COMP462 Final Project

MULTILAYER PERCEPTRON

GOKTUG KAYACAN

Contents

Multi Layer Perceptron

- MLP is a network of connected perceptrons.
- Every perceptron is connection has a weight.

- Weights are adjusted by using backpropogation.
- Backpropogation is a specialized method for gradient descent.

- Figure 1 is an output of the implementation, it shows the topography of a perceptron with 4 inputs, 2 hidden layers, 1 output.
- MLPs have several advantages on conventional machine learning algorithms:
  - More robust due to way weights are adjusted.
  - Neural network's performance is highly effected by training time, the learning can continue after initial training.
  - More adaptive to the noisy data.
  - Depending on the training period, neural networks generally have low error rate, high degree of accuracy.

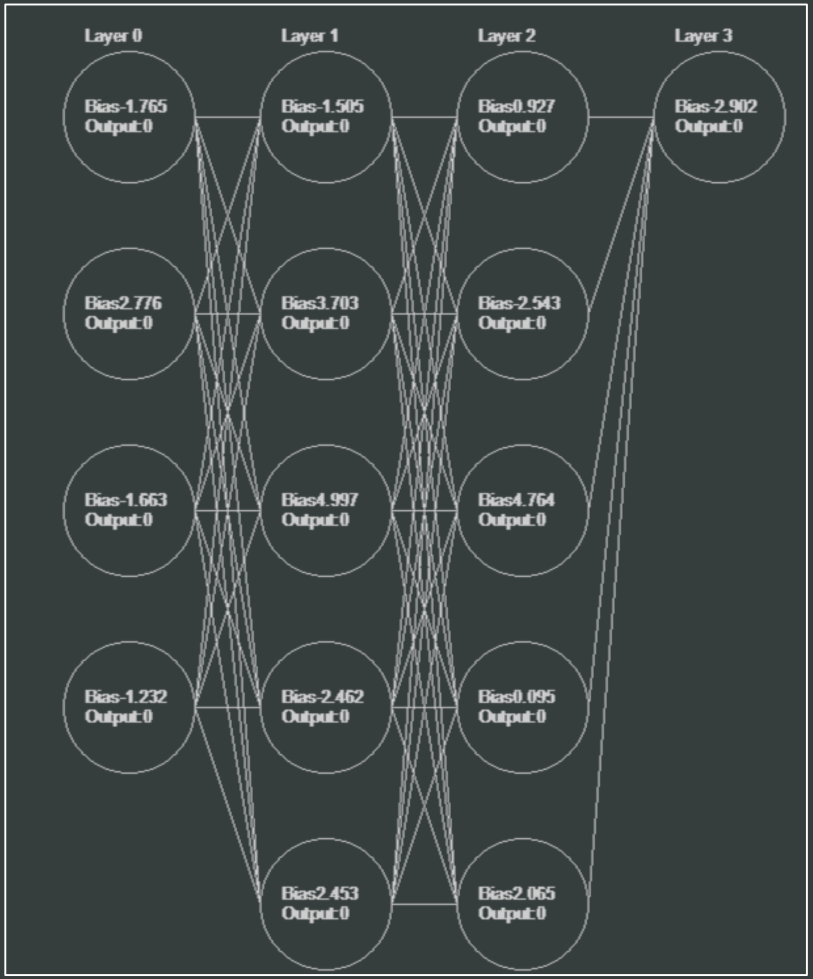


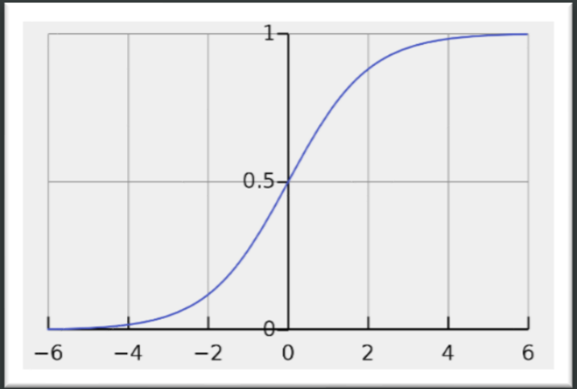
Fig1: 4 Layer Perceptron Network

Sigmoid

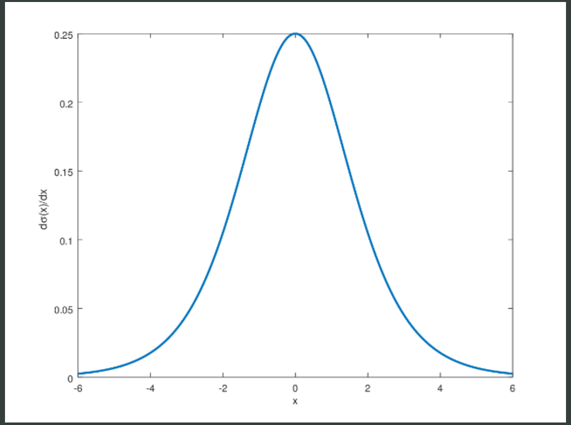
- Perceptrons uses “sigmoid” function as the activation function: $sigmoid(x) = \frac{1}{1+e^{-x}}$
- Implementation also uses derivative version of the sigmoid when doing backpropogation: $\frac{e^{x}}{(1+e^{x})^{2}}$.
- When applying backpropogation, these methods are used to change the internal values to adjust weights.
- Sigmoid function locks the output between 0 and 1. The results can be “normalized” in order to fit the required values.
- Normalized values can be “inverse normalized” to return back to the bounded values.

$$Normalize(x) = \frac{x - min}{\max - min} \qquad InverseNormalize(x) = x \ * (\max - min) + min$$

$$min = minimun\ value\ normalized\ x\ can\ take$$
$$\max = maximum\ value\ normalized\ x\ can\ take$$



Sigmoid Function

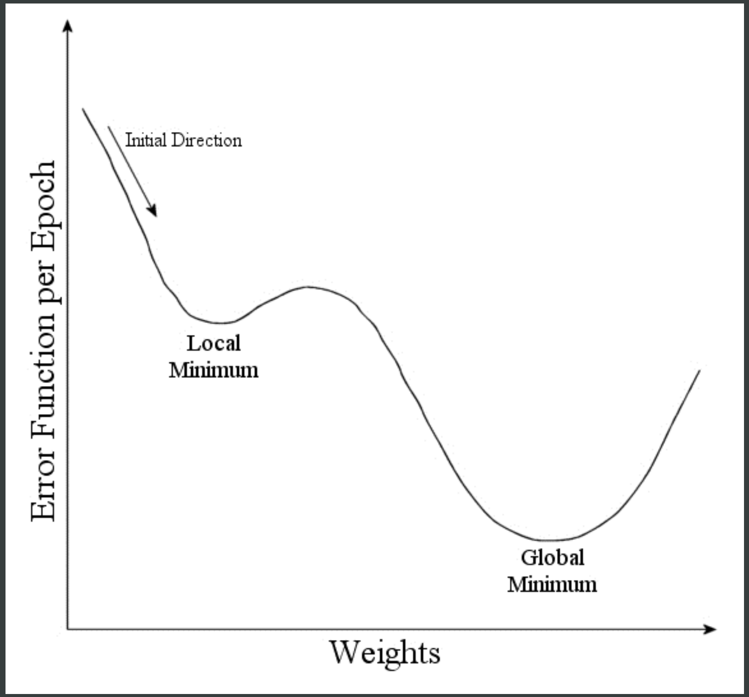


Derivated Sigmoid Function

Backpropogation

- Backpropogation is the method used to train the network by using gradient descent.
- Backpropogation is applied from output to input (final layer to beginning layer) with the gradient of the output layer being calculated first and the gradient of the beginning layer calculated last.
- Backpropogation error is calculated by the mean squared error

$$Error = \frac{1}{2N} * \sum_{i=1}^{N} (\bar{y}i \ - yi)^2$$



Backpropogation error over iterations

Implementation Backpropogation

- In order to update weights using backpropogation following method is used:
  - For each, Layer i in this network
    - For each, Perceptron j in the layer i
      - For the weight k of the perceptron j
        - Set the weights as: $w_i = w_i - \left(\alpha * \delta_{jk}^{i}\right)$

$\delta = delta\ value$s of this network
$\alpha = Learning\ rate$
$w = weights\ of\ the\ perceptron$

- In order to update delta values:
  - For each, Layer i in this network
    - For each, Perceptron j in the layer i
      - For the weight k of perceptron j
        - Set the delta values as: $\delta_{jk}^{i} = \delta_{jk}^{i} + \sigma_{ij} * Sigmoid(previous\ layer's\ output(i-1))$
        - This can be written as: $\delta_{jk}^{i} = \sum_{n=1}^{k}(\sigma_{ij} * Sigmoid(i-1))$

$\sigma = sigma\ value$s of this network

Backpropogation

- In order to adjust sigma values of the network:
  - For each, Layer i in this network
    - For each, Perceptron j in the layer i
      - If i is the output layer:
        - $\sigma_{ij} = (Sigmoid(Output(j)) - \hat{y}) * Sigmoid'(j.Output)$
      - If i is a hidden layer or the input layer:
        - For the weight k of the perceptron j
          - $\sigma_{ij} = (Sigmoid(Output(j)) - \hat{y}) * \sum_{i}^{N}(k * \sigma_{(i+1)j})$
- In order to adjust Bias of the perceptrons:
  - For each, Layer i in this network
    - For each, Perceptron j in the layer i
      - $Bias(j) = (Bias(j) - \alpha * \sigma_{ij})$

$\hat{y}$ = Expected output

$Sigmoid'$ = derivative of sigmoid function

Activation Function

- The activation function’s activation value is calculated by:
  - For each, Layer i in this network
    - For each, Perceptron j in the layer i
      - For the weight k of perceptron j
        - $\theta = Bias(j) + \sum_{n=1}^{N} k_n * x_n$ $\theta: Activation\ value$
        - Activation: $Sigmoid(\theta)$
- Activation function is applied by a layer by layer basis
- This is more effective, compared to Applying activation function on a perceptron by perceptron basis.

Training

- In order to apply backpropogation program updates sigma values, perceptron biases and the delta values for each input. Weight is updated after these values are updated.
- In order to train the network, algorithm iterates the following until a stop signal :
  - For every given input i,
    - Update Sigma values for i,
    - Update Biases for i,
    - Update Delta values for i,
      - Update Weights
        - Calculate SSE with updated Weights,
          - Repeat until stop signal.
- The stop signal can be a hard limit, such as maximum number of iterations or it can be an error threshold

Parameters and Outcomes

- The network results changes depending on these parameters:
  - Training time
  - Learning Rate
  - Number of perceptrons in each layer
  - Number of hidden layers

Following slides will show experimental data about the change in results when a single parameter is taken as a variable and others are kept static.

The IRIS dataset is used for most of the experimentations,

dataset: “IRIS_Train” is used for training and “IRIS_Validate” is used for validation.

IRIS_Train consists of first 25 entries of every class, IRIS_Validate consists of last 25 entries of every class.

Parameters and Outcomes: Datasets

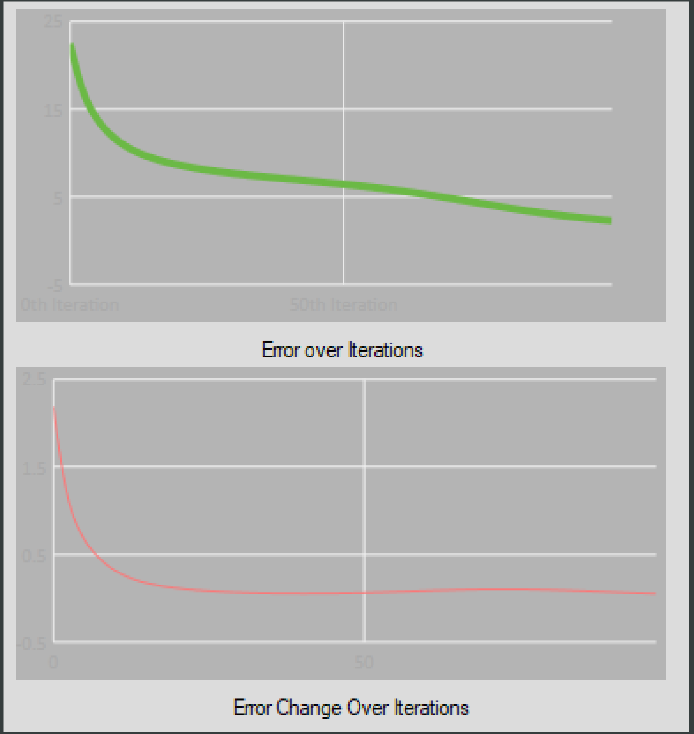
- IRIS Plant dataset has four features and three classes of 50 objects each.
- Features of the dataset are:
  - Sepal Length in cm
  - Sepal Width in cm
  - Petal Length in cm
  - Petal Width in cm

- MNIST digit dataset is a dataset with 60,000 training examples and a test set of 10,000 examples
- This dataset contains images of handwritten digits.
- This dataset can be vectorized on a pixel basis. Every single pixel colour denoting a feature.
- For this project I have download a vectorized version of this dataset from: https://pjreddie.com/projects/mnist-in-csv/
- Due to the size of this dataset experimentations are not done with this dataset.

Variable: Iteration Count

- For this experiment Constants are:
  - Learning Rate = 0.015,
  - Number of hidden Layers: 2,
  - 5 perceptrons in each hidden layer.

Iteration	Final Error	Time Elapsed	Error on Validation Set
100	2.2785763446621	28ms	36 Errors out of 75 Inputs
200	0.853265827070185	53ms	6 Errors out of 75 inputs
500	0.574344530788184	162ms	5 Errors out of 75 inputs
1000	0.520275111713132	383ms	6 Errors out of 75 inputs
1500	0.506294629052663	676ms	6 Errors out of 75 inputs
4000	0.412383901619222	2987ms	6 Errors out of 75 inputs



Change of Error over iterations and error change over iterations first 100 iterations

- Longer a network trains on a given datasets, results will usually get better and the general error will go down.

Variable: Learning Rate

- For this experiment Constants are:
  - Iterations: 1500
  - Number of hidden Layers: 2,
  - 5 perceptrons in each hidden layer.
  - 5 different networks are built and the average sum of these values are put into table

Learning Rate	Final Error	Best Error	Time Elapsed	Error on Validation Set
0.001	5.41012404785689	5.4101	438ms	50 Errors out of 75 Inputs
0.01	0.829682781664379	0.8296	545ms	6 Errors out of 75 inputs
0.05	0.593179353877293	0.5931	403ms	8 Errors out of 75 inputs
0.10	0.913247439860539	0.9132	407ms	13 Errors out of 75 inputs
0.2	0.506294629052663	1.7891	676ms	17 Errors out of 75 inputs
0.5	8.32762280968983	1.3305	449ms	60 Errors out of 75 inputs
1.0	8.97255711625508	0.4496	988ms	42 Errors out of 75 inputs

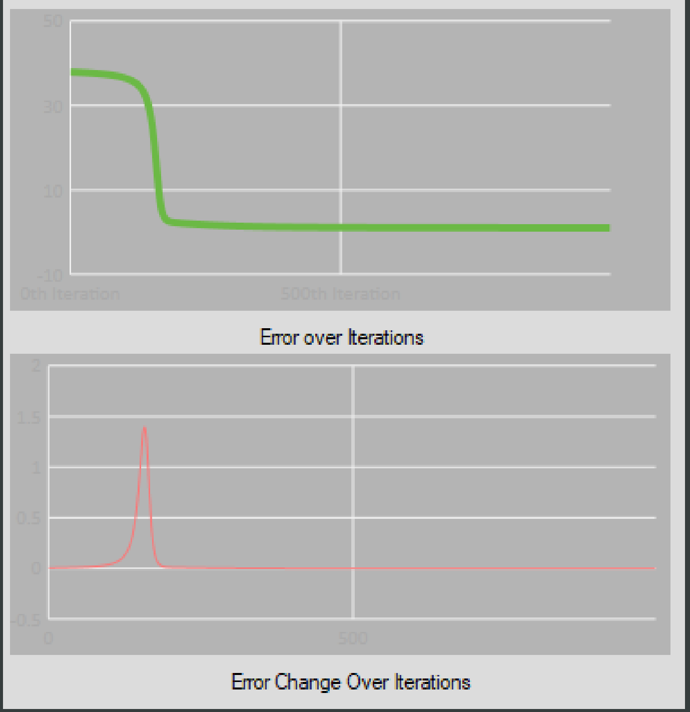


Figure1: Change in Error values over iterations at Learning Rate = 0.05

- High learning rates can cause fluctuation as algorithm can't manage to fit in a local or global minima.
- If the learning rate is too low, It takes much longer for the program to reach a minima.
- By manually or automatically changing the learning rate, global minima can be reach faster and more efficiently.

Learning Rate Graphs

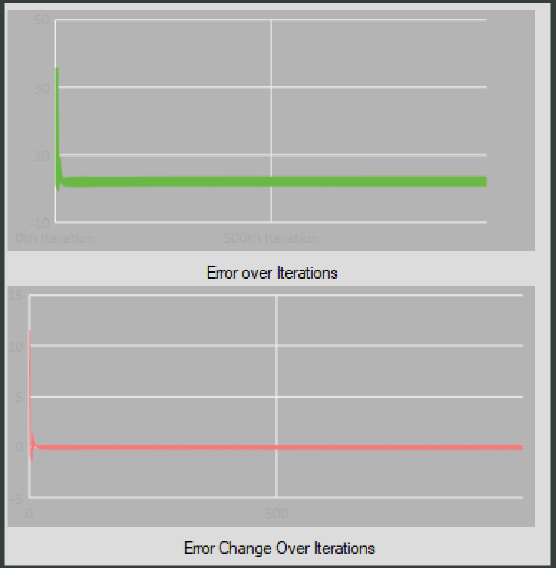


Figure1: Learning Rate: 0.2, Error rate fluctuating between 2.0 and 2.3

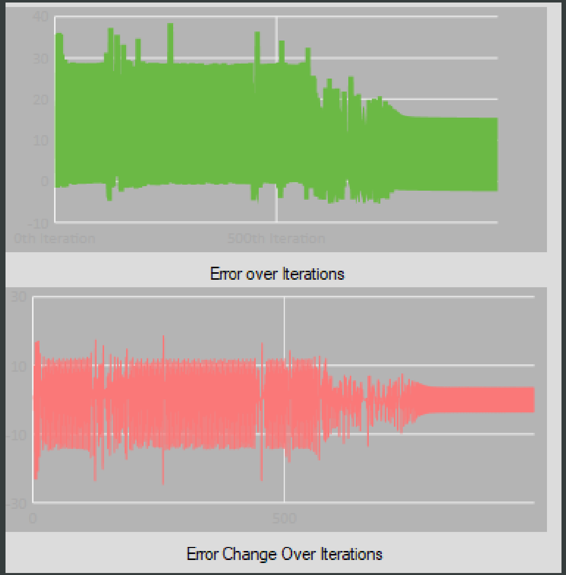


Figure2: Learning Rate: 0.5, Error rate fluctuating between 8.0 and 4.0

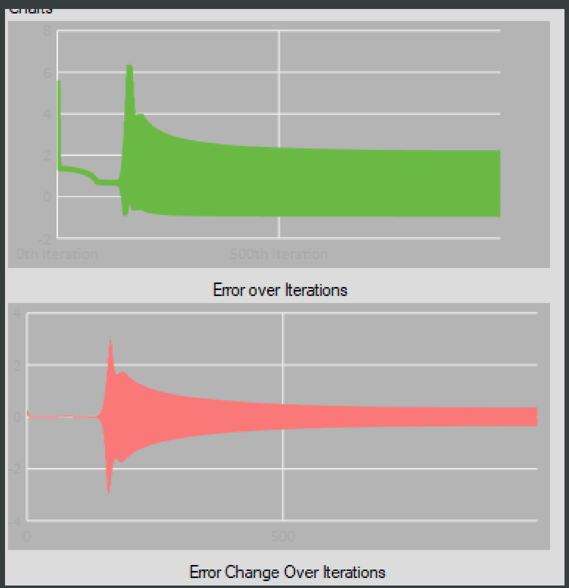


Figure3: Learning Rate: 1, Error rate fluctuating between 0.4, 0.8. This is the most successful high-learning rate outcome

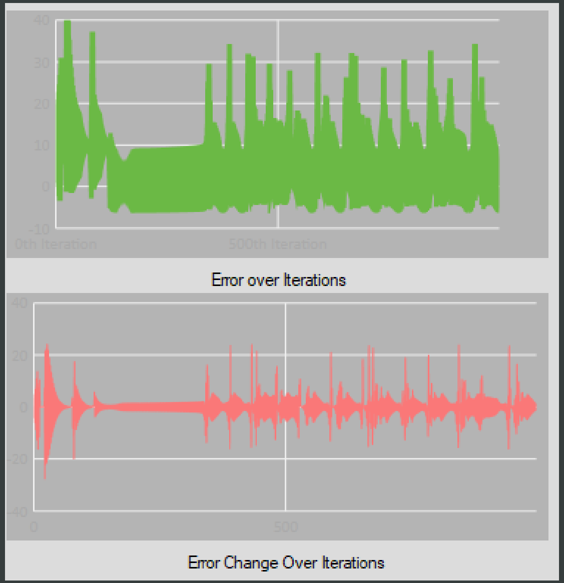


Figure4: Learning Rate: 1, Error rate fluctuating between 9.0, 2.0.

Variable: Perceptrons Per Layer, Network Width

- For this experiment Constants are:
  - Iterations: 1500
  - Number of hidden Layers: 2,
  - Learning Rate: 0.015
  - 5 different networks are built and the average sum of these values are put into table

- Wide networks will usually give a better result in the same amount of iterations but will take a lot longer to train.
- If a network is "reverse-cone" shaped, it might be harder for the network to reach the global minima

Perceptron Count in Hidden Layer 1	Perceptron Count in Hidden Layer 2	Final Error	Best Error	Time Elapsed	Error on Validation Set
5	4	0.863722652	0.8637	185ms	14 Errors
5	3	0.7840982	0.7841	102ms	10 Errors
5	2	1.24844776	1.2484	46ms	16 Errors
5	1	1.417381835	1.4174	6ms	23 Errors
4	5	1.276639443	1.2766	135ms	20 Errors
3	5	1.463385634	1.4634	68ms	25 Errors
2	5	5.5715296829	5.5715	24ms	50 Errors
1	5	5.5638937528	5.5638	6ms	50 Errors
10	10	0.30501507753	0.305	2039ms	4 Errors
20	20	0.11372489621	0.1137	6268ms	5 Errors
50	50	0.12599633003	0.126	32391ms	3 Errors
20	5	0.43398938757	0.434	2233ms	11 Errors
5	20	0.87997151674	0.88	2052ms	14 Errors
10	5	0.348735734703	0.3487	1065ms	6 Errors
5	10	0.91913169700	0.9191	1053ms	9 Errors

Variable: Number of Hidden Layers, Network Depth

- For this experiment Constants are:
  - Iterations: 1500
  - Perceptrons Per Layer: 5
  - Learning Rate: 0.015
  - 5 different networks are built and the average sum of these values are put into table

Hidden Layer Count	Final Error	Best Error	Time Elapsed	Error on Validation Set
1	2.014458263630	2.0145	20ms	31 Errors
3	1.079542634063	1.0795	1018ms	14 Errors
5	1.015556045709	1.0156	2063ms	16 Errors
10	5.563152868943	5.5556	4163ms	50 Errors
15	5.557186554761	5.5572	6496ms	50 Errors
25	5.612684751744	5.563	10972ms	50 Errors
50	5.5624657271875	5.6525	22244ms	50 Errors
100	5.556836504370	5.5568	69082ms	50 Errors

- Values approach the normal medium of 5.50, higher the number of hidden layers are.
- This is because we have very little data and algorithm can do only so much with it. If we had a larger, more convoluted dataset deeper networks could fare better than more shallow ones.
- The reason program finds 25 out of 75 outputs correct (50 errors out of 75 inputs) is because selection value is so high or low it will always round up to be correct for a single class

Weight Changes

- Weights of the individual perceptrons change depending on the backpropogation error.
- Figure1 shows the change of weights on a network with:
  - Structure of {4 5 5 1},
  - Learning Rate: 0.02,
  - 50 iterations
- The weight values change more sporadically, closer to a minima point and change more gradually further away from a point.
- In this test case, our starting general error was 8.166 and general error at 50 iterations is 3.449.

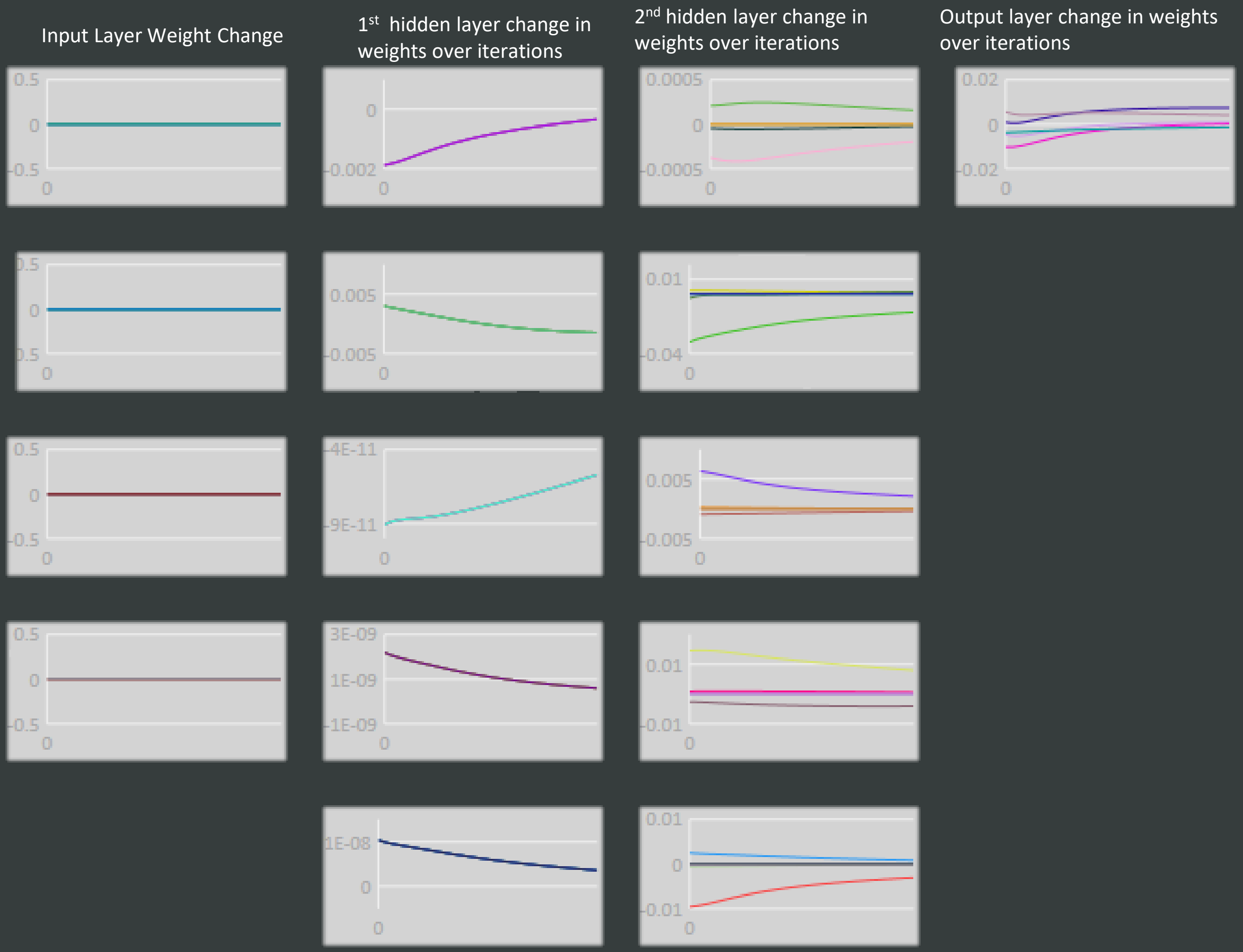
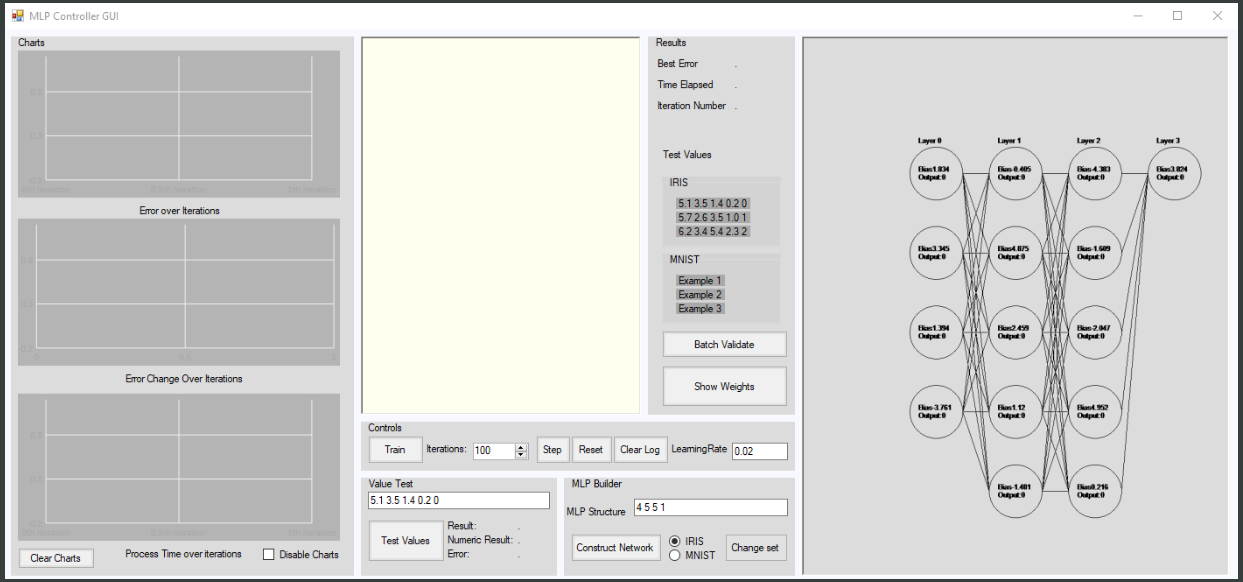


Fig1: Weight Change charts depending on the iterations

Project implementation

- Written in C#
  - Live Charts API* Used to draw
- Interface is built with Windows Forms
  - Interface allows user to control the training process by changing the learning rate at any time.
  - Interface shows:
    - General error per iteration,
    - Change of general error per iteration,
    - Perceptron weights and biases,
    - Validation results
  - Training set can be changed any time, dataset file format is .csv (Comma separated file)
  - New networks can be built with custom structures.



IRIS:
5.1 3.5 1.4 0.2 0
5.7 2.6 3.5 1.0 1
6.2 3.4 5.4 2.3 2

Iterations: 100
LearningRate 0.02

Value Test: 5.1 3.5 1.4 0.2 0

MLP Structure 4 5 5 1

[✓ IRIS] MNIST

MLP Controller Interface

References

- https://brilliant.org/wiki/backpropagation/ , John McGonagle, George Shaikouski, Andrew Hsu, Jimin Khim, Christopher William, Backpropogation
- http://www.vilipetek.com/2015/02/12/multilayer-perceptron/ , Vilipetek, MultiLayer Perceptron
- https://www.youtube.com/watch?v=L_PByyJ9g-I, “The One”, “Neural Network – Back-propogation Tutorial in C#”
- http://mathworld.wolfram.com/SigmoidFunction.html, Sigmoid Function
- http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-12.html, “What is a softmax function?”
- http://deeplearning.net/tutorial/mlp.html , Multilayer Perceptron
- https://machinelearningmastery.com/neural-networks-crash-course/, “Crash course on multi-layer perceptrons neural networks”, Jason Browniee
- http://www.uni-weimar.de/medien/webis/teaching/lecturenotes/machine-learning/unit-en-multilayer-perceptron.pdf
- https://www.uow.edu.au/~markus/teaching/CSCI323/Lecture_MLP.pdf , Lecture on MLP
- Introduction to Machine Learning Second Edition, Ethem Alpaydın, Section 11, Multilayer Perceptrons
- https://github.com/HectorPulido/Multi-layer-perceptron ,an Implementation of MLP
- http://www.morettícb.com/blog/mlp-topology-workbench-a-playground-for-multilayer-perceptrons/ , Neural network playgrounds
- http://home.lu.lv/~janiszu/papers/2005_mlp_extension.pdf, Mlp extension
- https://www.researchgate.net/post/How_to_decide_the_number_of_hidden_layers_and_nodes_in_a_hidden_layer, How to decide the number of hidden layers and nodes in a hidden layer
- http://www.irjes.com/Papers/vol3-issue12/Version%201/J3126569.pdf
- https://www.kaggle.com/donfuzius/vectordigits, Handwritten MNIST like dataset
- https://pjreddie.com/projects/mnist-in-csv/ vectorized MNIST dataset