# Introduction

Neural networks are used to learn concepts in a domain by mapping the input dataset into a different dimensional space and then trying to classify the mapped data. The new dimension is determined by the number of hidden nodes, the mapping is done using the weights from the input to the hidden nodes. The network tries to learn the concept by updating it's weights using the errors observed in the training dataset, the errors are usually reduced using gradient descend; once the errors are minimized, the network is tested using the test set (or can be used to classify real-life scenarios where the output is to be determined).

# Implementation details

## Weight updates

I used block updates to update the weight vectors. For each epoch, I accumulated the error for each input sample and updated the weights at the end of the epoch.

## Momentum

I used momentum in my implementation; for each weight update, I considered the change in weight and used a fraction (tunable, I used around 5% to 10%) of it for updating in the next epoch. This way we accelerate towards the minima and try to overcome any local minima.

## Handling discrete attribute

I used 'n' input nodes (where 'n' is number of values the attribute takes), each input node representing if the attribute takes a particular value. I also experimented using log(n) nodes (as described in the problem), however the classification accuracy wasn't affected significantly.

I also considered single input node for a discrete valued attribute, having values that are equally spaced between 0 & 1, but this brings in a complication where the neighboring values are similar, whereas they aren't in reality. For example, consider 5 values for a discrete attribute, if we assign the values 0, 0.25, 0.5, 0.75 & 1 to various values, then we imply that 0 & 0.25 are similar against 0 & 1, which is not what the attribute intends. I hence decided not to use this method.

## Normalization

All the attributes' values (discrete/continuous) were linearly scaled between -1 & +1. For discrete attributes, since we had 'n' inputs (each taking values 0 or 1), all ones were normalized to '+1' and all zeros were normalized to '-1'. Normalization was done so as to avoid a case when the change in value of a particular attribute affects the output more than that of another attribute; the data without normalization would converge as well but it would take a longer number of epochs (the weights corresponding to higher valued input node should be reduced to match the other attributes' impact).

## Threshold function

I used sigmoid function as the threshold function.

## Stopping criteria

I let the algorithm update weights for 500 epochs, then whenever there isn't a drop in MSE for the number of epochs equaling the epoch when least MSE was observed, then I stop the algorithm. I also set the upper epoch limit to 1,000,000. The initial 500 epochs were chosen, considering the fact that the errors are erratic initially and letting it stabilize before we stop.

### Pseudo code

1. while numEpochs < minEpochs
   1. train the network updating weights
2. minErrorEpoch = minEpochs
3. while numEpochs < 1,000,000
   1. train the network updating weights
   2. if currentMSE < minMSE
      1. minMSE = currentMSE
      2. minErrorEpoch = numEpochs
   3. if numEpochs >= 2 \* minErrorEpoch
      1. break the while loop
   4. numEpochs++

## Data representation

I represented data in matrix format so that the computations are simpler (multiplying input vector with weight matrix gives us the output vector, etc.). I also created helper methods for matrix operations. The input matrix is such that each row represents a single input and each column corresponding to various attributes. Similarly the output matrix has in each row, the expected output activations of each output node for the corresponding input. Weight matrix is such that, weight[i][j][k] represents the weight between node 'j' in layer 'i' & node 'k' in layer 'i+1' (input is considered layer 0 and output is considered the last layer) (j & k are 0-based indexes of nodes taken in order).

## Classification

Since we try to learn a classification problem, I have 'm' output nodes, each corresponding to a class, whichever node has the highest activation, the corresponding class is the classification result of the neural network.

## Classification Accuracy

I used 10-fold cross-verification to report the classification accuracy. I split the data into 10 subsets, used 1 subset (in turns) to test the trained network; used 70% of the rest (90%) to train the network and the rest (30% of 90%) to compute the MSE of the trained network. I used random ordering to group the input (used rand() function to determine which set an input is to be added).

## Parameters

The implementation is such that, all the parameters (learning rate, momentum rate, minimum number of epochs before termination, number of hidden layers, number of hidden nodes) are tunable parameters (the values for each are assigned to the corresponding variable).

## Number of hidden nodes

I felt the number of hidden nodes should be parameter of the number of input nodes (since we are trying to re-represent the data in some-other dimension), however the order could depend on the complexity of the concept and each attribute could contribute to different order. Hence I experimented with various fractions of different orders of the number of input nodes, it resulted in good outcomes.

# Results

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Dataset** | **Neural Network accuracy** | **95% confidence interval** | **ID3 accuracy** | **95% confidence interval** | **p value from T-test** | **eeta (learning rate)** | **alpha (momentum rate)** | **Number of hidden layers** | **Number of hidden nodes** |
| **Car Evaluation** | 92.89 | 0.9093 to 0.9484 | 90.9063 | 0.8618 to 0.9208 | 0.0965 | 0.03 | 0.05 | 0 | NA |
| **Iris** | 100 | 1 to 1 | 84.2302 | 0.7566 to 0.9762 | 0.0045 | 0.03 | 0.05 | 1 | 5 |
| **Chess (King-Rook vs. King-Pawn)** | 54.85 | 0.5206 to 0.5763 | 99.1217 | 0.9877 to 0.9989 | 0.00001 | 0.1 | 0.1 | 2 | 50,20 |
| **Tic-Tac-Toe Endgame** | 100 | 1 to 1 | 83.9206 | 0.8125 to 0.902 | 0.00001 | 0.05 | 0.05 | 0 | NA |
| **Wine** | 100 | 1 to 1 | 59.4167 | 0.5012 to 0.7861 | 0.00001 | 0.03 | 0.05 | 1 | 5 |

95% confidence interval (in the context of our problem) is a range of probability values for an arbitrary sample to get correctly classified; the probability (for classification) will be in this range with a probability of 95%.

## Analysis

The Chess dataset was having more MSE (~0.5) with single hidden layer, while it marginally reduced (~0.45) when using 2 hidden layers. The 'Car Evaluation' and 'Tic-Tac-Toe' datasets produced poor accuracies (~60%) with hidden layers while they performed very well without any hidden layers. The other two datasets produced good accuracies with single hidden layer of 5 nodes. It could be inferred that 'Car Evaluation' and 'Tic-Tac-Toe' are relatively easier to learn hence they didn't need any hidden layers, while the Chess dataset couldn't produce good accuracy with 0, 1, or 2 hidden layers (I tried various combinations of hidden nodes, alpha, eeta). I was hoping Chess dataset to perform better with increased number of hidden nodes (since there are more attributes involved), but that didn't turnout to be the case. The Chess dataset had a similar MSE (~0.5) without any hidden layer or with just 1 hidden layer. Hence there wasn't any convergence with this case, while tweaking the parameters.

The increased number of hidden layers or hidden nodes slowed down the computations, which is understandably due to more computations involved in tuning the weights. Though the training of Neural Networks could be very slow, it's accuracy in general is good and the runtime on test samples is pretty quick.

We could see that decision trees performed very good (~100% accuracy) for Chess dataset, while Neural networks sort of failed in its attempt. One reasoning to this could be that, since this dataset is a more predictable data, there were few attributes that had very good information-gain (in ID3), leading to good accuracy with decision trees; while Neural networks could have failed to understand the high dimensional (~70) attributes.

From the T-tests conducted, we could see that there were significant improvements for 'Iris', 'Tic-tac-toe', and 'Wine' datasets. Also there was a significant drop in classification accuracy (for the reasons discussed earlier) with the 'Chess' dataset. And there wasn't a significant change in accuracy for the 'Car evaluation' dataset, since both the algorithms performed fairly well on this dataset.

## Experimenting with different number of hidden layers

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Dataset** |  | **0 hidden layer** | **1 hidden layer** | | | | **2 hidden layers** | | | | **eeta (learning rate)** | | **alpha (momentum rate)** |
| **Car Evaluation** | **Number of hidden nodes** | - | 5 | 10 | 15 | 5,5 | | 5,10 | 10,5 | 0.03 | | 0.05 | |
| **Accuracy** | 92.89 | 71.03 | 71.03 | 71.03 | 71.03 | | 71.03 | 71.03 |
| **Tic-Tac-Toe Endgame** | **Number of hidden nodes** | - | 5 | 10 | 15 | 5,5 | | 5,10 | 10,5 | 0.05 | | 0.05 | |
| **Accuracy** | 100 | 62.85 | 62.85 | 62.85 | 62.85 | | 62.85 | 62.85 |
| **Wine** | **Number of hidden nodes** | - | 5 | 10 | 15 | 5,5 | | 5,10 | 10,5 | 0.03 | | 0.05 | |
| **Accuracy** | 100 | 100 | 100 | 100 | 100 | | 100 | 100 |

We could observe that with the presence of hidden layers, 'Car Evaluation' datasets accuracies were about 70%, which is close to it's majority classifier's accuracies; hence it classifies all the data to the majority class. Similarly, we can see that 'Tic-Tac-Toe' performed similar to its majority classifier when hidden layers were used. However, 'Wine' dataset performed with 100% accuracy irrespective of the presence/absence of the hidden layers, signifying that it is should be a simple concept to learn.

Hence increased number of hidden layers could be good for a few concepts (like Chess), while it could also not be good for few datasets (Car & Tic-Tac-Toe); we'll have to understand the concept to learn and experiment with various configurations to obtain the best accuracy for that dataset. I was hoping that more the hidden nodes, better the accuracy would be, but the results proved me wrong. One reasoning behind this could be that more hidden layers tend to over-fit data, there by failing to classify similar values correctly. Hence we can't generalize the number of hidden nodes or number of layers for all datasets, each dataset is unique in its own sense and different parameters would work well for it and may be only for it!

/////std. devs. for ID3 wrt. pruning:

3.281

12.051

0.7517

4.008

12.737