

MULTILAYER PERCEPTRON

*Neural network to validate the MNIST dataset*

Frankfurt University of Applied Sciences

Presented by,

NARAYAN NARVEKAR, Mat. Nr: 1230345

*Under the guidance of Prof. Damir Dobric*

CONTENTS

* Introduction
* What is a neural network?
* Perceptron
* Multilayer perceptron
* Architecture
* Stages
* Project Details
* Link to Git repository
* Class MLPerceptronAlgorithm
* Class BackPropagationNetwork
* Testing the neural network
* Training process flowchart
* Validation process flowchart
* MNIST Training and test dataset
* Files Used
* Results analysis
* Conclusion
* References

INTRODUCTION

**What is a neural network?**

Neural networks are one of the most beautiful programming paradigms ever invented. In the conventional approach to programming, the computer is told what to do, breaking big problems up into smaller ones, with the tasks being precisely defined that the computer can easily perform. In a neural network we don't tell the computer how to solve our problem. Instead, it learns from observational data, figuring out its own solution to the problems presented to it.

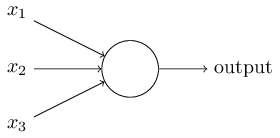
Learning from data automatically sounds promising. However, until 2006 there was no defined strategy to train neural networks to surpass more traditional approaches, except for a few specialized problems. What changed in 2006 was the discovery of techniques for learning in so-called deep neural networks. These techniques are now known as deep learning. They've been developed further, and today deep neural networks and deep learning have managed to achieve outstanding performance on many important problems in computer vision, natural language processing and speech recognition.

In the most general form, a neural network is a machine designed to model the way in which our brain can perform a particular task of interest. The heart of a neural network is the learning algorithm, whose function is to modify the weights of the neural network in an optimized manner to achieve the desired objective. How this is achieved would be explained in greater detail in the following chapters.

PERCEPTRON

**What is a Perceptron?**

A Perceptron, sometimes also referred to as an artificial neuron, takes several inputs and produces a binary output:



In the above example, the perceptron has three inputs, x1, x2 and x3. In general it could have more or fewer inputs. Key parameters of a perceptron are the ‘***weights’***.  Weights, usually denoted as w1, w2, w3 and so on, are real numbers which express the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum ∑jwjxj  is less than or greater than some **‘threshold value’**. Similarly, the threshold is a real number which is a parameter of the neuron. This can be represented in more precise algebraic terms as follows:

**Output = { 0 if ∑jwjxj <= threshold**

**{ 1 if ∑jwjxj > threshold**

That's all there is to how a simple perceptron works.

The condition ∑jwjxj > threshold could become cumbersome, and following notational changes can be made to simplify it. The first change is to write ∑jwjxj as a dot product, w.x = ∑jwjxj, where ‘w’ and ‘x’ are vectors whose components are the weights and inputs, respectively. The other change is to move the threshold to the LHS of the inequality, and replace it by what's known as the perceptron's ‘***bias’***, b = −threshold. Using the bias instead of the threshold, the perceptron rule can be rewritten:  
 **Output = { 0 if w.x + b <= 0**

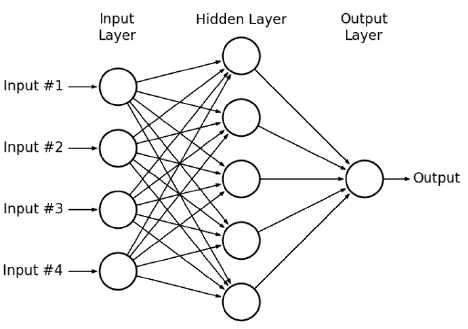
**{ 1 if w.x + b > 0**

The bias can be thought of as a measure of how easy it is to get the perceptron to output a 1. In other words, the bias is a measure of how easy it is to get the perceptron to fire. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1.

MULTILAYER PERCEPTRON

**Architecture:**

A multilayer perceptron (MLP) is a class of feed forward artificial neural network.  A MLP consists of at least three layers of nodes. Except for the input nodes, each node is a neuron that uses a nonlinear [activation function](https://en.wikipedia.org/wiki/Activation_function). An example of a simple MLP neural network is shown below:



The leftmost layer in this network is called the input layer, and the neurons within the layer are called input neurons*.* The rightmost or output layer contains the output neurons, or, as in this case, a single output neuron. The middle layer is called a hidden layer, since the neurons in this layer are neither inputs nor outputs. The term "hidden" perhaps sounds a little mysterious, but it really means nothing more than "not an input or an output". The above network has only one hidden layer, but there could be networks with multiple hidden layers.

Every neuron in a particular layer is connected via a specific weight to every neuron in the following layer. In the above network, each one of the four neurons in the input layer would be connected to each of the five neurons in the hidden layers by a unique weight assigned to each connection. Likewise, each of the five neurons in the hidden layer would be connected to the single neuron in the output layer by unique weights. In addition to the weights, every neuron in the hidden and output layers would be assigned a bias, which along with all the weights, contributes to something known as the weighted sum that is calculated at each node at these layers.

**Stages:**

Following are the important stages that we encounter in a MLP neural network:

1. **Input a set of training examples:**

Training the MLP neural network is the most important step and in this context, we would need precisely defined ***training examples***, also known as ***training datasets***. These training datasets consist of a set of inputs and also the actual outputs that correspond to these inputs. These datasets are then fed to the MLP neural network, which is thus trained by updating the weights and biases at each neuron such that the error at the output is minimized.

1. **Feedforward path:**

For each training example, set the corresponding input activation ax,1. In general, we can represent the activations at each neuron in the network as follows:

ax,l, where ‘x’ refers to the (x+1)th neuron(since we start numbering the neurons from zero and not one) and ‘l’ refers to the number of the layer that we are referring to.

For eg., a1,1 is the input activation at the 2nd neuron of the 1st layer(input layer) and a3,2 is the input activation at the 4th neuron of the 2nd layer(1st hidden layer).

If L is the total number of layers in the network, then for each l = 2, 3..., L, the weighted sum is computed at each neuron as follows:

**zx,l = wlax,l + bl**

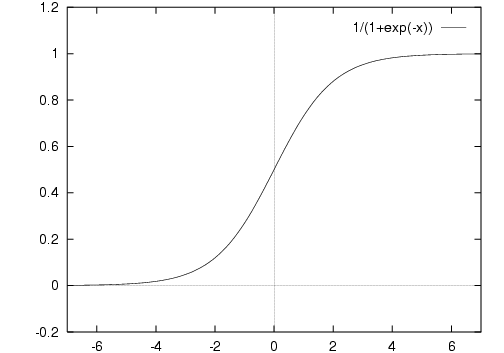
where, wl is the weight matrix which represents all the weights connecting the neurons in the lth layer to the neurons in the (l-1)th layer, ax,l is the vector of activations input to all the neurons in the lth layer and bl is the vector of biases applied to all the neurons in the lth layer.

This weighted sum is then used to calculate the output at each neuron by passing it through the ***activation function, σ***.

The two activation functions that are supported by the project are both sigmoid functions and are described by:

**σ(x) = tanh(x) and σ(x) = (1 + e-x)-1**

The first is a [hyperbolic tangent](https://en.wikipedia.org/wiki/Hyperbolic_tangent) that ranges from -1 to 1, while the other is the [logistic function](https://en.wikipedia.org/wiki/Logistic_function), which is similar in shape but ranges from 0 to 1. Consider the second function. Here is the shape of the function:

******

One big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1. They can output any real number between 0 and 1, so values such as 0.23, 0.45 are legitimate outputs. This can be useful if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network. But sometimes it can be a nuisance. Suppose we want the output from the network to indicate either "the input image is a 5" or "the input image is not a 5". Obviously, it'd be easiest to do this if the output was a 0 or a 1, as in a perceptron. But in practice we can set up a convention to deal with this, for example, by deciding to interpret any output of at least 0.5 as indicating a "5", and any output less than 0.5 as indicating "not a 5".

The outputs thus calculated serve as input activations to the neurons in the following layer. At the end of the feedforward path stage, we have thus calculated outputs at each of the neurons in the network, terminating with the output layer neurons.

1. **Output layer error:**

At the end of the feedforward path when the output is calculated at the output neuron, this value represents the “calculated output” of the network. The goal of the MLP learning algorithm is to calculate the error between the calculated output and the actual output of the different training datasets and minimize this error by adjusting the weights and biases in the network. The error at the output neuron is called the cost function. In this project we have used the quadratic cost and this is given by:

**C = (1/2n) \* ∑ | y(x) – aL(x) |2**

where n is the total number of training examples, y(x) is the actual output which is a part of the training dataset and aL(x) is the calculated output of the MLP network.

The output error for each neuron in the output layer is given by the equation:

**δx,L = ∇aCx ⊙ σ|(zx,L)**

where ∇aCxis vector which represents the gradient of the quadratic cost at each output neuron with respect to its corresponding input activation, σ|(zx,L)is the vector which represents the derivative of the sigmoid function applied to the weighted inputs calculated at the output layer. The symbol “⊙” represents the Hadamard product.

The Hadamard product between the above two vectors gives the error at the output layer, δx,L.

1. **Backpropagation:**

The error that is calculated at the output layer is then distributed backwards towards all the neurons in the network. In other words, the error is propagated backwards from the output layer to all the hidden layer neurons, starting from the last hidden layer and ending at the first hidden layer. Hence the name backpropagation.

For each l = L-1, L-2,…, 2, we need to compute the error at each neuron. This error is given by,

**δx,l = ((wl+1)T δx,l+1 ) ⊙ σ|(zx,l)**

where (wl+1)T is the matrix of weights at the (l+1)th layer, δx,l+1 is the vector of errors at the (l+1)th layer, σ|(zx,l)is the vector which represents the derivative of the sigmoid function applied to the weighted inputs calculated at the lth layer.

Once the errors have been determined and back propagated to each neuron in the hidden layers, the next step is to calculate the rate of change of the cost function with respect to all the weights and biases in the network.

The rate of change of cost function with respect to the biases is given by,

**∂C/∂blj = δlj**

where blj represents the bias of the (j+1)th neuron of the lth layer and δlj is the error calculated at the (j+1)th neuron of the lth layer.

This means that the error δlj is exactly equal to the rate of change ∂C/∂blj.

Similarly, the rate of change of cost function with respect to the weights is given by,

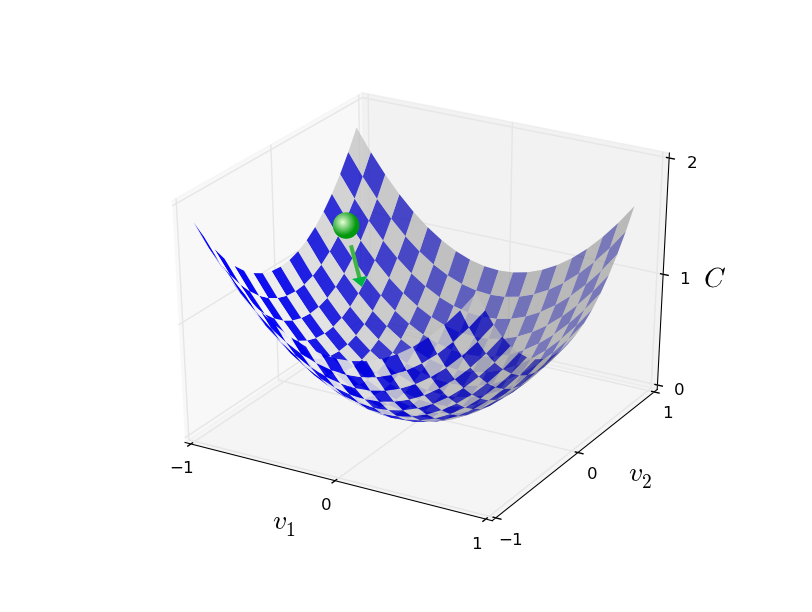
**∂C/∂wljk = akl-1 δlj**

where wljk represents the weight connecting the (j+1)th neuron in the lth layer to the (k+1)th neuron in the (l-1)th layer, akl-1 is the calculated output at the (k+1)th neuron of the (l-1)th layer and δlj is the error calculated at the (j+1)th neuron of the lth layer.

1. **Gradient descent:**

The main purpose of the MLP learning algorithm is to find a set of weights and biases that makes the cost as small as possible. This can be achieved by a process known as gradient descent. The working principle of gradient descent is explained below:

The quadratic cost is a function of weights and biases. Consider the below figure which shows an example of the quadratic cost, C, taking on a set of values for different values of weights, v1 and biases, v2.



We will succeed when the cost function has reached its minimum value. If suppose the cost function has a current value which is represented by the ball in the figure, then the slope of the tangent at that particular point will give us a direction to move towards. We then make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the learning rate α.

After each step is taken, we have a new set of weights and biases. Continuing this procedure of taking steps in the direction of the steepest descent, finally we end up at a point where the cost function is minimum. The weights and biases which are thus calculated when the cost function is minimum are said to be the optimal weights and biases of the network. Based on this principle, we can come up with the following equations:

For l = L, L-1, L-2,…, 2, the biases are updated according to the following rule,

**blj -> blj – (α δlj)**

Similarly, the weights are updated according to the following rule,

**wljk -> wljk – (α akl-1 δlj)**

These updates are carried out until all the training examples have been exhausted and all the defined number of iterations have been executed.

PROJECT DETAILS

**Link to Git Repository**

[*https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi/tree/mlp-backpropagation*](https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi/tree/mlp-backpropagation)

Algorithm: [*https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi/tree/mlp-backpropagation/LearningApi/src/MLAlgorithms/NeuralNetworks/NeuralNet.MLPerceptron*](https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi/tree/mlp-backpropagation/LearningApi/src/MLAlgorithms/NeuralNetworks/NeuralNet.MLPerceptron)

Test Project:

[*https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi/tree/mlp-backpropagation/LearningApi/test/MLPerceptron*](https://github.com/UniversityOfAppliedSciencesFrankfurt/LearningApi/tree/mlp-backpropagation/LearningApi/test/MLPerceptron)

Test Runner Project:

To be updated

CSV files:

*To be updated*

**Note:** MNISTFiles.rar contains the training and test dataset CSV files. MNISTFiles.rar needs to be unzipped before running the application.

**Class MLPerceptronAlgorithm**

This class contains the MLP learning algorithm. A brief description of the important methods contained in this class is given below:

1. ***IScore Run(double[][] featureValues, IContext ctx):***

This method accepts the training examples as input and performs the training of the MLP neural network.

The weights and biases are initialized to a random value between 0 and 1.

For each batch of training examples, the feedforward path is implemented by calculating the outputs at each neuron.

For each batch of training examples, a BackPropagationNetwork object is instantiated, and the methods to calculate the errors at each node, and cost function changes with weights and biases are executed. The new weights and biases are updated in the methods defined in BackPropagationNetwork class.

1. ***double[] Predict(double[][] data, IContext ctx)***

This method accepts the test data as input and determines the output for each sample of test data.

The return value of the function is the calculated output. The return value is then compared in the test project method with the correct outputs, and thus used to determine whether the unit test has passed or failed.

**Class BackPropagationNetwork**

This class contains methods that implement the back propagation part of the MLP learning algorithm. A brief description of the important methods contained in this class is given below:

1. ***void CalcOutputError (double[] output, int[] hiddenlayerneurons, double[] outputSum, double[] inputVector, IContext ctx)***

This method calculates the error at the output layer as defined by the equation,

**δx,L = ∇aCx ⊙ σ|(zx,L)**

1. ***void CalcOutputErrorSoftMax(double[] output, int[] hiddenlayerneurons, double[] inputVector, IContext ctx)***

This method calculates the error at the output layer if softmax activation function is used at the output layer, as defined by the equation,

**δx,L = ∇aCx ⊙ σ|(zx,L)**

1. ***void CalcHiddenLayersError (double[][] hidLayersOutputs, double[][,] weights, int[] hiddenlayerneurons, double[][] hidLayersSums, double[] featureValues)***

This method calculates the error that is back propagated at each neuron present in the hidden layers as defined by the equation,

**δx,l = ((wl+1)T δx,l+1 )⊙ σ|(zx,l)**

1. ***void CostFunctionChangeWithWeights(double[][,]currWeights, double[][]hidLayersOutputs, int[] hiddenlayerneurons, double learningrate, double[]inputVector)***

This method calculates the rate of change of cost function with respect to all the weights that are a part of the network as defined by the equation,

**∂C/∂wljk = akl-1 δlj**

1. ***void UpdateWeights(double[][,] currWeights, double[][] hidLayersOutputs, int[] hiddenlayerneurons, double learningrate, int numOfInputNeurons, out double[][,] newWeights)***

This method calculates the newly update weights as defined by the equation,

**wljk -> wljk – (α akl-1 δlj)**

1. ***void CostFunctionChangeWithBiases(double[][] currentbiases, int[] hiddenlayerneurons, double learningrate)***

This method calculates the rate of change of cost function with respect to all the biases that are a part of the network as defined by the equation,

**∂C/∂blj = δlj**

1. ***void UpdateBiases(double[][] currentbiases, int[] hiddenlayerneurons, double learningrate, out double[][] newbiases)***

This method calculates the newly update biases as defined by the equation,

**blj -> blj – (α δlj)**

TESTING THE NEURAL NETWORK

TestRunner:

* The TestRunner is a tool that allows the execution of multiple test cases simultaneously by assigning it to multiple threads.
* After building the TestRunner tool, it can be run using the following command:

**dotnet run <name of test file> <number of concurrent tests>**

Example: dotnet run MNIST\_Test\_01.txt 4

* The maximum value of “number of concurrent tests” depends on the number of cores of the CPU executing the TestRunner tool.
* The “name of test file” refers to the file which contains the details of the hyper parameters. The files can be found at the following location:

To be updated

* The hyper parameters files contain the values for the number of iterations, learning rate, batch size, number of hidden layers and neurons in each hidden layer, which are used to initialize the neural network.
* Before running the TestRunner tool, it is important to unzip the training and test dataset files from MNISTTestFiles.rar and save in at the following location:

To be updated

* During the execution of the test cases, if the trained neural network for any of the architectures(specified in the hyper parameters file) has a final prediction accuracy greater than a particular threshold value, for e.g., accuracy > 95%, then that model is saved by storing all the parameters(which include the hyper parameters and the weights and biases) in a json file at the following location:

To be updated

* The saved model can be loaded at a later point of time and can be directly used for evaluating the performance of the network of some new test dataset, without having to spend the additional time in training the neural network.
* After the execution of all test cases, the TestRunner tool gives an option of loading an already saved model and testing its performance on some test dataset.

Class MNISTFileRead:

* This class is part of the Test Project and contains methods ReadMNISTTrainingData and ReadMNISTTestData to read the content of the training and test dataset csv files and store the normalized data into jagged arrays which are then fed as input to the neural network training and prediction algorithms respectively.
* This class also contains the method UnitTestMNISTTestRunner which is invoked from the TestRunner module, which in turn contains function calls to the MLPerceptronAlgorithm Class methods, to firstly train the neural network by feeding it with the training dataset, and then evaluating its performance on the test dataset.

TRAINING PROCESS FLOWCHART



Using gradient descent update the new weights and biases based on results from back propagation

Execute the Back Propagation path on the same training example

Execute the feedforward path on a training example

Initialize the weights and biases with random values between 0 and 1

Call the Run method by passing the training examples as input parameters

No Yes No Yes

Repeat the training procedure on all training examples

Stop training the network

All iterations completed?

Loop to next training example

All training examples exhausted?

VALIDATION PROCESS FLOWCHART

Invoke the predict method from the unit test case. Pass the test data as parameter

All test data exhausted?

Loop to next test data

Compare the calculated result with the correct output

Calculate the output for a test data sample and update it in the result array

No

Yes

Return the result array

Calculated result == correct output?

Yes No

Unit test failed

Unit test passed

MNIST TRAINING AND TEST DATASET

Training dataset:

* The training dataset consists of a csv file which contains the pixel values of 60000 examples of handwritten digits.
* Every handwritten digit has a resolution of 28\*28 pixels. Thus the number of features in each of the training examples is given by,

28 \* 28 = 784 features, where each feature of a particular training example represents a corresponding pixel value at a certain location in the image.

* Every training example has associated with it a label. The value of this label is equal to the actual digit represented by the training example. E.g., if the training example is generated from an image of a handwritten digit 5, then label has a value equal to 5.
* 80% of the training dataset examples are used for training the neural network in each epoch.
* 20% of the remaining dataset examples are used for validating the neural network in each epoch, after the training is completed for that particular epoch.

Test dataset:

* The test dataset consists of another csv file which contains the pixel values of 10000 examples of handwritten digits.
* The test dataset is generated in a similar way to the training dataset.
* The test dataset is used to evaluate the performance of the neural network, after the network has been trained.

FILES USED

*mnist\_train.csv* and *mnist\_test.csv* files:

* These files represent the actual training and test datasets used by the neural network.
* After cloning the repository, these files can be accessed by unzipping the file MNISTTestFiles.rar.
* MNISTTestFiles.rar can be accessed from the following location:

To be updated

* After unzipping the file, the *mnist\_train.csv* and *mnist\_test.csv* files should be copied to the following location before running the TestRunner tool:

To be updated

*mnist\_performance\_params\_X.csv* files:

* The *mnist\_performance\_params\_X.csv* files, where ‘X’ represents the test case number, measures the performance of the neural network for a defined set of hyper parameters.
* It contains the following fields:

*Epoch Number*

*Training set Loss after every epoch*

*Training set Accuracy after every epoch*

*Validation set Loss after ever epoch*

*Validation set Accuracy after ever epoch*

*Time elapsed after every epoch*

* Ideally, the training set and validation set losses should reduce after every epoch of training. The training set and validation set accuracy should increase after every epoch of training and should ideally reach a value of 1.
* These files can be accessed from the following location:

To be updated

*mnist\_result\_X.csv* files:

* The *mnist\_result\_X.csv* files, where ‘X’ represents the test case number, contains the predicted result of each of the 10000 examples present in the test dataset.
* The last row of this file contains the model accuracy, which is calculated as follows:

*Model Accuracy = Number of correctly predicted digits/Number of test examples*

* These files can be accessed from the following location:

To be updated

*mnist\_savedmodel\_X.json* files:

* The *mnist\_savedmodel\_X.json* files, where ‘X’ represents the test case number, contain the saved model with all the relevant parameters for the neural network.
* The saved model can be loaded at a later point of time and can be directly used for evaluating the performance of the network of some new test dataset, without having to spend the additional time in training the neural network.
* These files can be accessed from the following location:

To be updated

*mnist\_savedmodel\_X\_result.csv* file:

* The *mnist\_savedmodel\_X\_result.csv* file, where ‘X’ represents the test case number, contains the predicted result of each of the 10000 examples present in the test dataset.
* This file is generated by loading the saved model and directly running the prediction algorithm on this saved model, without the need of spending time on training the neural network.
* These files can be accessed from the following location:

To be updated

*MNIST\_Test\_0X.txt* files:

* The *MNIST\_Test\_0X.txt* files, where ‘X’ ranges from 1 to 12, are the files which are passed as an argument while running the TestRunner tool.
* These files contains the details of the hyper parameters.
* The hyper parameters files contain the values for the number of iterations, learning rate, batch size, number of hidden layers and neurons in each hidden layer, which are used to initialize the neural network.
* These files can be accessed from the following location:

To be updated

*MNIST\_Test\_0X* folders:

* The folders *MNIST\_Test\_0X,* where ‘X’ ranges from 1 to 12, contain the graphical results for every test case that has been executed using the TestRunner tool.
* These graphical results are generated using a python script, which accepts the *mnist\_performance\_params\_X.csv* as input.
* To understand how these graphical results are organized, consider an example of the folder MNIST\_Test\_02.
* MNIST\_Test\_02 contains files mnist\_results\_x-y, where x represents every test case(one set of hyper parameters from file MNIST\_Test\_02.txt) and y can have the value 0 or 1.
* mnist\_results\_0-1 represents the plot of training and validation accuracy against the number of epochs.
* mnist\_results\_0-2 represents the plot of time elapsed in minutes against the number of epochs.

RESULTS ANALYSIS

CONCLUSION

The multilayer perception performs a good job at solving linearly non separable problems. Some of the key factors which affect the performance of an MLP network are listed down below:

**Activation function:**

The Sigmoid and HyperbolicTan functions are the activation functions that are being supported by the MLP network.

The main reason for choosing Sigmoid is because it exists between 0 and 1 and hence it is especially used for models where we have to predict the probability of the output. However, if the input to the Sigmoid function is too large, then it gives rise to the problem of “vanishing gradients”. The network refuses to learn or is drastically slow.

In case of the HyperbolicTan function, the characteristics are similar to sigmoid. The gradient however is stronger for HyperbolicTan than sigmoid (derivatives are steeper). Deciding between sigmoid or tanh will depend on the requirement of gradient strength. Like Sigmoid, HyperbolicTan function also has the vanishing gradient problem.

To solve the problem of vanishing gradients, the Rectified Linear Unit (ReLU) activation function is used. ReLUs are much simple computationally. ReLUs only saturate when the input is less than 0. And even this saturation can be eliminated by using leaky ReLUs. For very deep networks, saturation hampers learning, and so ReLUs provide a nice workaround.

**Gradient Descent vs. Stochastic Gradient Descent:**

In both gradient descent and stochastic gradient descent, we update the weights and biases in an iterative manner to minimize an error function.

While in gradient descent, we run through all the samples in our training dataset to do a single update for a parameter in a particular iteration, in stochastic gradient descent we use only one or a subset of training samples from the training dataset to perform the update of the parameter in a particular iteration.

Stochastic gradient descent often converges much faster compared to gradient descent but the error function is not as well minimized as in the case of gradient descent. In most cases, the close approximation that we get in stochastic gradient descent for the parameter values are enough because they reach the optimal values and keep oscillating there.

Another advantage of stochastic gradient descent is that the randomness or noise introduced by it allows escaping from local minima to reach a better minimum.

**Learning rate:**

Choosing of learning rate helps us in determining how fast we can reach the minima. If we choose larger value, then we might overshoot the minima and smaller values might take long time for convergence.

In general, we need to find a learning rate that is low enough for the network to converge to something useful, but high enough such that we don’t have to spend years training it.

**Epoch:**

One Epoch (iteration) is when the entire training dataset is passed forward and backward through the neural network only once.

It doesn’t make sense to pass the entire dataset through a neural network only once. We need to pass the full dataset multiple times to the same neural network. Gradient descent is an iterative process and updating the weights and biases with one epoch is not enough. There is no hard and fast rule on the right number of epochs for a network. The answer is different for different datasets and is generally dependant on how diverse the data is.

**Number of hidden layers:**

One of the common rules of thumb is that *“the optimal size of the hidden layer is usually between the size of the input and size of the output layers”.* Also, the situations in which the performance improves with a second (or third, etc.) hidden layer are very few. One hidden layer is sufficient for majority of the problems.

In general, we can get decent performance by setting the hidden layer configuration using the below two rules:

* Number of hidden layers equals one
* Number of neurons in that layer is the mean of the neurons in the input and output layers.

REFERENCES

1. <https://en.wikipedia.org/wiki/Multilayer_perceptron>
2. Neural Networks and Deep Learning – Michael Nielsen
3. Neural Networks, A Comprehensive Foundation – Simon Haykins
4. Investigation and improvement of multi-layer perceptron neural networks for credit scoring – Zongyuan Zhao, Shuxiang Xu, Byeong Ho Kang, Mir Md Jahangir Kabir, Yunling Liu, Rainer Wasinger
5. [Neural network approach to Iris dataset](https://www.kaggle.com/louisong97/neural-network-approach-to-iris-dataset) – Louis Ong
6. NET & C# In-Browser tutorial – Microsoft