

Parameter Tuning with Multilayer Perceptron and Radial Basis Function Neural Networks

Brandon Sladek

BRANDONSLADEK@GMAIL.COM

School of Computing

Montana State University

Bozeman, MT 59718, USA

Jared Thompson

J.A.THOMPSON22@GMAIL.COM

School of Computing

Montana State University

Bozeman, MT 59718, USA

Kyle Hagerman

HAGERMANKYLE96@GMAIL.COM

School of Computing

Montana State University

Bozeman, MT 59718, USA

Ryan Hansen

RYANHANSEN2222@GMAIL.COM

School of Computing

Montana State University

Bozeman, MT 59718, USA

Abstract

We implemented two versions of a feedforward neural network, the Multilayer Perceptron Network, and the Radial Basis Functions Network. After implementing the networks in Python, we tested both networks using various parameters and network configurations (number and size of layers). In general, we found that simpler network configurations led to the best results. In other words, the MLP networks achieved the best results with 0 hidden layers, and the RBF networks achieved the best results when the middle layer was configured with the K-Means cluster centroids. However, we did manage to improve performance on the Car data set using the MLP network with additional hidden layers. Both networks tended to have very long running times for training, which made it difficult to properly compare the effect of changing parameter values and network configurations.

Keywords: Neural Networks, Multilayer Perceptron, Radial Basis Functions

1. Introduction

Artificial neural networks have quickly become one of the most popular machine learning strategies in industry today. Their ability to learn complex decision boundaries for a wide variety of problems has led to the successful application of neural networks in fields as diverse as medical imaging and financial forecasting. Additionally, the substantial increase in available computing power over the last decade has further lead to a resurgence of interest in neural networks within the research community. This research explores the strengths

and weaknesses of two of the most popular neural networks - the Multi-Layered Perceptron (MLP) network and the Radial Basis Function (RBF) network.

2. Problem

In the greatly influential research paper “Multilayer Feedforward Networks are Universal Approximators” published in 1989, the authors rigorously established that standard multilayer feedforward neural networks, with arbitrary activation (“squashing”) functions, can approximate essentially any function to any degree of accuracy, provided the network is large enough (Hornik et al. (1989)). Furthermore, the authors claim that “failures” in the networks can be attributed to poor training results, poor choices for hyperparameter values, or simply a lack of a deterministic relationship between the input and the expected output. In other words, theoretically, a neural network can be built and trained to any degree of accuracy, provided there is an actual relationship between the input and expected output, and the network is built in such a way that it can adequately learn the features of the input space and map them to an accurate output space with respect to the expected output.

We seek to verify this claim by running two different neural network implementations on various data sets using various parameter values. With repeated training/testing of the networks using different parameter values, we will be able to gain insight into which parameter values work best for each data set and network configuration. Additionally, we aim to compare the two networks relative to each other. Identifying their strengths and weaknesses will help the community gain insights for when to use each network.

3. Networks

In this section we outline the fundamental mathematical concept behind feed forward neural networks. Additionally, we note our specific design choices for implementation (e.g. activation functions).

3.1 Feed Forward Neural Networks

Overview The core idea behind a neural network is to generate any arbitrarily complex function by simulating biological learning. Given some input, a series of neurons activate, eventually leading to some series of output neurons that represent a meaningful output in the context of the problem. Neurons are related to each-other via weights, and outputs are normalized via activation functions. Feed Forward Networks have the additional structure of layers, and a neuron in some layer only has weights pointing forward to the next layer.

Formalism The following explanation is visualized in figure 1. We can represent a Feed Forward Neural Network, N , as a collection of layers, $N = \{L_i\}$. Each layer L_i , is composed of a collection of n_i nodes $L_i = \{p_1, p_2, \dots, p_{n_i}\}$. Each node $p_j \in L_i$ for $p'_k \in L_{i-1}$ has an *activation value*, a , given by

$$a(p_j) = f(\beta_j + \sum_{k=0}^{n_{i-1}} w_k a(p'_k)) \quad (1)$$

In words, this says the activation value of each node is given by the activation value of the previous layer, scaled by some weight, shifted by some number, and finally normalized

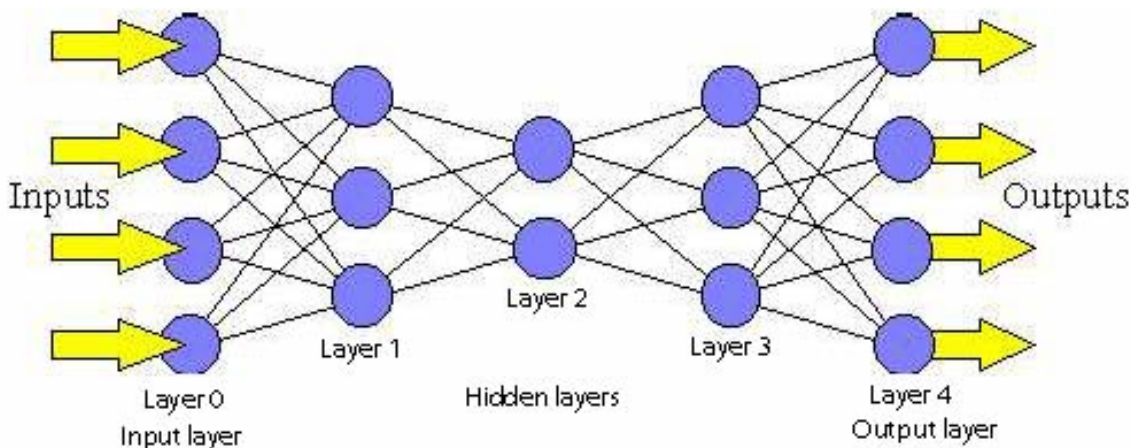


Figure 1: Pictorial representation of a feed-forward neural network. Image from Caroline Clabaugh and Pang (2000)

through some normalization function. Notice, then we can generalize the function to a series of matrix multiplications.

$$\mathbb{A}_i = f(\beta_i + \mathbb{W}_{i-1,i} \cdot \mathbb{A}_{i-1}) \quad (2)$$

Where \mathbb{A}_i is the activation vector of layer i , β_i is the shifting vector, and $\mathbb{W}_{i-1,i}$ is the weight matrix mapping from layer $i - 1$ to layer i . We call f the *activation function* and β is the bias term. For classification data, each output node represents a class. Regression includes only one output node - the class value of the regression data point.

Training The network is initialized with arbitrary weights and chosen activation functions. The network is trained through supervised learning. Given a training data point x , with desired output value Y , there will be a subsequent output activation in the output nodes $N(x)$. To train the network, we take the squared error, $Err = (Y - N(x))^2$, and attempt to minimize it through a gradient learning algorithm - backpropagation. To do this, we optimize ∇Err , where we are taking derivatives with respect to weights. The goal is to learn the weights and biases that minimize the squared error. Subsequently, we update each weight by the following formula.

$$\mathbb{W}_{new} = \mathbb{W}_{old} - \alpha \nabla Err \quad (3)$$

Where α is a tuned learning rate. We propagate through each layer, until each weight in our network has been updated. Finally, it is proven that given enough nodes and layers, any function can be approximated to infinite precision using one of these neural nets (J.R.M.Smiths) (Hornik et al. (1989)).

3.2 Multilayer Perceptron Network

This is a special case of a feed forward neural network. In this network, we get to choose an arbitrary number of layers, the number of nodes in each layer, and the activation function.

The entire network is trained with gradient descent using backpropagation.

Overview The MLP is perhaps one of the most general implementations of a feed forward network. The idea is to stack an arbitrary number of layers of nodes to try to capture patterns in data.

Formalism The main theory behind the MLP is encompassed in its activation functions. Common activation functions intend to normalize the activation of each node between 0 and 1, examples being the ReLU and Sigmoid functions. This means a low activation should have negligible impacts on the output, and high activation should have larger effects, as desired (M.W Gardner (1998)).

Our Implementation We implemented the option to use both ReLU or the Sigmoid, but ran experiments with Sigmoid. Then, we varied the hyperparameters - number of nodes in each layer and number of layers - in order to find good results with respect to our loss functions.

3.3 Radial Basis Function Network

This is another special case of a feed forward neural network. In this network, there are exactly three layers - an input, an output, and a hidden layer. The number of nodes in the hidden layer is in general defined by some set of statistically significant data points, each defining a center for a radial basis function. Finally, the first weight matrix is left untrained.

Overview The RBF is a network designed to gauge similarity of an input point to various significant points. The idea is if an input point is close in distance to combinations of these significant "landmark" points, we hope to be able to generate some kind of conclusion on the class value of the point. The following image provides a pictorial representation of the RBF network.

Formalism The first important part of an RBF is determining the number of hidden layer nodes. These are often chosen through statistical aggregates of an initial data set (e.g. k-means clustering points). We will call these the set of landmark points, L . Each point in L is assigned a node, where it represents the center of the Radial Basis Function activation function for that node. A radial basis function is defined as a function f_r centered at r such that two points yield the same value so long as their distance from the center is the same:

$$f_r(x) = f_r(y) \iff d(x, r) = d(y, r) \quad (4)$$

Additionally, the function is normalized to yield a result between 0 and 1. Since the initial weight matrix is defined to simply apply the identity on the input to each node, this, in essence, gauges the similarity of the input to each specific landmark point. One common radial basis function is the gaussian:

$$f_r(x) = ce^{-b(r-x)^2} \quad (5)$$

for constants b and c . When normalized, $c = 1$. The last important change in the RBF is in training. Instead of using backpropagation all the way through the network, it stops

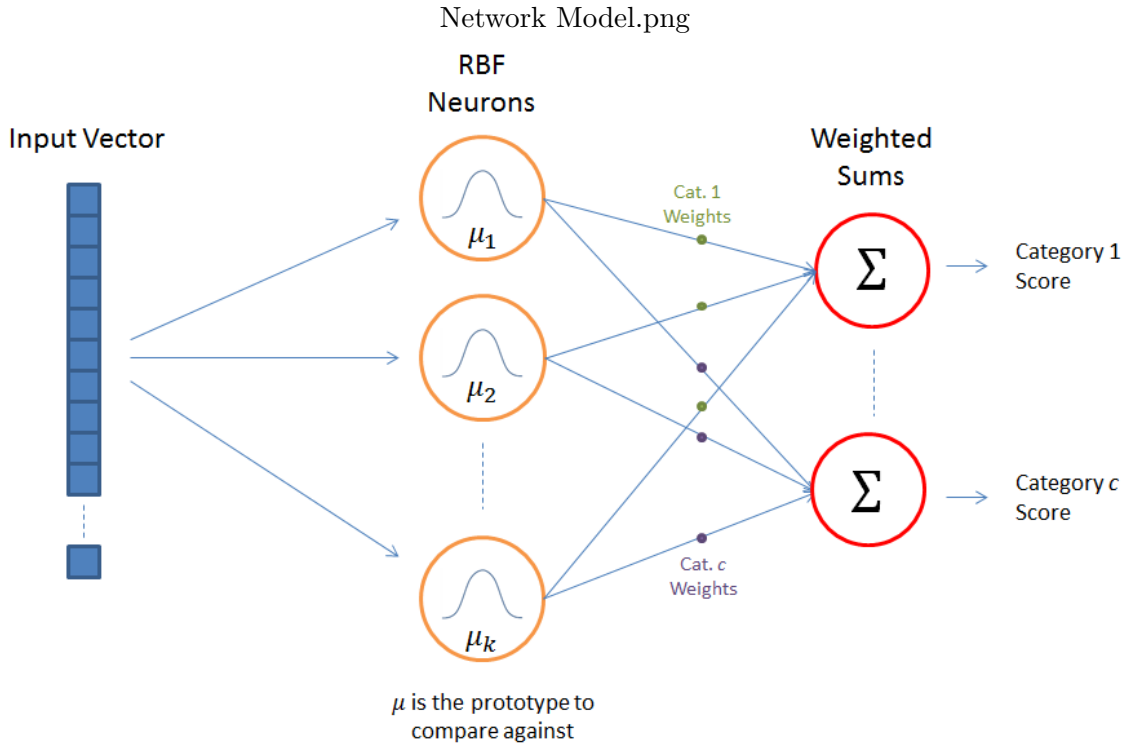


Figure 2: A general RBF network, activation functions centered at μ_i . Notice the weights from the input layer to the hidden layer are not specified - this is because they are untrained identity functions on the input. Image from McCormick (2013)

after one single propagation. This is because we want the first weight matrix to always yield the identity for each node, otherwise it will stop measuring similarity between input and landmark points. Note, because of these qualities, RBF is widely used in regression or function approximation (McCormick (2013), Mohit (2017)).

Our Implementation We implemented three algorithms to yield our set of landmark points - k-means, k-medoids, and edited-nearest-neighbors. Our radial basis function of choice was the normalized gaussian, as described above.

3.4 Momentum

Incorporating momentum into the training phase typically results in improved performance by decreasing the time required for convergence, and increasing the range of parameter values with which network learning will still converge.

Theory Recall Equation (3) from section 3.1, discussing the update strategy for learning weights.

$$\mathbb{W}_{new} = \mathbb{W}_{old} - \alpha \nabla Err \quad (6)$$

Momentum is a strategy to help push gradient optimization techniques out of shallow local optima and reduce convergence time. Mathematically, it manifests itself as a simple term tacked on at the end of the former update strategy.

$$\mathbb{W}_{new} = \mathbb{W}_{old} - \alpha \nabla Err_{current} - \lambda Err_{previous} \quad (7)$$

Where λ is some tuned scaling parameter. Momentum helps solve two big issues with gradient descent algorithms. First, it helps to get out of local minima. Consider figure 2. Under normal circumstances, landing in the local minimum means converging to the local minimum. However, incorporating a momentum means sometime the previous gradient update is enough to push it out. Second, imagine the left hand side of figure 2 going off to infinity. If the initial weight system started points far out where nothing is interesting, momentum helps reach the interesting spots faster, increasing the update magnitude each time (Qian (1999)).

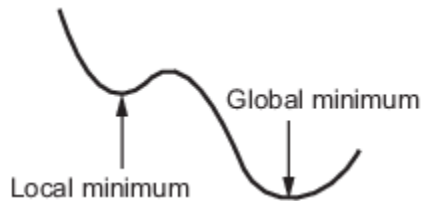


Figure 3: Advantage of Momentum - figure from Mathworks (2019)

Our Implementation We implemented momentum as an option when training either of the neural networks. When momentum is used, the “*momentum_beta*” hyperparam configures the value for λ in Equation (7) above.

3.5 Mini-batching

Building neural networks on large data sets typically means long runtimes. Often times, to help mitigate this issue, a technique called minibatching is used.

Theory Each input x yields a weight update $\Delta\mathbb{W}$. Standard implementations of back-propagation update each weight after every x . This is computationally expensive. To help speed things up, instead of updating after each x , a common practice is to choose some batch size n and obtain n potential update weights, $\Delta\mathbb{W}$. These are then averaged, and the single average is propagated throughout the network (Masters and Luschi (2018)).

Our Implementation We use mini-batching with a configurable batch size.

4. Experiment

In this section we explain our general approach for accepting or rejecting our hypothesis for the various data sets, network configurations, and corresponding parameter values.

4.1 Approach

Our goal was to compare the effectiveness of both networks - the MLP and the RBF. Across 6 constant data sets, we tested the MLP networks with 0, 1, and 2 hidden layers, with variable numbers of nodes in each layer, and the RBF networks with the middle layer consisting of the Project 2 results from ENN, K-Means, and K-Medoids.

Using Python and the Pandas library, we opted to run 10-fold class validation on each of the data sets. Cross validation helped to ensure our results were not particular to a specific training/test set. In probabilistic experiments, it is possible to get good or bad results due to random chance. Cross validation helps mitigate this problem by running the same process (in this case the algorithms) on various samples of the data (Kohavi (1995)). Finally, to ensure that each column attribute in our feature vector held equal weight, we normalized the column attributes to be values between zero and one.

Finally, we specify our loss functions for comparing the algorithms. We are interested in two components of the algorithms - some form of accuracy and runtime. *Accuracy*: Accuracy refers to how correct the network predicts outputs. For classification data, we use accuracy (0-1 loss) and for regression data we use mean squared error. *Convergence Time*: Convergence time refers to how long the program took to train and execute. We compare these times directly.

4.2 Hypothesis

Our hypothesis was a ranking based on the performance of each network on each of the six data sets. We outline our ranking and justification in the table below.

1	Algorithm	Accuracy Rank	Convergence Time Rank
2	MLP - 0 Hidden Layer	6 - 0 Hidden layers means no sophistication, just linear analysis	1 - No hidden layers
3	MLP - 1 Hidden Layer	5 - Should be similar to RBF, but MLP nodes are not intelligently chosen	4 - Likely more nodes
4	MLP - 2 Hidden Layer	1 - 2 Hidden layers should theoretically add lots of pattern capability	6 - 2 hidden layers
5	RBF - ENN	4 - ENN adds a ton of points, many of which won't be meaningful (like a mean)	5 - More nodes
6	RBF - KMeans	2 - Should represent a cluster well	3 - Slower than kmedoids
7	RBF - KMedoids	3 - Close to Kmeans, but worse since it is not the radial center of its cluster	2 - Faster than kmeans

Figure 4: Table depicting our performance hypothesis. Justification is included.

4.3 Results

Figure 5 below shows our results for the MPL network. Figure 6 below shows our results for the RBF network. A higher accuracy indicates better performance for the classification data sets, and a lower error indicates better performance for the regression data sets. Valid results were not obtained for every data set and network permutation.


Segmentation	Car	Abalone	Machine	Forest Fires	Wine
(50, 10, 5) [19, 7] accuracy: 83.3% time: 16m10s	(10, 100, 3) [6, 4] accuracy: 73.7% time: 18m44s	N/A	(20, 4, 5) [9, 1] error: 0.004 time: 2m5s	(10, 20, 3) [12, 1] error: 0.015 time: 4m25s	(10, 100, 3) [11, 7, 3, 1] error: 0.02 time: 113m38s
(10, 10, 5) [19, 19, 7] accuracy: 22.9% time: 8m38s	(10, 50, 3) [6, 10, 4] accuracy: 81.3% time: 30m13s	N/A	(20, 4, 5) [9, 3, 1] error: 0.0065 time: 3m35s	(10, 20, 3) [12, 6, 1] error: 0.0035 time: 5m49s	N/A
(10, 100, 3) [19, 10, 5, 7] accuracy: 18% time: 6m12s	(10, 10, 3)  [6, 4, 2, 4] accuracy: 84.9% time: 21m41s	N/A	(10, 20, 3) [9, 6, 3, 1] error: 0.084 time: 3m42s	(10, 20, 3) [12, 10, 6, 1] error: 0.015 time: 13m57s	N/A

Figure 5: MLP Results - tuple values are (max iterations, batch size, eta), and square brackets denote the layer sizes for each network. N/A means unavailable/inconclusive results.


	Segmentation	Car	Abalone	Machine	Forest Fires	Wine
ENN	(10, 50, 3) [19, x, 7] accuracy: 19% time: 8m	(10, 50, 3) [6, x, 4] accuracy: 17% time: 12m	(10, 50, 3) [8, x, 28] accuracy: 22% time: 15m	(10, 50, 3) [9, x, 1] accuracy: 11% time: 4m	(10, 50, 3) [12, x, 1] accuracy: 12% time: 6m	(10, 50, 3)  [11, x, 1] accuracy: 8% time: 18m
K-Means	(10, 50, 3) [19, x, 7] accuracy: 44.8% time: 1m15s	(10, 50, 3) [6, x, 4] accuracy: 73% time: 45m	(10, 50, 3) [8, x, 28] accuracy: 68% time: 57m	(10, 50, 3) [9, x, 1] accuracy: 7.4% time: 36m	(10, 50, 3) [12, x, 1] accuracy: 8% time: 42m	(10, 50, 3) [11, x, 1] accuracy: 5.6% time: 52m
K-Medoids	(10, 50, 3) [19, x, 7] accuracy: 18.1% time: 48m59s	(10, 50, 3) [6, x, 4] accuracy: 22% time: 49m	(10, 50, 3) [8, x, 28] accuracy: 43% time: 98m	(10, 50, 3) [9, x, 1] accuracy: 8.9% time: 5m	(10, 50, 3) [12, x, 1] accuracy: 13% time: 47m	(10, 50, 3) [11, x, 1] accuracy: 11% time: 112m

Figure 6: RBF Results - tuple values are (max iterations, batch size, eta), and square brackets denote the layer sizes for each network.

5. Analysis

The long running times required to train these networks made it difficult to do meaningful comparisons between networks and parameter/network configurations. However, we still have interesting results that lead to a few insights.

In general, we found that simpler network configurations led to the best results. For example, our overall best prediction accuracy results were achieved with the Segmentation data using the MLP network with 0 hidden layers and 50 iterations of gradient descent, reaching 83.3% prediction accuracy. However, when we changed the network to have 1 hidden layer with 19 neurons, our prediction accuracy on the same data dropped to only 22.9%. And when we changed the network again to have 2 hidden layers with 10 and 5 neurons respectively, our prediction accuracy further dropped to 18%. So it seems like adding layers only resulted in decreasing performance.

Decreasing performance with respect to increasing layer complexity likely is expected in this case. If there is no particular structure in the data that could potentially be picked up by hidden layers, then adding hidden layers will only serve to complicate the network function, consequently making it more difficult to train, and more difficult to achieve satisfactory performance. Although adding hidden layers can, in some cases, lead to better results, if the relevant underlying structure is missing from the data, better accuracy is not likely to be achieved by simply adding layers in a random fashion (Hornik et al. (1989)).

In contrast to the Segmentation data set, we found that the Car data set performance using the MLP network actually increased with adding hidden layers. The initial MLP configurations with 0 hidden layers achieved a prediction accuracy of 73.7%, which increased to 81.3% with adding one hidden layer, and increased again to 84.9% with adding a second hidden layer. In this case, this likely means there is more inherent structure in the Car data, which was picked up by the hidden layers, leading to an overall increase in prediction accuracy for the Car data.

One thing we noticed while testing and running the experiments is that the performance results are sensitive to the initial random configuration of weights and biases. Each cross validation partition would start gradient descent with a different initial accuracy/error value, and the starting point often determined how much the results could improve over the gradient descent iterations for that partition. This is a known obstacle that hinders all neural networks trained with gradient descent. Put more formally, the initial location of the parameter vector in the overall topological space of the cost function plays a significant role in determining which local minimum is reached (Erhan et al. (2010)). Momentum can help alleviate this issue, but in many cases, the final results are dependent on the initial starting point for gradient descent.

6. Conclusion

This project involved a significant amount of programming, along with a fair amount of reasoning about the data sets and how best to approach and analyze each one. Ideally, we would have been able to test many parameter combinations for each test in an attempt to maximize accuracy. However, as runtimes were long, generating meaningful results proved difficult. With that said, we did manage to get good results on the Segmentation data set using 0 hidden layers, and improved results on the Car data set using extra hidden layers. Our main takeaway from this work is that each data set needs to be analyzed and tuned independently, using a wide range of parameter values and network configurations, in order to gain insight into which configurations achieve the best performance results, both in terms of prediction accuracy and training time.

References

- Dave Myszewski Caroline Clabaugh and Jimmy Pang. Neural networks-architecture. 2000.
- Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.
- Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
- L.M.C. Buydens G.Kateman J.R.M.Smiths, W.J.Melssen. Using artificial neural networks for solving chemical problems: Part i. multi-layer feed-forward networks. *Using artificial neural networks for solving chemical problems: Part I. Multi-layer feed-forward networks*, 22:165–189.
- Ron Kohavi. A study of cross validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 1995.
- Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018. URL <http://arxiv.org/abs/1804.07612>.
- Mathworks. What is global optimization? 2019.
- Chris McCormick. Radial basis function network. 2013.
- Deshpande Mohit. Using neural networks for regression: Radial basis function networks. 2017.
- S.R Dorling M.W Gardner. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric Environment*, 32(Aug):2627–2636, 1998.
- Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.