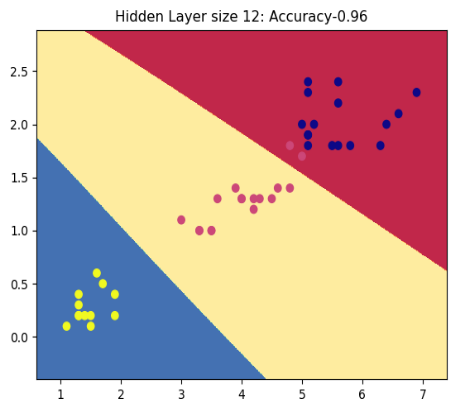
(g) Hidden Layer Size - 9



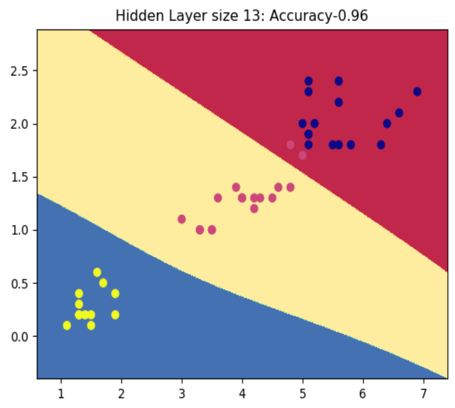
(h) Hidden Layer Size - 10



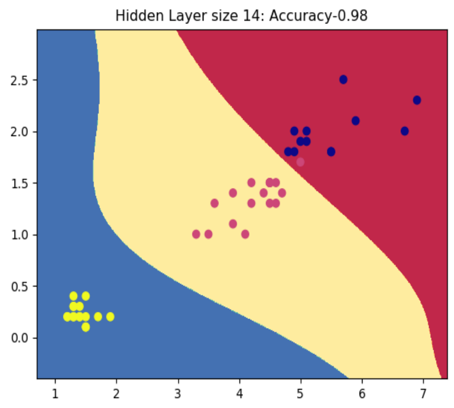
(i) Hidden Layer Size - 11



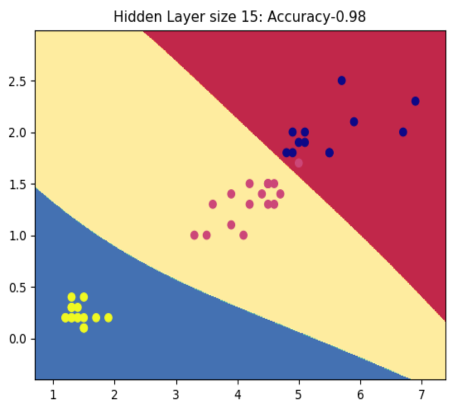
(j) Hidden Layer Size - 12



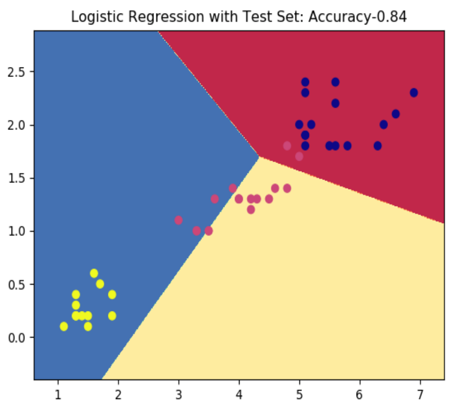
(k) Hidden Layer Size - 13



(l) Hidden Layer Size -14

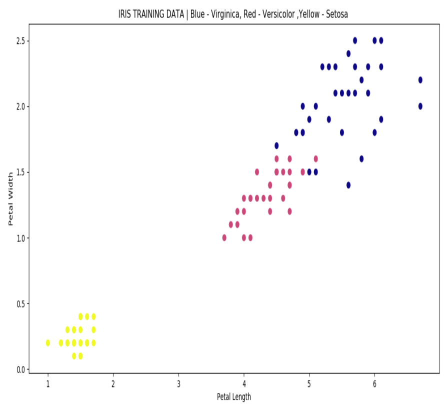


(m) Hidden Layer Size -15

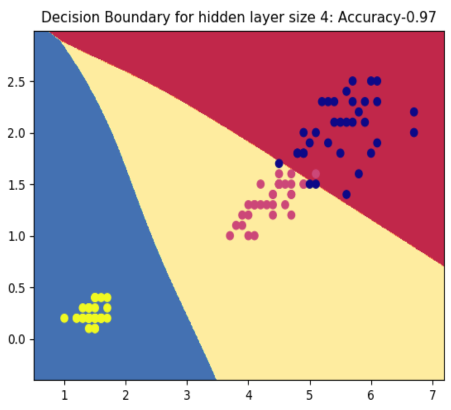


(n) Logistic Regression Accuracy

Figure 10: Our Implementation of Neural Networks and how it fares with the Logistic Regression

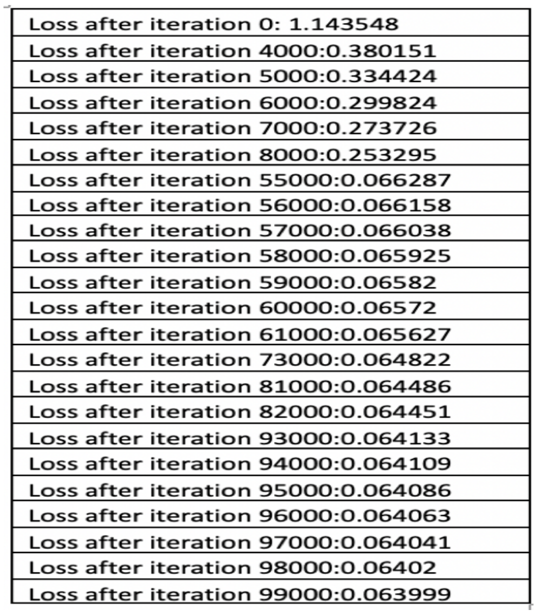


(a) Training Data Set, 70% of IRIS Data set randomly selected



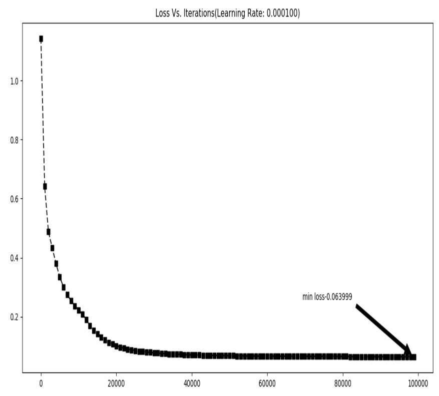
(b) Training Data, Decision Boundary

Figure 11: The Training data and the boundary of Training Data



Loss after iteration 0: 1.143548
Loss after iteration 4000:0.380151
Loss after iteration 5000:0.334424
Loss after iteration 6000:0.299824
Loss after iteration 7000:0.273726
Loss after iteration 8000:0.253295
Loss after iteration 55000:0.066287
Loss after iteration 56000:0.066158
Loss after iteration 57000:0.066038
Loss after iteration 58000:0.065925
Loss after iteration 59000:0.06582
Loss after iteration 60000:0.06572
Loss after iteration 61000:0.065627
Loss after iteration 73000:0.064822
Loss after iteration 81000:0.064486
Loss after iteration 82000:0.064451
Loss after iteration 93000:0.064133
Loss after iteration 94000:0.064109
Loss after iteration 95000:0.064086
Loss after iteration 96000:0.064063
Loss after iteration 97000:0.064041
Loss after iteration 98000:0.06402
Loss after iteration 99000:0.063999

(a) Table for Loss/Iteration for Training Data



(b) Loss/Iteration Curve for Training Data

Figure 12: The Loss/Iteration on the Training data

Info: Accuracy Measurement for the above comparison has been done using a simplistic accuracy measurement tool, wherein we just calculate the number of samples that belong to the correct bin/-total no. of points.

We achieve a single layer neural network with accuracy of 98% vs. the Logistic Regression achieving an accuracy of 84% with the same data set. Thus we are able to fully implement and prove increment of accuracy with our implementation as was the target for the Project.

5 Efforts and Challenges

Following are the efforts and Challenges that we made for achieving our results:

1. Efforts: Post the task of creating a neural network, we tweaked the Neural Net to optimize it to give the best performance. We firstly chose the two best feature sets, played around with the Learning Rate, played with weight vector initialization values, the hidden layer size, number of iterations. To finally achieve the accuracy of 98%

2. Challenges: We planned to conduct the similar analysis for three variables with three feature sets, that would require 3D rendering of graphs and plotting of 2-Dimensional Planes as the separating boundary. Also, we weren't able to achieve the accuracy fo 100% with the data set in picture even after tweaking with the parameters.

6 Future Work

Here we achieve a single layer Neural Network, we in future would want to extend and see the impact of the following.

1. Extend the number of hidden layers, and move towards deep Networks to see the impact on the result.

2. Secondly, our back propagation works for one hidden layer. How would we implement the back propagation for multi layer is worth a topic to ponder.

3. Currently we have in our implementation chosen two features set that we found optimally best fitting. We would like to take this ahead by selecting three features simulatneously and plotting a separation plane in 3D for three Classes.

References

[1] Understanding Machine Learning: From Theory to Algorithms Textbook by Shai Ben-David and Shai Shalev-Shwartz

[2] https://archive.ics.uci.edu/ml/datasets/iris

[3] https://www.towardsdatascience.com

[4] https://medium.com/@martinpella/logistic-regression-from-scratch-in-python-124c5636b8ac

[5] https://www.wildml.com

[6] http://adventuresinmachinelearning.com/neural-networks-tutorial/

[7] https://skymind.ai/wiki/neural-network