MATH69111 Scientific Computing

Project 2

Neural Networks

Student ID:
11582048

December 13, 2024

Introduction

- Artificial Neural Networks are used to solve a large number of “hard” problems such as autonomous driving systems, machine translation or algorithmic trading.
- Objective of current task is to create a simple Neural Network from scratch for binary classification of complex data patterns using activation function
- Neural Network should be based on feed-forward algorithm and trained using stochastic gradient descent.
- Model results should be analyzed for different data patterns and network architecture using cost function and plots of boundary lines

Introduction

- Neural network is computational model inspired by the biological neurons in brain
- Network usually consists of multiple layers of neurons that are interconnected by weights and biases
- The main advantage of these kind of models is the opportunity to learn patterns and generalize them to perform classification or predictions
- Main purpose of the project is to create object-oriented implementation of neural network for binary classification problem
- Training of the model will be done by using stochastic gradient descent

Introduction

- The idea of the algorithm is following:
  - The input layer receives features from the dataset

$$a_j^{[1]} = x_j, j = 1, \dots, n_1.$$

  - Hidden layers perform transformations using activation functions, weights and biases

$$a^{[l]} = \sigma_l\left(W^{[l]}a^{[l-1]} + b^{[l]}\right), for\ l = 2, 3, \dots, L.$$

  - Output layer outputs the final classification
- Activation function used is Hyperbolic tangent as it non-linear and has centered output (from -1 to 1)

$$\sigma(z) = \tanh(z)$$

Introduction

- Training algorithm will be performed the following way:
  - Weights (W) and biases (b) will be initialized randomly
  - Feed-forward algorithm will be used for evaluating the output

$$a^{[l]} = \sigma_l\left(W^{[l]}a^{[l-1]} + b^{[l]}\right)$$

  - Measurement of function fit will be measured using cost function

$$C = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left|\left|y^{(i)} - a^{[L]}\left(x^{(i)}\right)\right|\right|_2^2$$

  - Gradients for backpropagation will be computed with respect to biases and weights

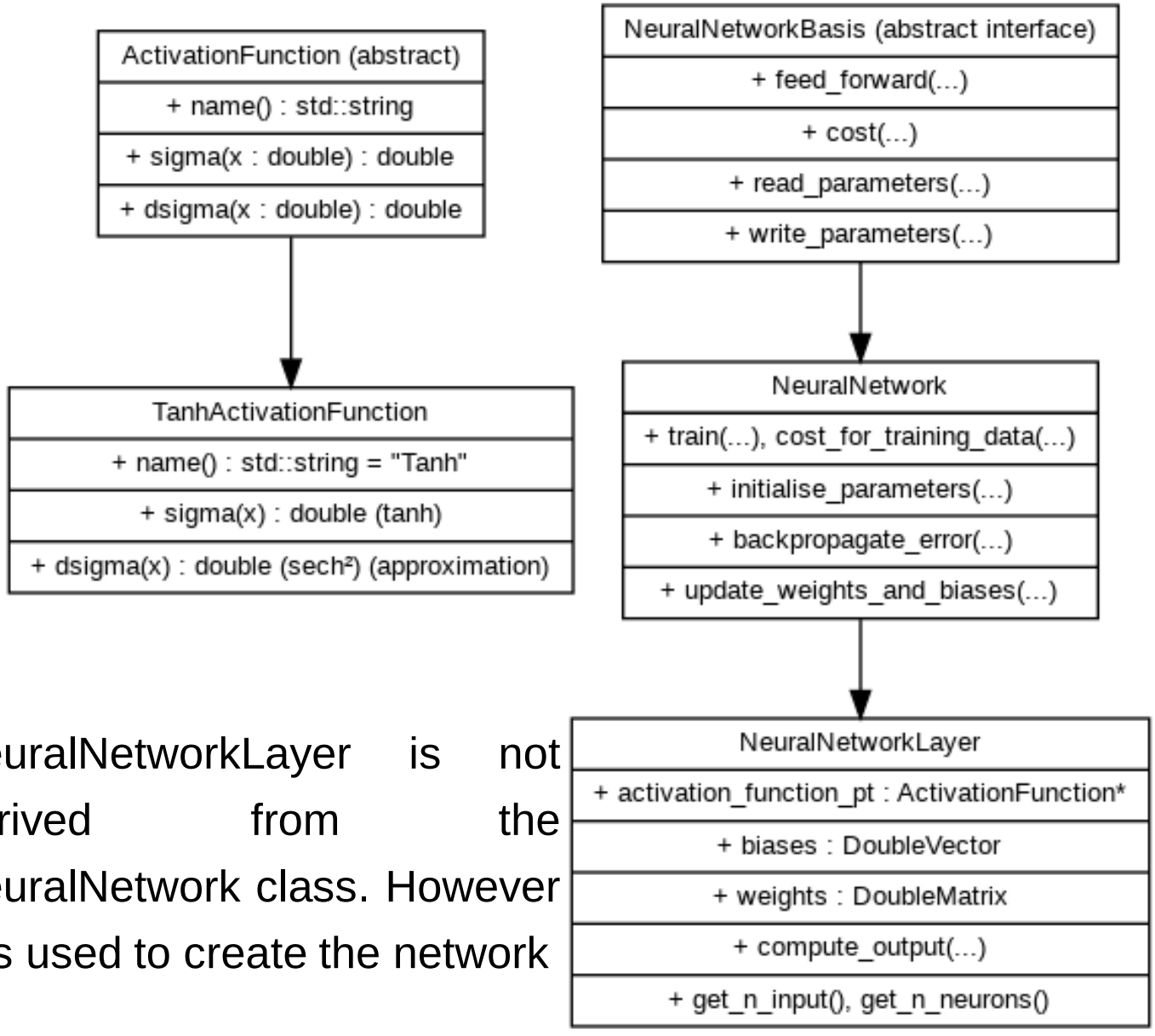
$$\frac{\delta C_{x^{(i)}}}{\delta w_{jk}^{[l]}} = \delta_j^{[l]} \cdot a_k^{[l-1]}, for\ l = 2, \ldots, L$$

  - Weights and biases will be updated using this approach

$$w_{jk}^{[l]} \leftarrow w_{jk}^{[l]} - \mu\frac{\delta C_{x^{(i)}}}{\delta w_{jk}^{[l]}}$$

Introduction

- Code structure is the following:



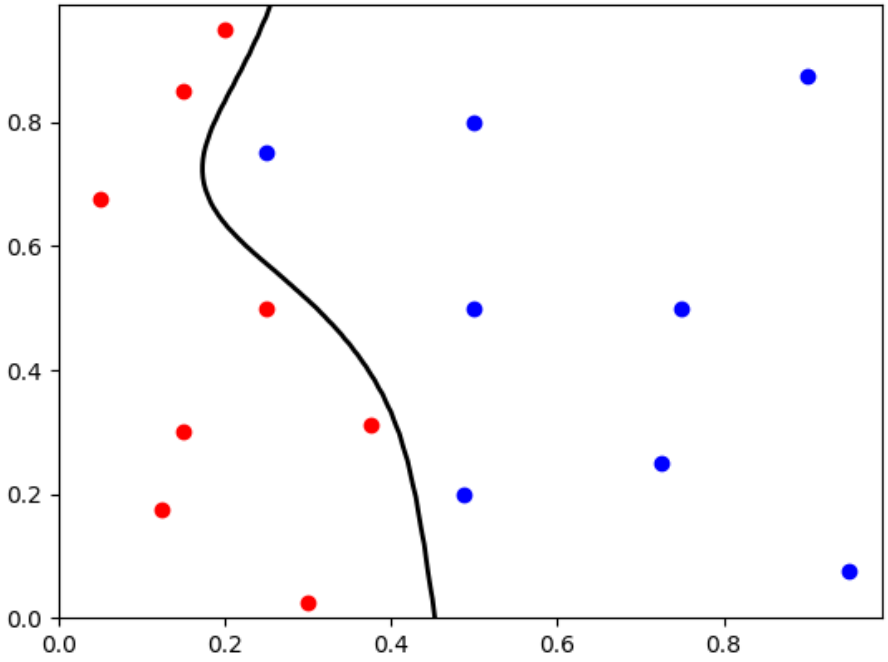
- NeuralNetworkLayer is not derived from the NeuralNetwork class. However it is used to create the network

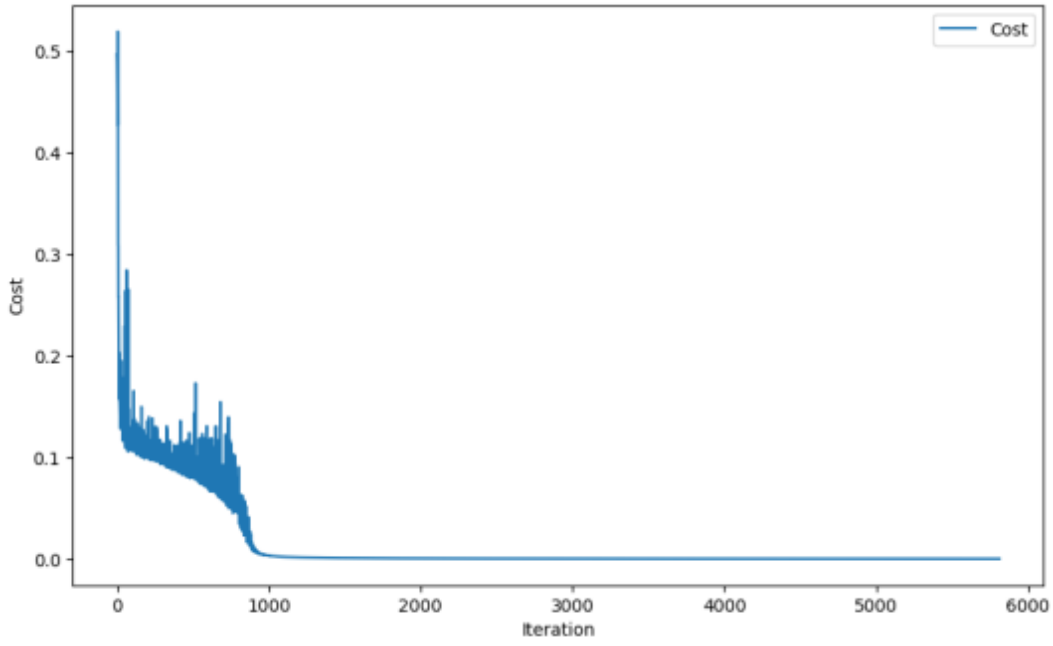
Introduction

- NeuralNetworkLayer – class for creating layers as objects
  - Data members: biases, weights, activation function_pt
  - Methods: compute_output(input, output)
- NeuralNetwork – class that forms a sequence of NeuralNetworkLayer objects into network
  - Main methods:
    - feed_forward(input, ouput): passes information through layers
    - cost(input, target): computes MSE
    - train(training_data, learning rate, tol, max_iter): trains the network
    - initialize_parametres(mean, std_dev): initializes parameters for training

Simple test case

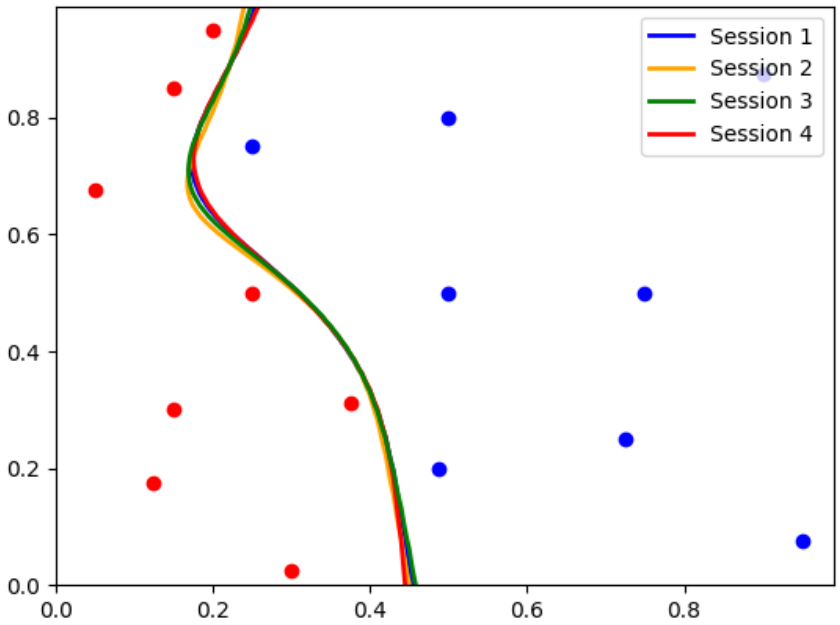
- The idea of task was to divide line between two regions containing red and blue dots on the plot marked as 1 and -1 respectively
- Suggested ANN architecture contains 4 layers: input layer, two hidden layers and output layer, containing (2, 3, 3, 1) neurons respectively. Tanh activation function was used.
- Learning rate η was specified at the value of η = 0.1, target cost was equal to $\tau = 1 \cdot 10^4$. Random values were drawn from the random distribution with zero mean and standard deviation of 0.1.
- I also shuffle the data to get faster convergence and better generalization
- Result was obtained in less then 6000 iterations.

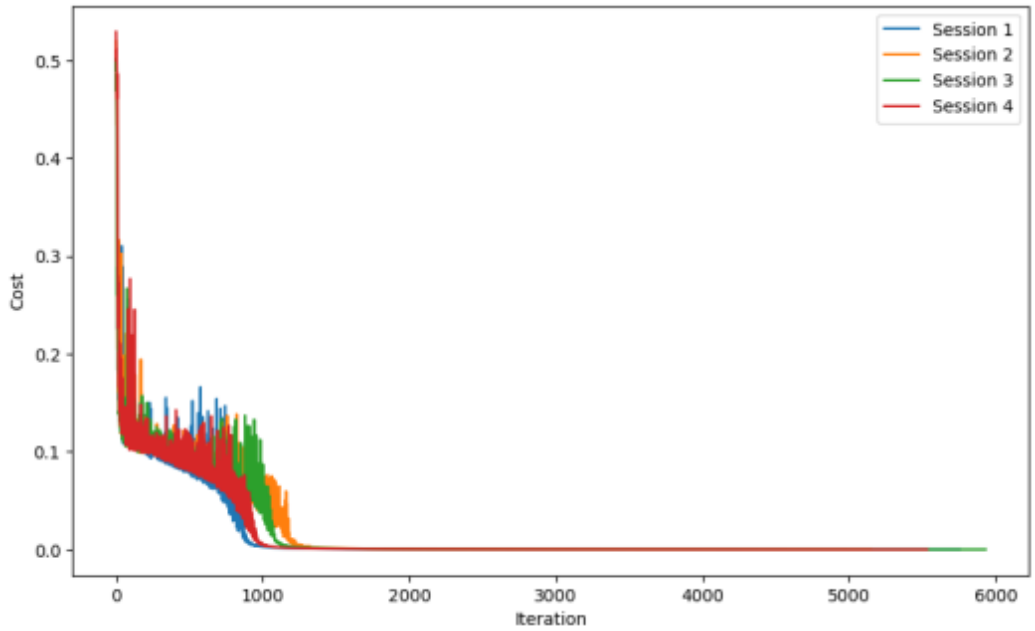




Simple test case

- If we repeat the training process four times, we might get such plots:

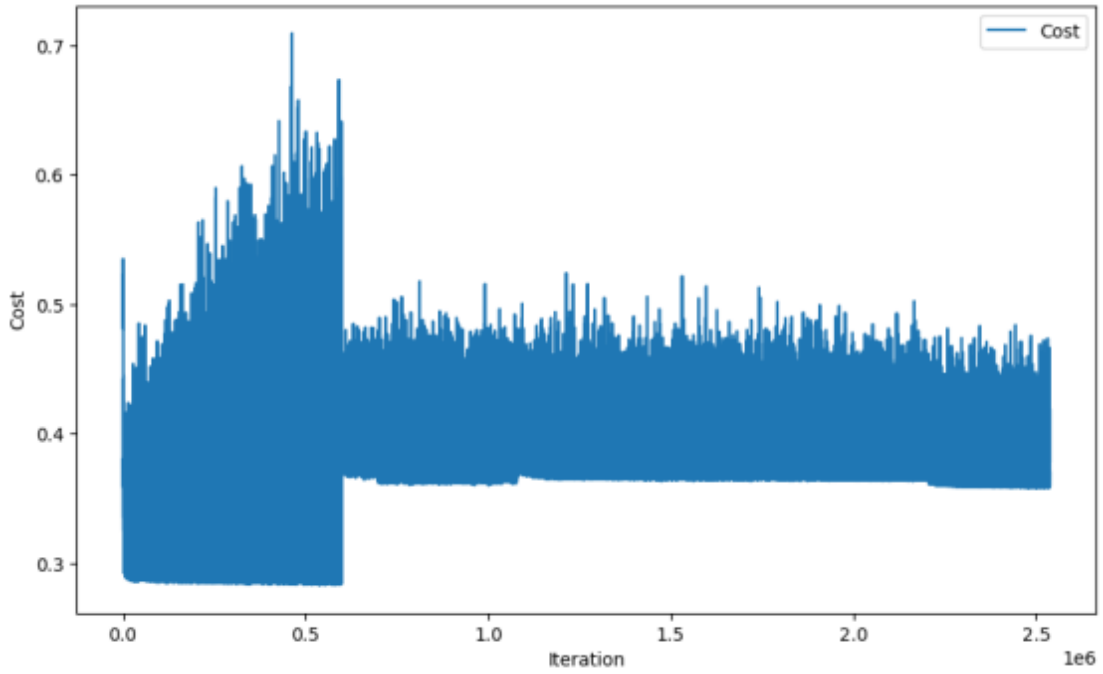




- From the graphs we might see that four runs have led to almost the same result with similar convergence speed. That is why the process is reliable.
- Any discrepancies might be explained by randomness in data shuffling and weights initialization. In case we try to run training process four times with equal random seed, obtained results will be similar

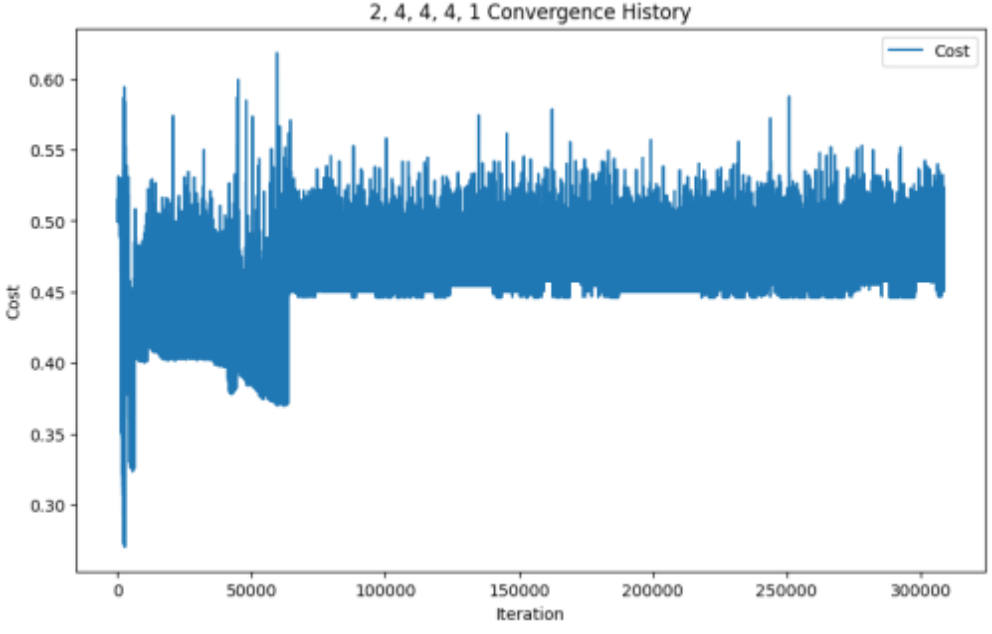
More challenging test case

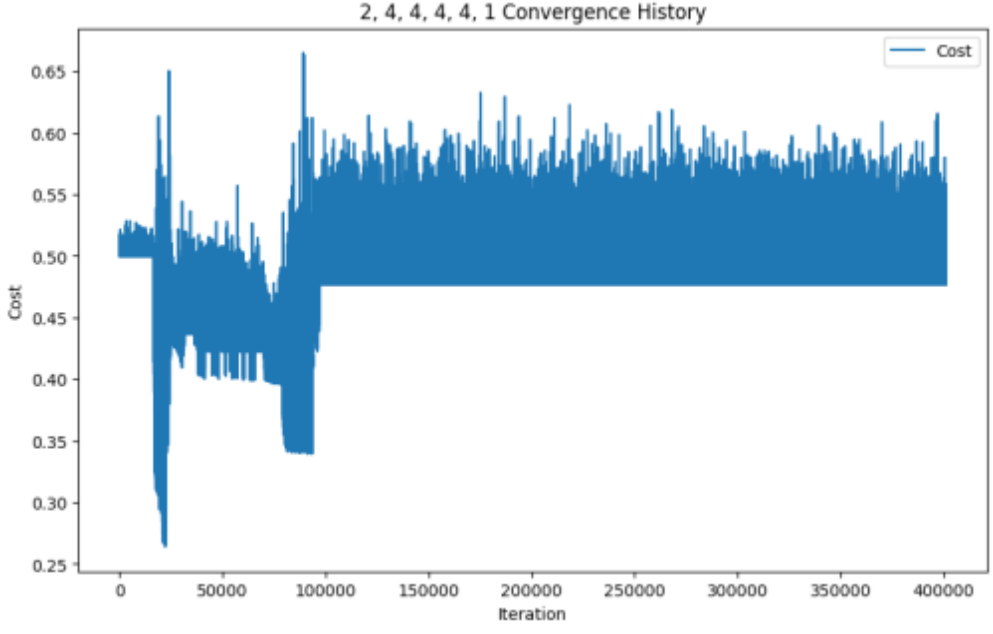
- If we create a network with 4 layers: input layer, two hidden layers and output layer, containing (2, 4, 4, 1) neurons respectively. Tanh activation function was used.
- Learning rate $\eta = 0.01$, total cost below $\tau = 1 \cdot 10^{-3}$, with normal distribution with zero mean and a standard deviation of 0.1. Maximum number of iterations equals to $4 \cdot 10^{6}$
- Model with such architecture did not manage to converge when my computer ran out of RAM



More challenging test case

- When ANN models do not converge we have two methods to achieve convergence: make model deeper (increase number of layers), make model wider (increase number of neurons in each layer)
- If we investigate the first method and try to use following architectures: (2, 4, 4, 4, 1) and (2, 4, 4, 4, 4, 1).

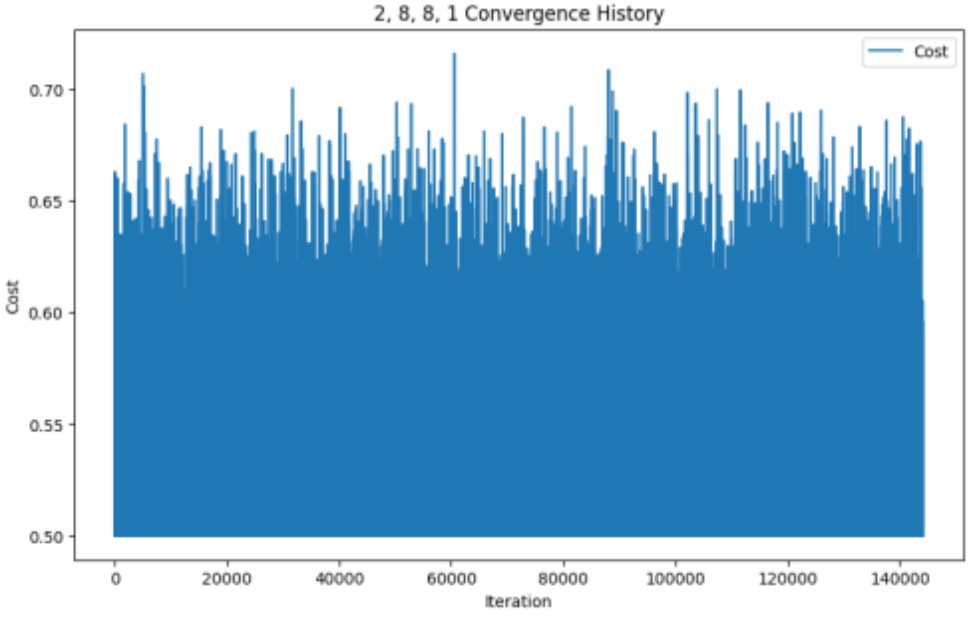


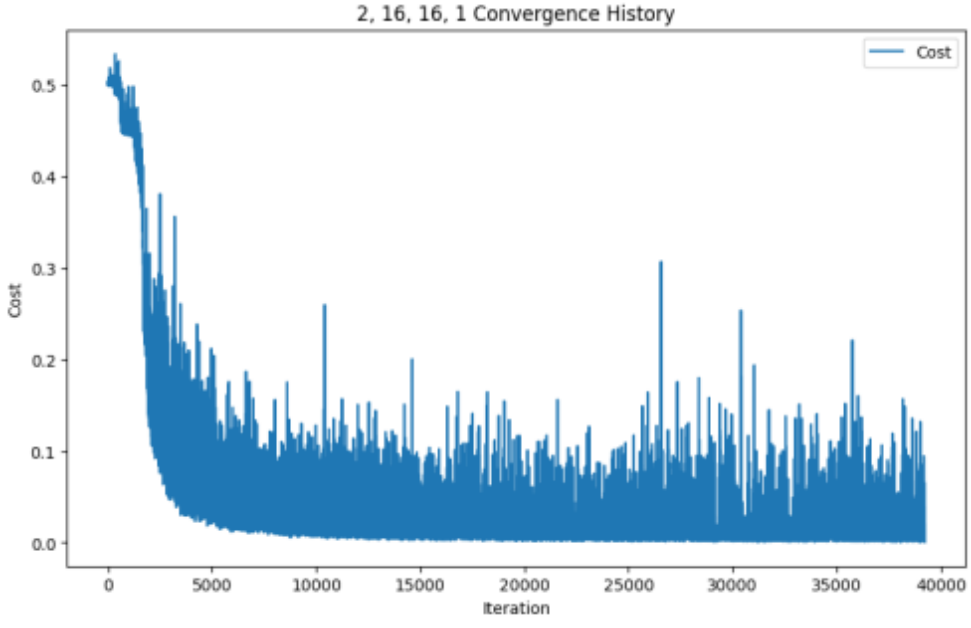


- Both models also do not manage to converge to required value up to the moment when my computer ran out of RAM

More challenging test case

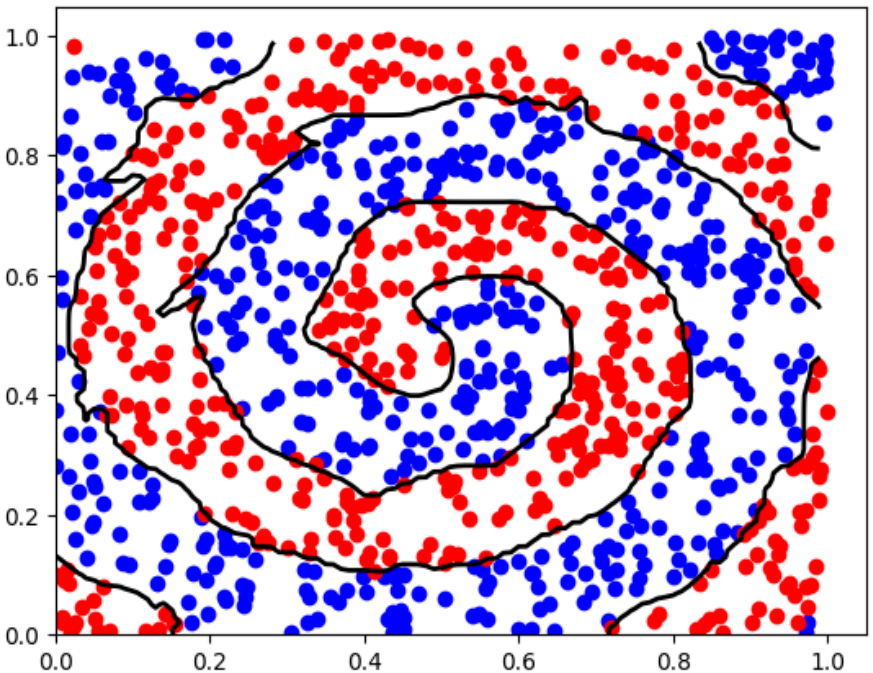
- As making the model deeper did not help, let's try to make the model wider.
- I used two suggested architectures: (2, 8, 8, 1) and (2, 16, 16, 1).
- Model with (2, 16, 16, 1) managed to produce satisfactory result which might be defined as minimizing loss to a stated level under stated amount of iterations. That means that model managed to converge



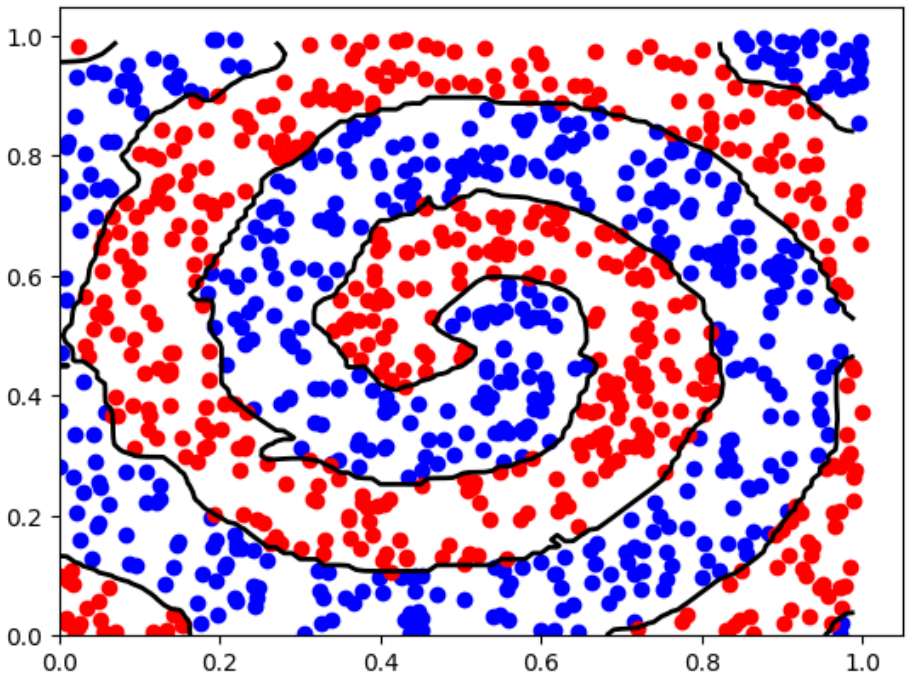


More challenging test case

- The widest suggested model managed to converge to a required loss level (right picture)
- This might be due to the factor that usually wider model better generalize simple tasks such as classification than deeper models, that are usually used during such complicated tasks as image recognition, NLP or order book analysis.
- If we stop the training of the model halfway through, model will not manage to classify all the dots correctly. However, overall result is satisfactory as model skips only a small percentage of dots



Partially trained model



Trained model

Computational costs analysis

- Single feed-forward operation computational costs involve computing of activations of each layer from the input layer to the output layer
- For one layer weight matrix has a size of $N_{neuron} \cdot N_{neuron}$ and activation vector of size N_{neuron}
- Process of matrix multiplication of weights multiplied by activation vector involves $N_{neuron} \cdot N_{neuron}$which means complexity of N_{neuron}^2
- As we have N_{layer} time complexity scales as $N_{neuron}^2 \cdot N_{layer}$
- In the task we have assumed that N_{neuron} and N_{layer} are large enough, so we can neglect N_{layer} complexity against N_{neuron}^2which means, that final complexity for single feed forward operation is N_{neuron}^2
- Reference:
- https://jingyuexing.github.io/Ebook/Machine_Learning/Neural%20Networks%20and%20Deep%20Learning-eng.pdf
- http://users.isr.ist.utl.pt/~wurmd/Livros/school/Bishop%20-%20Pattern%20Recognition%20And%20Machine%20Learning%20-%20Springer%20%202006.pdf

Computational costs analysis

- Computing derivatives using Finite Differencing method involves approximating the derivative of the cost function with the respect to each w_i by slightly perturbing the it and performing two forward passes.

- Process of matrix multiplication of weights multiplied by activation vector is similar to the feed-forward algorithms and involves $N_{neuron} \cdot N_{neuron}$ which means number of weights equal to N_{neuron}^2. If we multiply it by N_{layer} we'll get total number of weights $N_{neuron}^2 \cdot N_{layer}$.

- As we need two forward passes total cost might be expressed as

$$Cost_{fin\_diff} = 2 \cdot N_{weights} \cdot Cost_{forward}$$

- Recalling feed-forward cost from the previous derivation:

$$Cost_{fin\_diff} = 2 \cdot \left(N_{neuron}^2 \cdot N_{layer}\right) \cdot \left(N_{neuron}^2 \cdot N_{layer}\right) = N_{layer}^2 \cdot N_{neuron}^4$$

Computational costs analysis

- Back-propagation method propagates error signals backwards through the network, allowing us to compute all derivatives with approximately the same computations costs as feed-forward algorithm.

- For backpropagation we have similar one layer weight matrix with a size of $N_{neuron} \cdot N_{neuron}$ and activation vector of size N_{neuron}

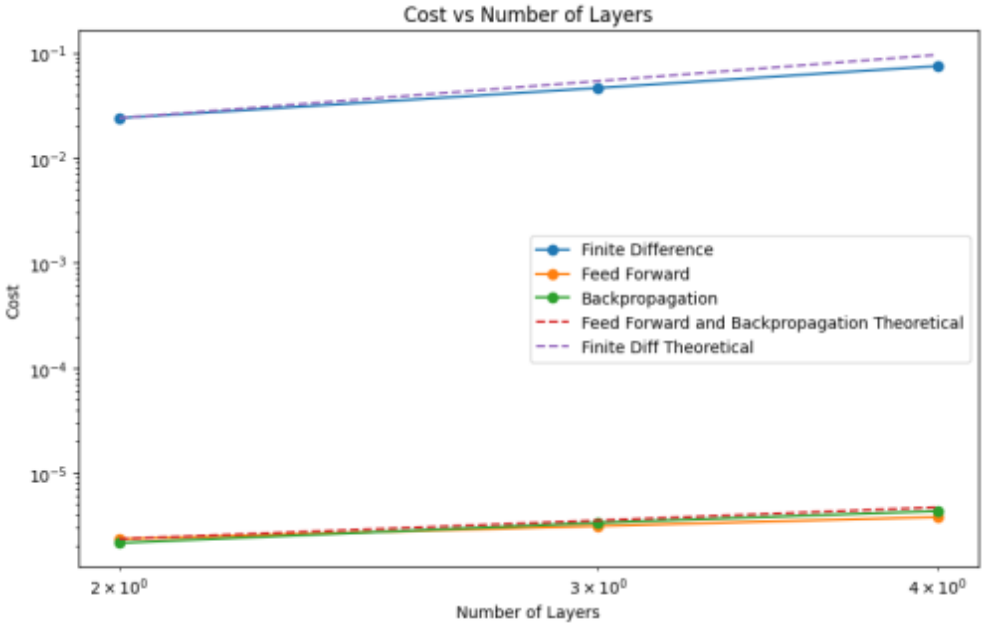
- Process of matrix multiplication of weights multiplied by activation vector involves $N_{neuron} \cdot N_{neuron}$which means complexity of N_{neuron}^2

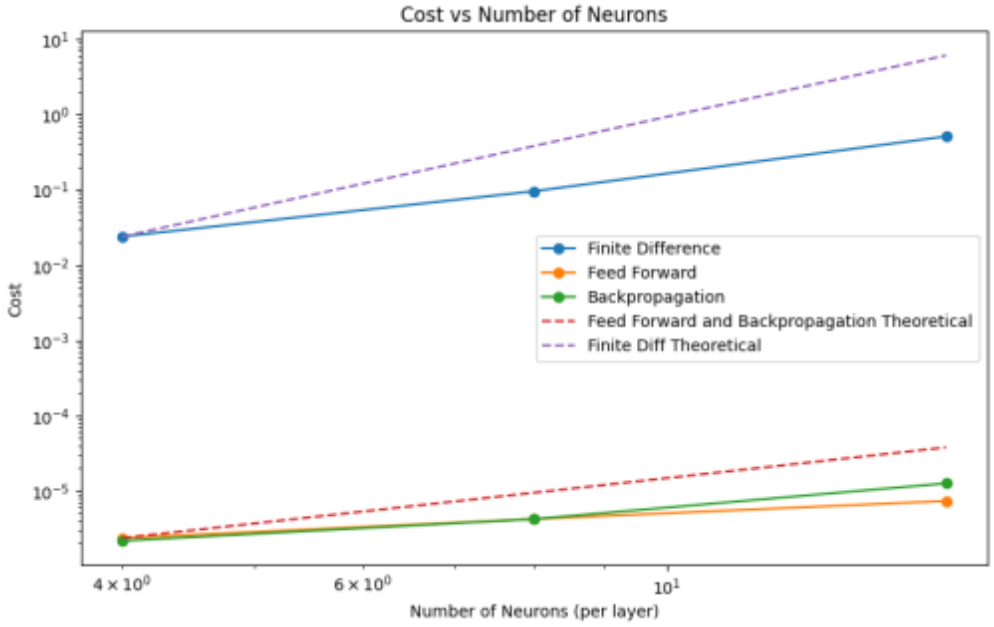
- As we have N_{layer} involved time complexity scales as $N_{neuron}^2 \cdot N_{layer}$. Which is equal to N_{neuron}^2 with N large enough

- Reference:
- https://jingyuexing.github.io/Ebook/Machine_Learning/Neural%20Networks%20and%20Deep%20Learning-eng.pdf
- http://users.isr.ist.utl.pt/~wurmd/Livros/school/Bishop%20-%20Pattern%20Recognition%20And%20Machine%20Learning%20-%20Springer%20%202006.pdf

Computational costs analysis

- Actual results are close to theoretical computational costs mentioned in previous slides as might be observed from the loglog plot.
- Finite difference method's costs increase more rapidly with the additional layers and neurons. Which means that the costs will grow quickly if we increase the size of the model
- Backpropagation and feed-forward algorithm demonstrate the same cost which is approximately equal to N_{neuron}^2





Computational costs improvement

- Current approach requires cost derivatives computation and updating weights and biases to be two separate processes
- However, if we interlace these steps, we might compute gradients and update corresponding weights and biases simultaneously
- This approach might reduce memory usage due to lack of separate gradient storage and reduce CPU usage due to lower number of data passes during backpropagation
- To implement this process, we might apply updates to the layer's weights and biases right after delta for a layer is computed.

Computational costs improvement

- If we run two versions of the script for the equal number of iterations (50000), we might observe acquired performance boost with similar training speed
- After 50000 iterations training speed reduced by more than 40%. However, increase of iterations might increase achieved speedup due to the memory efficient usage.