Analysis of Neural Network Architectures For Supervised Classification

Abhinandan Dubey
Stony Brook University
New York, USA

Teja Madiraju
Stony Brook University
New York, USA

Bharath Kumar Darapu
Stony Brook University
New York, USA

[adubey, lmadiraju, bdarapu]@cs.stonybrook.edu

Abstract

Many supervised learning algorithms are available today for solving prediction problems. Of them, artificial neural networks have been consistently known to perform well for most classes of problems. In our project, we perform a high-level comparative analysis of five different neural network architectures. The motivation behind this series of experiments is to understand the nature of these neural networks in the backdrop of certain basic variations in the dataset.

Introduction

Problem : To analyze the performance of various types of neural network architectures including Feed-forward Neural Networks and Competitive Networks over a supervised classification problem using various metrics.

Motivation : With many choices being available for data scientists and industry experts dealing with classification tasks, it becomes critical to analyze the performaces of these models on varieties of problems available to us. In 5 years, we can expect neural networks to manage services and ports (Karagiannis 2016). Thus, we need to perform a holistic analysis which gives us results for these classification problems

Contributions : For our analysis, we performed a backward feature selection algorithm to select the most dominant features for our dataset which involves the identification of credit defaulters based on their demographics and history of accounts. On the best selection set of features, we implemented and analyzed the performance of the most commonly used neural network architectures:

1. Single Layer Perceptrons (SLP)
2. Multilayer Perceptrons (MLP)
3. Hopfield Recurrent Networks
4. Learning Vector Quantization (LVQ)
5. Competitive Layer - Kohonen Network

Learning Outcomes : From our extensive analysis, we were able to understand the inner working of various neural networks which have been explained below. Additionally, we were able to analyze the performance of different neural network architectures for varieties of classification task. We learnt that neural network models crafted specifically for pattern finding perform poorly at binary classification tasks involving a set of training data.

Neural networks are highly adaptable models which get trained over abstract functions that are built up from a cascade of interleaved layers, each of which is a group of perceptrons termed as 'nodes.' Although a single perceptron is just a binary linear classifier which partitions space into different categories or classes. A perceptron has a capability of deciding whether a set of inputs lie within or outside a boundary, which is a linear $n-1$ dimension hyperplane for n input features.

The linearity of these baseline models prevents them from being used in training on datasets with non-linear function mappings. Thus, we introduce some non-linearity using `ReLU` or *Rectificied Linear Units*. A smoothing function approximation to such rectifier is the analytic function defined as;

$$f(x) = \ln(1 + e^x)$$

which is called the softplus function. More specifically, perceptrons associate a weights w_i with each feature to calculate. Thus creating a lot of weights that are initialized with random values. The goal of training is to get the appropriate values for these weights. The whole neural network model works by calculating the values using weighted sums over a threshold function that binarizes the output.

The choice of the smoothing functions can cause huge variations in the metrics. It is critical for a model to use appropriate functions. For our analysis, we have used the sigmoid function wherever possible. The sigmoid function is a special case of the logistic function defined as;

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

We use the library `neurolab` (Evgenij 2013) which provides helper functions for creating and training these networks. We use `sklearn` (Pedregosa et al. 2011) for evaluation purposes.

Description

Backward Feature Selection

We implemented a backward feature selection algorithm starting with 11 features and selectively removing the features which corrupt the learning models. This allowed us to achieve considerably good accuracies. We have tried to include as much detail as possible about it in the report. The full results can be found in the results files in the code attached with our submission.

Single Layer Perceptron

The single layer perceptron is the most basic feed-forward neural network. The input that we have supplied to it are 22498x6-element feature vectors.

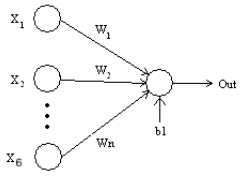
The perceptron is effectively a function that maps its input x (a real-valued vector) to an output value. The goal is to compute the sum:

$$\sum_{i=0}^{m} w_i x_i$$

, where m is the number of inputs, which is to the perceptron and b is the bias. The bias can shift the decision boundary away from the origin. The bias value cannot be supplied explicitly, however, the input feature ranges have to be supplied to the library function as:

```
net = nl.net.newp([[-7, 7]]*6, 1)
```

Figure 1: Single Layer Perceptron



Multilayer Perceptron

A Multi Layer Perceptron (MLP) extends the idea of single layer perceptron to multiple layers. An MLP contains one or more hidden layers (apart from the input and the output layer). The basic idea is to extend Single Layer Perceptron to be able to learn non-linear functions.

We represent the error in output node j in the nth data point (training example) by

$$e_j(n) = d_j(n) - y_j(n)$$

where d is the target value and y is the value produced by the perceptron. In our case, y is a binary value 0 or 1 depending upon the class of the output. The goal is to update the weights of the nodes based on those corrections which minimize the error in the entire output, given by

$$\mathcal{E}(n) = \frac{1}{2} \sum_j e_j^2(n)$$

Using gradient descent, we find our change in each weight to be

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)}$$

where y_i is the output of the previous neuron and η is the learning rate, which is carefully selected to ensure that the weights converge to a response fast enough, without producing oscillations. This is taken care of, by our library, however, one can pass a learning rate value. We have defined our network as

```
net = nl.net.newff([[-7, 7]] * 6,
[hidden_nodes, 1], transf =
[nl.trans.LogSig()] * 2)
```

It is important here to note that we have explicitly supplied the Sigmoid function to be used. The `hidden_nodes` denote the number of hidden layer nodes you want in the network. We have used 2 and 3 nodes to train on our models and evaluate their performance.

Competitive Layer - Kohonen Network

A Kohonen network is composed of an initial input layer and an a preset output layer with multiple competing units in the hidden layer in between. The network learns on the training patterns provided to the input layer. The neurons in the competing layer compete to respond to the input pattern. The unit whose weights are closest to the current input (Euclidean distance), becomes activated as the layer influencing the input. It is called the Best Matching Unit (BMU). Based on the position of the BMU, the neighboring neurons are adjusted for weights. The weight to adjust for each neighbor is given by the below formula.

$$n_i(t+1) = n_i(t) + h * [v(t) - n_i - n_i(t)]$$

$n(t)$ = weight vector of neuron i at regression step t
$v(t)$ = input vector at regression step t
h = neighborhood function

Each set of feature values in our data set was treated as an input pattern. Each output of the network is a tuple that represents the result of the classification on the data point. The Neurolab implementation used in our calculations is given as below

```
net = neurolab.net.newc(minmax, cn)
```

$minmax$ is the range of values for each of our 6 features
cn is the number of neurons in the output layer.

Learning Vector Quantization

Learning Vector Quantization (LVQ) is a classification technique implemented as a supervised version of vector quantization. It is similar to a Self-Organizing Map (SOM) with input vectors (x) and weight vectors except (W_i) that it has associated class information on its input data points. The LVQ algorithm starts from a trained SOM (same as the one used in Kohonen Network) and uses the classification labels of input data to generate the classification labels of each W_i, the nearest neighbor neurons. Each input cell without a class label can be assigned to the class of the nearest neighbor cell

it falls within. The Neurolab implementation used in our calculations is given as below

```
net = neurolab.net.newlvq(minmax, cn0,
                          pc)
```

$minmax$ is the range of values for input features
$cn0$ is the number of neurons in the input layer
pc is the percent list that sums to 1

Hopfield Networks

The purpose of a Hopfield net is to store specific patterns and to recall the target patterns based on some (generally distorted) input patterns. Our problem had some training features. We used these training features over a HopField Network that accepts six features as the input. The network gets trained to identify such patterns and produces a most likely pattern as an output. NeuroLab implements them as:

```
net = nl.net.newhop(target)
```

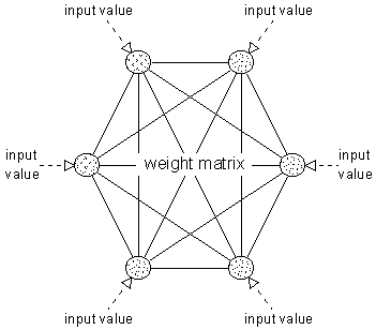
The variable `target` denotes the floating point values for the target feature patterns that have to be identified in the input. More specifically, since Hopfield nets can serve as content-addressable memory systems with binary threshold nodes, we are just creating a fully connected 6-layer network [1] which can adapt according to these training data patterns. Hopfield Network uses The Hebbian Theory (Hebb 2002) for training the neurons. It can be briefly understood as
"Neurons that fire together, wire together. Neurons that fire out of sync, fail to link".

The Hebbian rule is both local and incremental. For the Hopfield Networks, it is implemented in the following manner, when learning n binary patterns:

$$w_{ij} = \frac{1}{n} \sum_{\mu=1}^{n} \epsilon_i^\mu \epsilon_j^\mu$$

where ϵ_i^μ represents bit i from pattern μ.

Figure 2: Hopfield Network



[1] self loops hold zero weights

Evaluation

Single Layer Perceptron

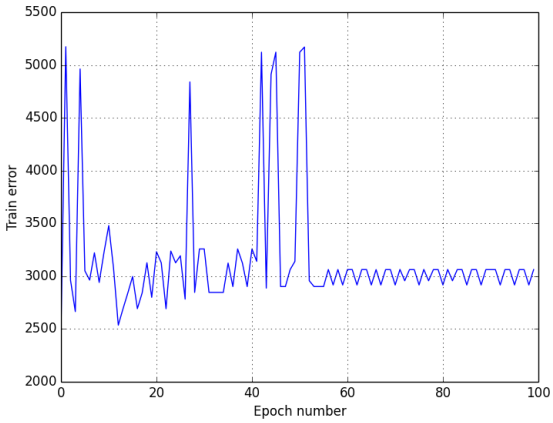
The single layer perceptron model achieves a quite good accuracy with an average precision of 0.73. As shown below, the single layer perceptron model is a baseline model and thus this much of accuracy is expected.

Table 1: Evaluation - Single Layer Perceptron

	Precision	Recall	F1 Score	Support
0 - Non-Defaulter	0.83	0.84	0.83	5842
1 - Defaulter	0.41	0.38	0.39	1660
Average	0.73	0.74	0.74	7502

The most likely reason is the linearity of the functions that can be trained in a single layer perceptron. Since the model cannot learn non-linear functions, it won't be able to achieve a great acccuracy. The figure shows the training over the epochs.

Figure 3: Single Layer Perceptron - Sum-Squared Errors



Multilayer Perceptron

The multilayer perceptron achieves a very good accuracy and precision

Table 2: Evaluation - Multilayer Perceptron - 3 Hidden Nodes, 6 Features

	Precision	Recall	F1 Score	Support
0 - Non-Defaulter	0.84	0.96	0.90	5842
1 - Defaulter	0.73	0.35	0.47	1660
Average	0.81	0.83	0.83	7502

The most likely reason of such high accuracy (compared to other models) is likely because of the fact that multilayer perceptron can deal with non-linearity very effectively because of having a cascade of layers.

Figure 4: Multilayer Perceptron - 11 Features

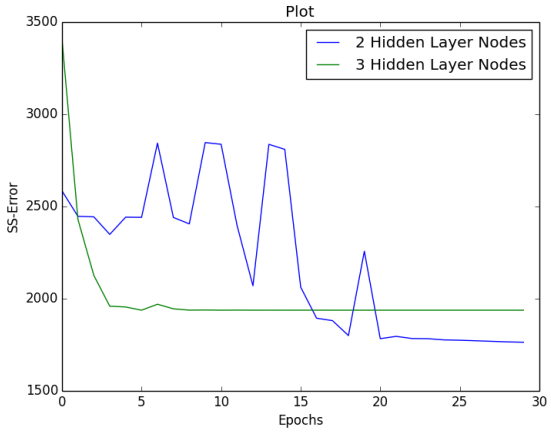
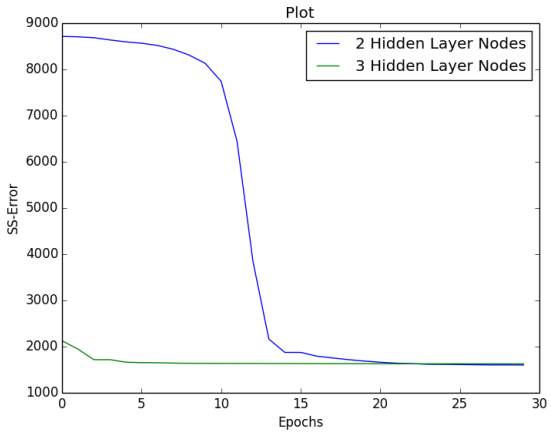


Figure 5: Multilayer Perceptron - 6 Features



Competing Layer- Kohonen Networks

The evaluation measures obtained upon training the data set on Kohonen Network indicate that the network has a good performance. Kohonen Network works well on our dataset because of the consistent nature of distribution of data resulting in the presence of sufficient number of completely defined input patterns. As a result, the self-organizing nature of the weights on the competing layer is strengthened to produce good predictions on test data.

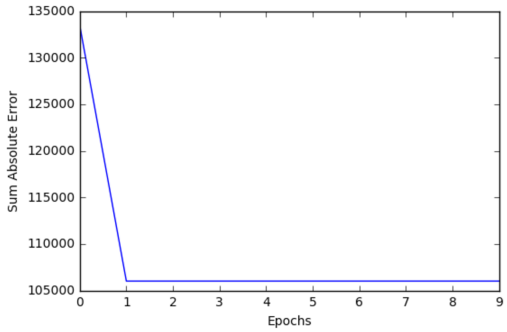
Table 3: Evaluation - Competing Layer- Kohonen Networks

	Precision	Recall	F1 Score	Support
0 - Non-Defaulter	0.84	0.96	0.88	5842
1 - Defaulter	0.62	0.36	0.46	1660
Average	0.79	0.81	0.79	7502

Learning Vector Quantization

The strength of the LVQ technique lies in its ability to classify data inputs properly. However, our data set comes with

Figure 6: Kohonen Network - Sum-Squared Errors



predetermined classification of a user being a defaulter or othwerwise. This is possibly why LVQ produces distinctly low results on our datasets prediction problem. There are almost no unclassified input data to be grouped according to the class similarities, therefore resulting in low F1 score. The below table is an evaluation of the LVQ implementation on our data set.

Figure 7: Learning Vector Quantization - Sum-Squared Errors

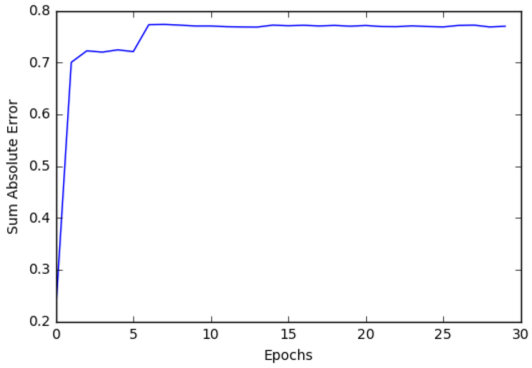


Table 4: Evaluation - Learning Vector Quantization

	Precision	Recall	F1 Score	Support
0 - Non-Defaulter	0.76	0.02	0.03	5842
1 - Defaulter	0.22	0.98	0.36	1660
Average	0.64	0.23	0.11	7502

Hopfield Networks

The metrics obtained upon the results clearly show that Hopfield Networks perform poorly on basic classification task which cannot be properly framed as pattern-recognition problems. Another reason why their performance was considerably lower than the other network architectures is the converging criterion. Although Hopfield Networks are guaranteed to converge to a local minimum, they might converge to a wrong local minimum denoting a false pattern rather than the actual pattern (expected local minimum). With the

size of the training data (24000) rows, we can expect that there will be a lot of noise added to the pattern and thus the network fails to converge to an expected local minimum.

Table 5: Evaluation - Hopfield Networks

	Precision	Recall	F1 Score	Support
0 - Non-Defaulter	0.86	0.86	0.86	5842
1 - Defaulter	NA	NA	NA	1660
Unclassified	NA	NA	NA	0
Average	0.67	0.67	0.67	7502

This makes it impossible for the network to identify some test inputs and thus it renders a fraction of the test dataset ”unclassifiable”.

Conclusion

Most neural networks techniques are good candidates for classification problems. However, based on the inherent properties of the data set (for example, in the case of LVQ implementation) and the architecture of the network in use, we cannot generate a 'one fit for all' model. From our experiments, we conclude that the Multi-layer Perceptron is a better neural network model among the variants we tried. A possible reason is that the MLP network deals is a good fit for data sets with non linear properties.

References

Evgenij, Z. 2013. Neurolab. In *A simple and powerful Neural Network Library for Python*.

Hebb, D. 2002. The organization of behavior: A neuropsychological theory. *JLawrence Erlbaum*.

Karagiannis, K. 2016. Rise of hacking machines. In *RSA Conference - San Francisco, Moscone Center*.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, E. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12:2825–2830.