# MLP on MNIST Dataset

## Introduction

The MNIST dataset is a widely, commonly used database within the field of machine Learning. It is often used to test different machine learning methods and, can testing your algorithm through this dataset may been seen as an industry standard. The goal when utilising this dataset is to train an algorithm to spot the difference between handwritten characters and separate them correctly into the 10 different classifiers, i.e., 0 – 9.

*The set of images in the MNIST database was created in 1998 as a combination of two of*[*NIST*](https://en.wikipedia.org/wiki/NIST)*'s databases: Special Database 1 and Special Database 3. Special Database 1 and Special Database 3 consist of digits written by high school students and employees of the*[*United States Census Bureau*](https://en.wikipedia.org/wiki/United_States_Census_Bureau)*, respectively[1]*

The database consists of two different datasets, trainingdata[images,labels] and testdata[images,labels]. Training data is split into 60,000 datapoints and testdata is split into 10000 entries. The data is a breakdown of a 28x28 image consisting of a centred handwritten digit ranging from 0 – 9. Each pixel is converted to binary and represented within a matrix.

### MNIST versus IRIS Dataset

The obvious difference between these datasets is the complexity of algorithm required to handle the various dimensions. IRIS, the simpler dataset, consists of 4 different features septal length, septal width, petal width & petal length. These features once processed with be used to guess one of 3 different classes of flower. Whereas the MNIST dataset consists of 784 features, each feature being the binary value of a pixel on a 28X28 image. Once processed the features will be used to guess 1 out of 10 classes(digits 0-9).

### Effective Algorithm Performance

The MNIST dataset has been used to train different Machine learning models such as KNN, linear classification and neural networks. According to *LeCun et al. 1998 applying a linear classifier with a single neural network has an error rate of 12% [2]* after applying *pre-processing methods such as deskewing, drops the error rate to 8.4%[2].* It is also stated that using a *neural network with 2 hidden layers, 300 hidden units, MSE & no pre-processing produces an error rate of 4.7%[2].* Our model closely resembles this, and I hypothesise that our model will produce a similar error rate.

## Building MLPs

I have successfully built both a single hidden layer Multi layered perception and Multiple hidden layers. These have been run against the XOR problem.

## Building MLP with one hidden layer

Expected output when running the base MLP.m file within testMLPTrain131.m Weights are not initialised; Forward propagation is just setting the values to a matrix of zeros. Backwards propagation is currently empty.  
Graphical user interface, application, Word

Description automatically generated

### Forward Propagation

Forward propagation requires the following method to be effective.

* Give an initial set of input values, these inputs will also include a bias that been hardcoded to be 1.
* assign associated weights. These weights will be used to calculate the activation value for each hidden unit(node) by doing the following matrix(weights) \* matrix(inputs)**.**
* Activation value is then multiplied by the sigmoid function. There were alternative functions such as SoftMax.

Graphical user interface, application

Description automatically generatedThe weights must be assigned to the MLP before beginning forward propagation. To assign the associated weights was used, standard deviation \* randommatrix + 1. Timing by standard deviation controls the spread of the values.   
Once again, I tested my implementation under testMLPtrain131.m. I am now able to see an output being calculated and shown however it’s not updating or changing. No learning takes place, so the output value is repeated.

Chart, scatter chart

Description automatically generated

### Backwards Propagation

To further the development of our MLP I must implement backwards propagation (Backward propagation of errors). To achieve this, weights in the MLP are updated using gradient decent, the aim of this is to have the next set of inputs to be multiplied by a set of weights that more closely resemble our desired output. By updating the weights depending on input it makes the algorithm learn.

Our MLP was initialised with the following: 0.05 learning Rate, three hidden layers, two inputs, and a single output. Data was iterated 1000 times.   
  
The learning rate was set this low so that our algorithm can do smaller jumps. Each jump is a guess upon where expected output is.

Chart

Description automatically generated

A picture containing graphical user interface

Description automatically generatedInitial Output Finished Output

Graphical user interface, application

Description automatically generated

### Multiple Hidden Layer Propagation

To advance the currently implemented MLP into for multilayer propagation. Some matrixes i.e., hiddenLayerWeights will be kept inside a Vector to produce a 3D space. Within the 3D space our algorithm can iterate through the vector and grab a set of values within the matrix. This allows us to iterate through multiple hidden layers while keeping the matrix sizes the same.   
  
The Vector was added to allow indexing, yet other adaptations to our code must be made to allow for forward and backpropagation to go through multiple hidden layers.

#### Properties & Methods

To iterate the chosen number of times, hidden layers was added to the Properties and initialised within methods. hiddenLayerWeights was wrapped within a vector for iteration.

#### Forward Propagation

A for loop added inside of the forward propagation so that it could calculate the activation value for multiple hidden layers.

#### Backwards Propagation

To successfully calculate backwards propagation with multiple hidden layers. A split had to be made to ensure that that the algorithm took the correct weights. A for loop was added splitting backwards propagating into three distinct scenarios.

* 1. First Hidden Layer
  2. If the Hidden layer is set to one
  3. Every other hidden layer

#### MLPVis

To allow MLPVis.m to visualise our XOR we had to initialise the hidden layers & assign them within the mlp.

Chart, scatter chart

Description automatically generatedGraphical user interface, chart, scatter chart

Description automatically generatedAfter applying these adaptations, the visualisation is as follows:

## Experimentation

The following analysis is based upon how effectively our algorithm correctly classifies a digit from the testing dataset.

### Pre-processing the Data

The MNIST dataset is already processed piece of data, where each image has been size-normalized and centred in a fixed-size image. To add to this pre-processing, I have added a few checks upon the data to see if it’s the correct dataset.   
  
There are two checks in place to warn us about loading in faulty or corrupted data. The first test checks all labels to see if they are a value between 0 – 9. Any values detected outside of this means that the dataset is untrustworthy. Additionally, I am checking all images to see if the binary value of the pixel exceeds one or is less than zero. If the data fails this check, its once again untrustworthy.   
  
All datasets of the same class are combined and fed through these tests. I.e., all images, all labels.

### Train Data, Training error

The entire training dataset consisting of 60,000 images & labels are being processed by the MLP. Each time the dataset has been completed; backward propagation will ensure it is learning. Testing the same data again will produce the training error. This is done within a single loop so that the data isn’t randomised again.

### Test Data, Test error

The MNST dataset separates the data into test and training data. The test data consisting of 10,000 images & labels are processed by the algorithm, after it has been trained by the training data. Error rate is calculated by having a predicted value and actual value. A difference between the two will be counted towards the error.

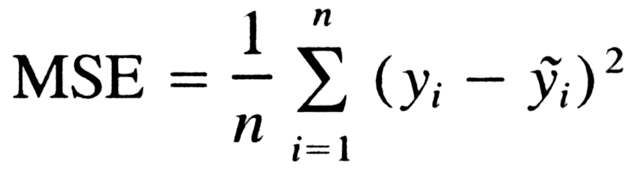
### KFold Cross validation

Kfolds is where the data is separated into equal distinct datasets i.e. 0 – 2000 would be a kfold of the overall dataset consisting of 10,000 values. Within the algorithm, the test data is separated into 5 different folds. For each fold we test the entirety of the dataset. Each time we run through a Kfold we get the MSE value. At the end we add up mean squared error and divide by the dataset length to get a fair error rate. The closer the MSE is to zero the better preforming our algorithm is.

N – number of data

Y – expected value

Ὗ - predicted value



### Test Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test | Learning Rate | Number of hidden layers | Number of Hidden units | Error percentage Train % | Error Percentage Test % | MSE |
| 1 | 0.25 | 2 | 75 | 2% | 4.72 | 0.8118 |
| 2 | 0.8 | 2 | 75 | 16.9583% | 17.17 % | 3.0957 |
| 3 | 0.5 | 2 | 75 | 3.88% | 5.84% | 1.0458 |
| 4 | 0.3 | 2 | 75 | 1.8983% | 4.14% | 0.759 |
| 5 | 0.35 | 2 | 75 | 2.5083% | 4.35% | 0.7989 |
|  |  |  |  |  |  |  |
| 6 | 0.3 | 2 | 75 | 2.0883% | 4.43% | 0.8533 |
| 7 | 0.3 | 3 | 75 | 3.2483% | 5.43% | 0.914 |
| 8 | 0.3 | 8 | 75 | 38.9267% | 47.99% | 9.9482 |
| 9 | 0.3 | 4 | 75 | 5.8867% | 6.86% | 1.1554 |
| 10 | 0.3 | 5 | 75 | 7.67% | 9.74% | 1.526 |
| 11 | 0.3 | 6 | 75 | 13.3817% | 14.73% | 2.3529 |
|  |  |  |  |  |  |  |
| 12 | 0.3 | 2 | 20 | 3.4467% | 5.72% | 0.9665 |
| 13 | 0.3 | 2 | 50 | 2.0617% | 4.71% | 0.81 |
| 14 | 0.3 | 2 | 75 | 2.0333% | 4.96% | 0.8633 |
| 15 | 0.3 | 2 | 100 | 11.6533% | 13.52% | 2.1703 |
| 16 | 0.3 | 2 | 150 | 21.3% | 22.51% | 4.4211 |
| 17 | 0.3 | 2 | 200 | 30.6017% | 27.8% | 4.5295 |

Tests 1 – 5

Tests 1 – 5, I experimented upon different learning rates while keeping the same number of hidden layers and hidden units. Initially I measured extremally different learning rates of 0.25 and 0.8. These have a considerable difference in ‘jump’ length for gradient descent. From the results it seems that a lower learning rate is favoured and has the best chance of giving the expected outcome. After testing multiple learning rates 0.25 + 0.3 produced the lowest error rate, with 0.3 being the lowest.

Tests 6-10

Tests 6 – 10 was to find the optimum number of hidden layers. Once again, I began the tests with a low value and a higher one to determine whether a higher number of hidden layers lowered the error rate. It became clear that by increasing the number of hidden layers increases the error rate for both test & train data. 2 Hidden layers meant that our algorithm has highest number of correct guesses , lowest error rating. During these tests I kept the optimum learning rate.

Tests 11-16

Tests 11 – 16. Here I experimented with a different number of hidden units. During these tests I kept the previous optimum learning rate at 0.3 & HiddenLayers at 2. The goal for these tests was to find the lowest error rate depending on the number of hidden units. I steadily increased the number of hidden layers to determine a pattern with the error rating. From my data it’s clear that when the number of hidden units increases so does the error rating. With a substantial increase from 100 to 150 & 150 to 200. The optimal hidden units were 50.

### Kfold

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test | Fold1 Correct guesses | Fold2 Correct guesses | Fold3 Correct guesses | Fold4 Correct guesses | Fold5 Correct guesses | Average : |
| 1 | 95.65% | 95% | 95.35% | 95.9% | 94.45% | 95.272% |
| 2 | 94.45% | 94.1% | 95.4% | 95.65% | 95.8% | 95.08% |
| 3 | 95.95% | 95.6% | 94.6% | 96% | 95.25% | 95.48% |
| 4 | 95.3% | 94.85% | 95.75% | 94.75% | 95.8% | 95.29% |
| 5 | 95% | 95.45% | 96.15% | 94.7% | 95.7% | 95.4% |

Kfold cross validation produced an average correct guess % of 95.3044%. The kfold validation was done from my best parameters, 0.3 learning rate, 2 hidden layers, 50 hidden units.

### Conclusion

The conclusion of our experiment, producing a multi-layered neural network is the following. From the test table above, we can make some conclusions. The optimal value for each parameter is as follows. Learning rate 0.3, MSE 0.759, as soon as the learning rate exceeded 0.3 the error rate drastically increased. Our data hints that values 0.5.. 0.6… 0.7.. etc. was causing the gradient descent to overshoot. Thus, increasing the error rate. Hidden layers optimum value was 2, 0.8533. It was clear from the test table that by increasing the number of hidden layers the error rate drastically increased. Finally for the algorithm our optimum number of hidden units was 50. From my testing, lower values than 50 started to increase the error rate. A possible explanation for this is that with that little amount of backwards propagation the algorithm isn’t learning enough. Exceeding 100 hidden units drastically increased the error % (lower accuracy). A potential explanation for this rapid increase is the model is being overfitted to the training data.

From comparing this the most closely related model from [LeCun et al. 1998](http://yann.lecun.com/exdb/mnist/). They created a neural network with the following, 2-layer NN, 300 hidden units and this produced an error rating of 4.7%. Our algorithms best case, 2 hidden layers, 50 hidden units & a learning rate of 0.3. Our test error rate was 4.71%. While this can be seen as positive as we almost matched the learning rate of the Neural network. The parameters for the algorithm were completely different and seem to go against our model. As they used 6 times the hidden units.

Comparing our choice of neural network to another method done by lecun et al. 1998 such as linear classifier. From his findings the less complex algorithm produces a test error rate of 12%. Even with pre-processing data methods such as deskewing. The error rate remains higher at 8.4%. From this it may be concluded that the choice of a neural network is recommended for accurately classifying the MNIST dataset.  
To improve our algorithm further data pre-processing may be done upon the MNIST dataset.   
While the MNIST data set already has a few methods to process the data, I believe further implementation of pre-processing data would improve our results. Such as deskewing or noise removal. Additionally adding further complexity to our learning rate so that it adjusts itself nearing the global cost minimum. Would heavily increase algorithms accuracy.

The results taken from the test table were obtained after repeating the value 3 times and taking an average. While repeating the test improves the reliability of the dataset. Only repeating 3 times may not give us a full indication of the result. Further testing would improve reliability of the test data.

References:

1. <https://en.wikipedia.org/wiki/MNIST_database>
2. <http://yann.lecun.com/exdb/mnist/>

CODE:

MNIST

function MNIST

% Setting the Datasets

clear all

close all

Trainimages = loadMNISTImages('train-images-idx3-ubyte');

Trainlabels = loadMNISTLabels('train-labels-idx1-ubyte');

Testimages = loadMNISTImages('t10k-images-idx3-ubyte');

Testlabels = loadMNISTLabels('t10k-labels-idx1-ubyte');

combinedImages = [Trainimages Testimages];

combinedLabels = [Trainlabels; Testlabels];

%% Data Processing

labelcheck = 0;

%Checks for Correct Label information

for i=1:size(combinedLabels)

label = combinedLabels(i);

if label == 0 || label == 1 || label == 2 || label == 3 || label == 4 || label == 5 || label == 6 || label == 7 || label == 8 || label == 9

labelcheck = 1;

else

labelcheck = 0;

disp("This dataset contains incorrect labels and may not be trusted")

break;

end

end

if labelcheck == 1;

disp("The dataset only contains Labels from 0 - 9. It may be used for our MLP")

end

imagecheck = 0;

for i=1:size(combinedImages)

pixelValue = combinedImages;

if pixelValue >= 0 | pixelValue < 1

imagecheck = 1;

else

disp("This dataset contains incorrect pixel values and should not be trusted")

break;

end

end

if imagecheck == 1;

disp("The dataset associated images pixel values are within the correct range. It may be used for our MLP")

end

fprintf("\n")

%% Data manipulation

[pixelcount, allimages] = size(Trainimages); %Combines the dataset

trainDataLimit = 60000;

% Randomize the training data %

indexing = randperm(trainDataLimit);

Trainimages = Trainimages(:,indexing);

Trainlabels = Trainlabels(indexing);

testdatalimit = 10000;

indexing = randperm(testdatalimit);

Testimages = Testimages(:,indexing);

Testlabels = Testlabels(indexing);

%%

%MLP + weights

m = MLP(pixelcount, 50, 10, 2); %Number of inputs, Number of hidden units, Number of outputs, Number of hidden layers

m = m.initializeWeightsRandomly(1.0);

learningrate = 0.3;

%%

% %Training the Dataset + training error

comparematrix = zeros(10,1);

errorrate = 0;

count = 0;

for i = 1:10

%Training the Dataset

for allimages=1:length(Trainimages)

a = Trainlabels(allimages,:); %index Value between 0 - 1

comparematrix(a+1) = 1;

m.train\_single\_data(Trainimages(:, allimages), comparematrix, learningrate);

[value,index] = max(m.compute\_output(Trainimages(:, allimages)));

if a == index-1

count = count + 1;

else

errorrate = errorrate +1;

end

comparematrix=(zeros(10,1));

end

fprintf("Train DATA \n")

disp("You have correctly guessed: " + count + " " + count/60000\*100 + "%");

disp("You have incorrectly guessed: " + errorrate + " " + errorrate/60000\*100 +"%");

fprintf("\n")

errorrate = 0;

count = 0;

end

testerror = 0;

countt = 0;

%Error Rate for TestData

for i=1:length(Testimages)

prediction = m.compute\_output(Testimages(:, i));

[Max,Index] = max(prediction);

if(Index-1 == Testlabels(i))

countt = countt + 1;

else

testerror = testerror+1;

end

end

fprintf("Test DATA \n")

disp("You have correctly guessed: " + countt + " " + countt/length(Testimages)\*100 + "%")

disp("You have incorrectly guessed: " + testerror + " " + testerror/length(Testimages)\*100 + "%")

fprintf("\n")

%%

%K - Folds

Kfold1 = Testimages(:,1:2000);

Kfold2 = Testimages(:,2001:4000);

Kfold3 = Testimages(:,4001:6000);

Kfold4 = Testimages(:,6001:8000);

Kfold5 = Testimages(:,8001:10000);

Klabel1 = Testlabels(1:2000,:);

Klabel2 = Testlabels(2001:4000,:);

Klabel3 = Testlabels(4001:6000,:);

Klabel4 = Testlabels(6001:8000,:);

Klabel5 = Testlabels(8001:10000,:);

counter = 0;

mse = 0;

totalcounter = 0;

[k1,mse1] = workfold(Kfold1,Klabel1);

[k2,mse2] = workfold(Kfold2,Klabel2);

[k3,mse3] = workfold(Kfold3,Klabel3);

[k4,mse4] = workfold(Kfold4,Klabel4);

[k5,mse5] = workfold(Kfold5,Klabel5);

mseerror = ((mse1+mse2+mse3+mse4+mse5)/10000);

fprintf("KFolds \n")

disp("Kfold1 correct guessed: " + k1 + ", " + k1/length(Kfold1)\*100 + "%")

disp("Kfold2 correct guessed: " + k2 + ", " + k2/length(Kfold1)\*100 + "%")

disp("Kfold3 correct guessed: " + k3 + ", " + k3/length(Kfold1)\*100 + "%")

disp("Kfold4 correct guessed: " + k4 + ", " + k4/length(Kfold1)\*100 + "%")

disp("Kfold5 correct guessed: " + k5 + ", " + k5/length(Kfold1)\*100 + "%")

disp("MultiSquared Error is: " + mseerror)

%%

%Kfold Calculation

function [counter, mse] = workfold(foldimage,foldlabel)

counter = 0;

mse = 0;

m.train\_single\_data(Trainimages(:, allimages), comparematrix, learningrate);

for t=1:length(foldimage)

predic = m.compute\_output(foldimage(:, t));

[M,I] = max(predic);

if(I-1 == foldlabel(t))

counter = counter + 1;

end

mse = mse + ((I-1) - foldlabel(t))^2;

end

end

end

MLP

function MNIST

% Setting the Datasets

clear all

close all

Trainimages = loadMNISTImages('train-images-idx3-ubyte');

Trainlabels = loadMNISTLabels('train-labels-idx1-ubyte');

Testimages = loadMNISTImages('t10k-images-idx3-ubyte');

Testlabels = loadMNISTLabels('t10k-labels-idx1-ubyte');

combinedImages = [Trainimages Testimages];

combinedLabels = [Trainlabels; Testlabels];

%% Data Processing

labelcheck = 0;

%Checks for Correct Label information

for i=1:size(combinedLabels)

label = combinedLabels(i);

if label == 0 || label == 1 || label == 2 || label == 3 || label == 4 || label == 5 || label == 6 || label == 7 || label == 8 || label == 9

labelcheck = 1;

else

labelcheck = 0;

disp("This dataset contains incorrect labels and may not be trusted")

break;

end

end

if labelcheck == 1;

disp("The dataset only contains Labels from 0 - 9. It may be used for our MLP")

end

imagecheck = 0;

for i=1:size(combinedImages)

pixelValue = combinedImages;

if pixelValue >= 0 | pixelValue < 1

imagecheck = 1;

else

disp("This dataset contains incorrect pixel values and should not be trusted")

break;

end

end

if imagecheck == 1;

disp("The dataset associated images pixel values are within the correct range. It may be used for our MLP")

end

fprintf("\n")

%% Data manipulation

[pixelcount, allimages] = size(Trainimages); %Combines the dataset

trainDataLimit = 60000;

% Randomize the training data %

indexing = randperm(trainDataLimit);

Trainimages = Trainimages(:,indexing);

Trainlabels = Trainlabels(indexing);

testdatalimit = 10000;

indexing = randperm(testdatalimit);

Testimages = Testimages(:,indexing);

Testlabels = Testlabels(indexing);

%%

%MLP + weights

m = MLP(pixelcount, 50, 10, 2); %Number of inputs, Number of hidden units, Number of outputs, Number of hidden layers

m = m.initializeWeightsRandomly(1.0);

learningrate = 0.3;

%%

% %Training the Dataset + training error

comparematrix = zeros(10,1);

errorrate = 0;

count = 0;

for i = 1:10

%Training the Dataset

for allimages=1:length(Trainimages)

a = Trainlabels(allimages,:); %index Value between 0 - 1

comparematrix(a+1) = 1;

m.train\_single\_data(Trainimages(:, allimages), comparematrix, learningrate);

[value,index] = max(m.compute\_output(Trainimages(:, allimages)));

if a == index-1

count = count + 1;

else

errorrate = errorrate +1;

end

comparematrix=(zeros(10,1));

end

fprintf("Train DATA \n")

disp("You have correctly guessed: " + count + " " + count/60000\*100 + "%");

disp("You have incorrectly guessed: " + errorrate + " " + errorrate/60000\*100 +"%");

fprintf("\n")

errorrate = 0;

count = 0;

end

testerror = 0;

countt = 0;

%Error Rate for TestData

for i=1:length(Testimages)

prediction = m.compute\_output(Testimages(:, i));

[Max,Index] = max(prediction);

if(Index-1 == Testlabels(i))

countt = countt + 1;

else

testerror = testerror+1;

end

end

fprintf("Test DATA \n")

disp("You have correctly guessed: " + countt + " " + countt/length(Testimages)\*100 + "%")

disp("You have incorrectly guessed: " + testerror + " " + testerror/length(Testimages)\*100 + "%")

fprintf("\n")

%%

%K - Folds

Kfold1 = Testimages(:,1:2000);

Kfold2 = Testimages(:,2001:4000);

Kfold3 = Testimages(:,4001:6000);

Kfold4 = Testimages(:,6001:8000);

Kfold5 = Testimages(:,8001:10000);

Klabel1 = Testlabels(1:2000,:);

Klabel2 = Testlabels(2001:4000,:);

Klabel3 = Testlabels(4001:6000,:);

Klabel4 = Testlabels(6001:8000,:);

Klabel5 = Testlabels(8001:10000,:);

counter = 0;

mse = 0;

totalcounter = 0;

[k1,mse1] = workfold(Kfold1,Klabel1);

[k2,mse2] = workfold(Kfold2,Klabel2);

[k3,mse3] = workfold(Kfold3,Klabel3);

[k4,mse4] = workfold(Kfold4,Klabel4);

[k5,mse5] = workfold(Kfold5,Klabel5);

mseerror = ((mse1+mse2+mse3+mse4+mse5)/10000);

fprintf("KFolds \n")

disp("Kfold1 correct guessed: " + k1 + ", " + k1/length(Kfold1)\*100 + "%")

disp("Kfold2 correct guessed: " + k2 + ", " + k2/length(Kfold1)\*100 + "%")

disp("Kfold3 correct guessed: " + k3 + ", " + k3/length(Kfold1)\*100 + "%")

disp("Kfold4 correct guessed: " + k4 + ", " + k4/length(Kfold1)\*100 + "%")

disp("Kfold5 correct guessed: " + k5 + ", " + k5/length(Kfold1)\*100 + "%")

disp("MultiSquared Error is: " + mseerror)

%%

%Kfold Calculation

function [counter, mse] = workfold(foldimage,foldlabel)

counter = 0;

mse = 0;

m.train\_single\_data(Trainimages(:, allimages), comparematrix, learningrate);

for t=1:length(foldimage)

predic = m.compute\_output(foldimage(:, t));

[M,I] = max(predic);

if(I-1 == foldlabel(t))

counter = counter + 1;

end

mse = mse + ((I-1) - foldlabel(t))^2;

end

end

end

MLP  
  
% Note: this file merely specifies the MLP class. It is not meant to be

% executed as a stand-alone script. The MLP needs to be instantiated and

% then used elsewhere, see e.g. 'testMLP131train.m'.

% A Multi-layer perceptron class

classdef MLP < handle

% Member data

properties (SetAccess=private)

inputDimension % Number of inputs

hiddenDimension % Number of hidden neurons

outputDimension % Number of outputs

hiddenLayerWeights % Weight matrix for the hidden layer, format (hiddenDim)x(inputDim+1) to include bias terms

outputLayerWeights % Weight matrix for the output layer, format (outputDim)x(hiddenDim+1) to include bias terms

hiddenNum %Number of hidden layers

end

methods

% Constructor: Initialize to given dimensions and set all weights

% zero.

function mlp=MLP(inputD,hiddenD,outputD,hiddenNum)

mlp.inputDimension=inputD;

mlp.hiddenDimension=hiddenD;

mlp.outputDimension=outputD;

mlp.hiddenLayerWeights={zeros(hiddenD,inputD+1)};

mlp.outputLayerWeights=zeros(outputD,hiddenD+1);

mlp.hiddenNum = hiddenNum;

end

function mlp=initializeWeightsRandomly(mlp,stdDev)

% Note: 'mlp' here takes the role of 'this' (Java/C++) or

% 'self' (Python), refering to the object instance this member

% function is run on

% mlp.hiddenLayerWeights = stdDev \* randn(mlp.hiddenDimension, mlp.inputDimension+1)

% mlp.outputLayerWeights = stdDev \* randn(mlp.outputDimension, mlp.hiddenDimension+1)

%Multiplying a random Matrix by a random Variance, the +1 adds

%the bias

%Randomised the input weights

mlp.hiddenLayerWeights{1} = stdDev \* randn(mlp.hiddenDimension, mlp.inputDimension+1);

%Randomised any weights thats not contained within output/input

for a=2:mlp.hiddenNum

mlp.hiddenLayerWeights{a} = stdDev \* randn(mlp.hiddenDimension, mlp.hiddenDimension+1);

end

%Randomise the Output Weights

mlp.outputLayerWeights = stdDev \* randn(mlp.outputDimension, mlp.hiddenDimension+1);

end

function [hidden,output]=compute\_forward\_activation(mlp, inputData)

activation = inputData;

for a = 1:mlp.hiddenNum

Activationhidden = mlp.hiddenLayerWeights{a} \* [activation; 1];

hidden{a} = mlp.sigmoidfunction(Activationhidden);

activation = hidden{a};

end

Activationoutput = mlp.outputLayerWeights \* [hidden{mlp.hiddenNum}; 1];

output = mlp.sigmoidfunction(Activationoutput);

end

% SigMoid function

function [output] = sigmoidfunction(mlp, i)

output = 1./(1+exp(-i));

end

% This function calls the forward propagation and extracts only the

% overall output. It does not have to be altered.

function output=compute\_output(mlp,input)

[~,output] = mlp.compute\_forward\_activation(input);

end

function mlp=train\_single\_data(mlp, inputData, targetOutputData, learningRate)

[h,o] = mlp.compute\_forward\_activation(inputData);

derivativek = (o - targetOutputData) .\* (o .\* (1-o));

mlp.outputLayerWeights = mlp.outputLayerWeights - (learningRate \* derivativek \* [h{mlp.hiddenNum}; 1]');

weights = transpose(mlp.outputLayerWeights(:, 1: end-1));

for a = (mlp.hiddenNum): -1 :1

if mlp.hiddenNum == 1 %If the hidden layer is one

derivativek = (weights \* derivativek) .\* (h{a} .\* (1-h{a}));

gradientupdate = derivativek \* transpose([inputData; 1]);

elseif a == 1 %First Hidden Layer

weights = transpose(mlp.hiddenLayerWeights{2}(:, 1: end-1));

derivativek = (weights \* derivativek) .\* (h{a} .\* (1-h{a}));

gradientupdate = derivativek \* transpose([inputData; 1]);

else

derivativek = (weights \* derivativek) .\* (h{a} .\* (1-h{a})); %Rest of the hidden Layers

gradientupdate = derivativek \* transpose([h{a-1}; 1]);

weights = mlp.hiddenLayerWeights{a}(:, 1: end-1);

end

mlp.hiddenLayerWeights{a} = mlp.hiddenLayerWeights{a} - (learningRate \* gradientupdate);

end

end

end

end

MLPVIS  
clear all

close all

% random seed

rng(2);

% specify target output function (represented by function pointers here)

target = @outputXOr

targetName='XOR'

%choose number of hidden neurons

hiddenSize = 3;

hiddenLayers = 4;

initRange = 1.0

% number of training steps between two plot renderings

speedUp = 1000;

record=0; % set to 1 to record video

% WARNING: these videos are /uncompressed/ at first and VERY LARGE

if record

mov(1:1)=struct('cdata',[],'colormap',[]);

frame=1;

title=['mlp-' targetName];

writerObj = VideoWriter([title '.avi'], 'Uncompressed AVI');

open(writerObj);

end

% create training data

Neach = 20;

Ntotal = 4\*Neach

% 0/1 values

X = [repmat([0;0],1,Neach) repmat([0;1],1,Neach) repmat([1;0],1,Neach) repmat([1;1],1,Neach)];

% plus noise

X = X + randn(2, Ntotal)\*0.1;

%... plus constant feature:

XF = [X; repmat([1],1,Ntotal)];

% outputs

Y = repmat(0, 1, Ntotal);

for i=1:Ntotal

Y(i) = target(X(:,i));

end

%show data (color chosen by labels in Y)

scatter(X(1,:), X(2,:), 25, Y, 'filled')

colormap('jet');

colorbar();

pbaspect([1 1 1]) % quadratic aspect ratio;

hold on;

% initialize MLP

m = MLP(2, hiddenSize, 1, hiddenLayers);

m = m.initializeWeightsRandomly(initRange);

% initialize hidden neuron visualization

for i=1:hiddenSize

points = perceptronBoundary(m.hiddenLayerWeights{1}(i,1:3), 0);

strength = m.outputLayerWeights(1,i);

boundary(i) = plot(points(1,:), points(2,:), 'LineWidth',2, 'Color', [0.5 0.5 0.5]);

end

% initialize output visualization (countours generareted through a grid of input values)

meshSize = 20;

as = linspace(-0.3,1.3,meshSize);

bs = linspace(-0.3,1.3,meshSize);

[A B] = meshgrid(as,bs);

E = zeros(meshSize,meshSize);

for i=1:meshSize

for j=1:meshSize

E(i,j) = m.compute\_output([A(i,j); B(i,j)]);

end

end

[colorMat outputContour] = contour(A,B,E);

caxis([-0.1 1.1]);

% enforce axes limits

xlim([-0.3 1.3]);

ylim([-0.3 1.3]);

% initialize marker indicating current data

marker = scatter([], [], 100, 's', 'filled');

hold off;

for t = 1:10000

for i=1:speedUp

% choose random sample from data

index = randi([1 Ntotal], 1, 1);

% evaluate MLP's output (fwd prop)

yest = m.compute\_output(X(:,index));

% perform learning step (back prop)

m.train\_single\_data(X(:,index), Y(index), 0.05);

end

% update visualizations

for i=1:hiddenSize

points = perceptronBoundary(m.hiddenLayerWeights{1}(i,1:3), 0);

strength = m.outputLayerWeights(1,i);

strength = 1+2\*sqrt(abs(strength));

if strength>15

strength = 15;

end

set(boundary(i), 'XData', points(1,:), 'YData', points(2,:), 'LineWidth',strength);

end

for i=1:meshSize

for j=1:meshSize

E(i,j) = m.compute\_output([A(i,j); B(i,j)]);

end

end

set(outputContour, 'ZData', E);

drawnow;

% keep frame for video

if record

mov(frame)=getframe(gcf);

writeVideo(writerObj,mov(frame));

frame=frame+1;

end

end

% store video

if record

close(writerObj);

end