Creating a Single Layer Neural Network From Scratch

Ayala Zecharia (1221193285)
Caleb Jones (1219212228)
Hibah Usmani (1218793810)

Dr. Robert McCulloch
STP 540 - Computational Statistics

May 7th, 2025

Introduction

One machine learning method that can be used as a predictive model in statistics is a neural network. It implements the ideas behind how neurons work in the human brain connecting many nodes in order to process information and make conclusions (IBM). A neural network can contain many hidden layers to connect the input to the output using nonlinear functions throughout.

Neural networks can be used for prediction or classification. Some examples of their use include image recognition, medical diagnosis, financial predictions, and pattern recognition (SAS). They are especially powerful when trying to model complex situations with a large amount of training data that would not be possible with simpler models in exchange for less interpretability (James).

One of the downsides of neural networks is that they are known to be a "black box" tool where it can be unclear how an output was determined. However, by learning the statistical concepts behind neural networks, the question of how they work can be answered and applied to actually create one. For this project, the goal was to build a single layer neural network from scratch. This meant coding the essential concepts of the model in R including the activation function to non-linearly transform a linear combination, the forward pass using input data to obtain a prediction, and gradient descent to update the parameters of the forward pass. The resulting neural network can then be tested on both simulated and real datasets to confirm that it runs and makes predictions as expected.

Methodology

The single-layer neural network iterates through two processes – the forward pass and the backward pass – where each iteration is referred to as an epoch. In the forward pass, linear combinations are taken of the observations and transformed to produce a prediction. In the backward pass, the weights are then updated using gradient descent and used in the next iteration of the forward pass. This process continues until the loss function has been minimized. The methodologies described in this section are a vectorized extension of the content described in *Introduction to Statistical Learning* by Gareth James, et al.

Forward Pass

In the input weight matrix, W, a given row contains the weights corresponding to the k^{th} unit of the hidden layer, including an intercept term (w_{k0}). To obtain a weighted value for a single observation, a row from W is multiplied by a row from the transposed data matrix X. For the k^{th} unit, the weighted summation of the predictors across all observations is captured in a row of:

$$Z = WX^T, \quad X \in \mathbb{R}^{n\times(p+1)} \quad W \in \mathbb{R}^{k\times(p+1)}$$

where X contains a column of ones to account for the input weight intercept w_{k0}. All elements in the Z matrix are then passed through a nonlinear function, referred to as the activation function. Here, the sigmoid function was used:

$$A = \frac{1}{1+e^{-Z}}, \quad A \in \mathbb{R}^{(k+1)\times n}$$

where A contains a row of ones to account for the output weight intercept b_{k0}. Each row in the activation matrix A – the transformed linear combination of predictors for a given unit – is then multiplied by a weight b_k to obtain a final prediction:

$$f(X) = \beta A, \quad f \in \mathbb{R}^{1\times n}$$

The prediction is then evaluated using a loss function. More specifically, the mean square error:

$$L = \frac{1}{n}\sum_{i=1}^{n}(y-f)^2$$

Backward Pass

In the backward pass, the loss function is partially derived with respect to each of the input and output weights. Using the chain rule, the gradient vector for the output weights is:

$$\nabla L_\beta = \frac{\delta}{\delta\beta}\left[\frac{1}{n}\sum_{i=1}^{n}(y-f)^2\right] = \frac{\delta L}{\delta f}\cdot\frac{\delta f}{\delta\beta} = -\frac{2}{n}(A\cdot(y-f))^T, \quad \nabla L_\beta \in \mathbb{R}^{1\times(k+1)}$$

Similarly, the gradient matrix for the input weights is:

$$\nabla L_W = \frac{\delta}{\delta W}\left[\frac{1}{n}\sum_{i=1}^{n}(y-f)^2\right] = \frac{\delta L}{\delta f}\cdot\frac{\delta f}{\delta A}\cdot\frac{\delta A}{\delta Z}\cdot\frac{\delta Z}{\delta W} = -\frac{2}{n}\left[(y-f)\cdot\beta\cdot A(1-A)\right]^T X, \quad \nabla L_W \in \mathbb{R}^{k\times(p+1)}$$

where the output weight vector B doesn't contain the intercept term b_{k0}. Each gradient is multiplied by a scalar α – the learning rate – and then subtracted from the corresponding weights to produce updated weights for use in the next forward pass.

Stochastic Gradient Descent

To improve the computational efficiency of the backward pass, neural networks typically employ stochastic gradient descent. Here, the data is partitioned into disjoint batches where an iteration through all batches is considered a single epoch. For each batch, the gradients are calculated based on the corresponding partition of data and the weights are updated accordingly (McCulloch).

Adaptive Learning Rates

Adaptive learning rates are customized learning rates for individual parameters and avoid overly-aggressive adjustments to the weights (Santosh). Adaptive Gradient (adagrad). divides the fixed learning rate by the square root of the summation of the prior and current second-order gradients plus an arbitrary epsilon.

$$\theta_{i,t+1} = \theta_{i,t} + \frac{\alpha}{\sqrt{A_{i,t+1}+\epsilon}}\nabla L_{i,t}, \quad A_{i,t+1} = A_{i,t} + \nabla L_{i,t}$$

This reduces the learning rate for a parameter with historically high gradient values (McCulloch). However, the Adaptive Gradient accumulates all of the second-order gradients and will consequently decrease the learning rate for each iteration, potentially causing non-convergence (GeeksforGeeks). The Root Mean Square Propagation (rmsprop) addresses this drawback by taking a moving average of the second-order gradient and introducing a smoothing constant ρ (PyTorch):

$$\theta_{i,t+1} = \theta_{i,t} + \frac{\alpha}{\sqrt{A_{i,t+1} + \epsilon}} \nabla L_{i,t}, \quad A_{i,t+1} = \boldsymbol{\rho} A_{i,t} + (\mathbf{1} - \boldsymbol{\rho}) \nabla L_{i,t}$$

Data

To test the efficacy and debug the neural network, simulated data was used. The first step was drawing $n * p$ values from a uniform distribution such that $X \sim Uniform(a, b)$. Then, the values were restructured into a $n\ x\ p$ matrix.

$$\mathrm{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}$$

Next, the function $y_i = \sum_{j=1}^{p} [(x_{ij})^2 + \epsilon_i]$ was used, where $\epsilon_i \sim Normal(0, \sigma^2)$, to draw from the conditional distribution y|X. Essentially, each column of the X matrix was squared, then the y vector was created by taking all the row sums and adding a random error term. Thus, when the simulation was done on only one p, the equation simplified to $y = x^2 + \epsilon$, which is a quadratic equation with some additional random error. Throughout the process, many parameters were tested, including a, b, p, n, and σ^2.

Besides the simulated data, three of the built-in R datasets were used as inputs for the single layer neural network. The "iris" dataset was the first one used (Anderson, Fisher). It contained the widths and lengths of the sepals and petals of 150 flowers separated into three species. Although this data is typically used for classification, for this project it was used to predict the petal length of each flower using the provided sepal width and length.

The next R dataset was "mtcars" containing car road test information from *Motor Trend* magazine in 1974 (Henderson and Velleman). There were 32 observations with columns like displacement, gross horsepower, rear axle ratio, weight, and quarter mile time. These five variables were used to predict the miles per gallon of each car.

The third and final built-in R dataset used was "airquailty" with 153 days of air quality measurements taken from New York in 1973 (Chambers). These included measurements of the ozone, solar, wind, and temperature with the first three variables being used to predict

temperature. Observations with missing values in any of the columns were removed from the dataset leaving 111 observations before the neural network was trained. In practice, it would be best to impute the missing values with a neutral value like the median.

For each of these datasets, all observations were used to train the neural network with 25 units in the single layer over 1,000 iterations. The type of adaptive learning rate used was rmsprop with values of $\rho = 0.9$.

Results

During the first attempt at building the single layer neural network, the main goal was to create functions that split up the statistical methods. Originally, the functions for back propagation with gradient descent used loops as a way to ensure that the calculations were completed correctly. The neural network was trained using 10 iterations with a single, simulated observation and a single layer unit (p=1, n=1, k=1). The mean squared error loss was 232.091 initially, but after updating the parameters during the backward pass while training, the loss dropped to 0.023. This process was then repeated but with vectorized versions of the functions using matrix operations in R which resulted in the same output. This careful construction of the neural network functions reassured that the code worked as intended before moving on to other variations of the input data.

The created single layer neural network was then trained with 25 units over 100 iterations on simulated data which contained a single predictor with 50 observations. In this case, the loss dropped from over 57,000 to 39. Figure 1 shows the decrease in loss across the training iterations.

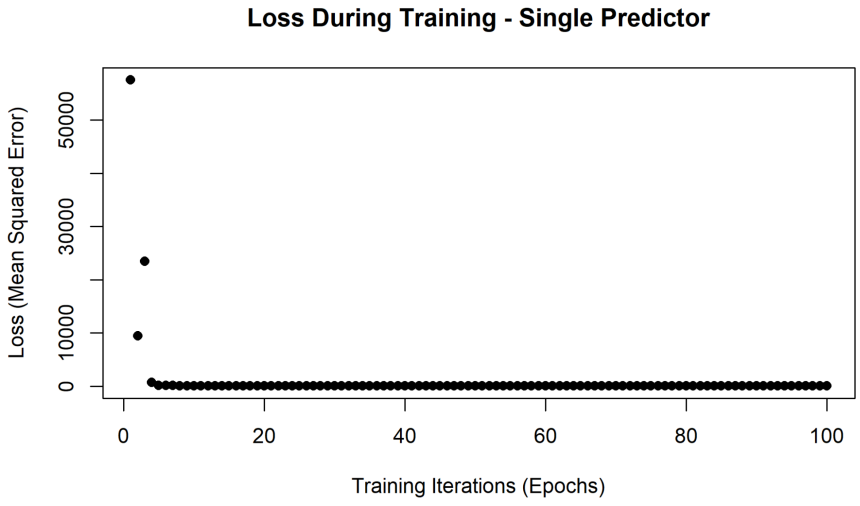

Figure 1: Training Loss with Single Predictor

At this stage, one way that the neural network was improved was with the use of an adaptive learning rate as opposed to a fixed learning rate. This was incorporated into the functions that computed the gradients of the loss function with respect to each parameter during back

propagation. With adaptive learning rates, two main methods were considered and coded in the functions: adagrad and rmsprop. For this project, focus was placed on rmsprop. To test this new addition, the same setup as the single predictor with a fixed learning rate example was used, but this time with smoothing constant of $\rho = 0.9$. The loss over the training iterations, shown in Figure 2, does not immediately drop like in Figure 1. However, the final loss value is lower at 6.47 compared to the 32 before which was an improvement.

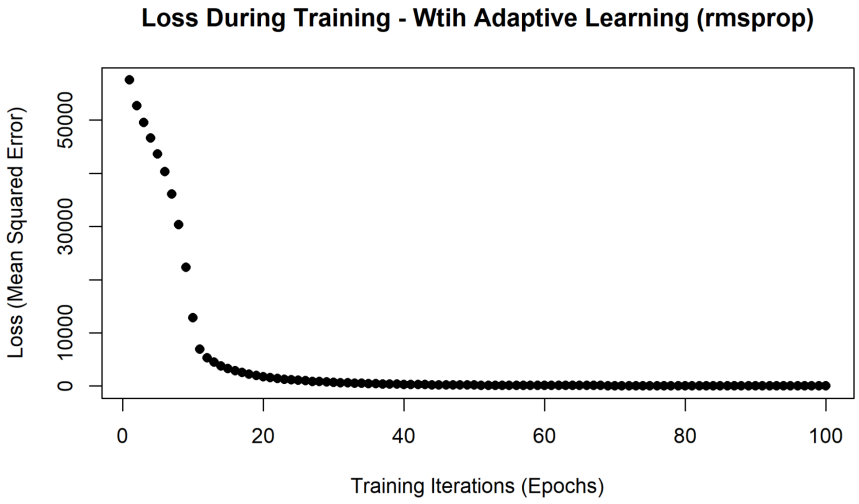

Figure 2: Training Loss for Single Predictor with Adaptive Learning Rate

With adaptive learning rates set up, the neural network was then tested on multiple predictors as opposed to a single predictor. In this case, there were five predictors, 100 observations, and 100 units in the single layer. The training was done over 1,000 iterations. The loss ended up following the same decreasing pattern seen with the single predictor, but the loss starts off very high at around 12 million and takes more than 100 iterations to level off. After the full training loop was run, the final mean squared error loss was 13.59 which was a substantial improvement compared to the initial loss value.

Due to the addition of more computationally complex methods, stochastic gradient descent was used in hopes of reducing the time until convergence of the loss metric. This approach required the data to be partitioned. We chose to complete this step by randomly assigning the data to 10 different groups such that each group was disjoint. In order to accurately compare, the same parameters from the previous step were used, including the number of epochs. Therefore, the model completed 10,000 training iterations, $epoch * number\ of\ batches$. Once again, shown in Figure 3, the loss initially started at 12 million and took many iterations to reach a more stable value. Additionally, after the last training loop, the resulting mean squared error loss was at 0.788. This outcome illustrated how effective stochastic gradient descent was at more efficiently utilizing the data.

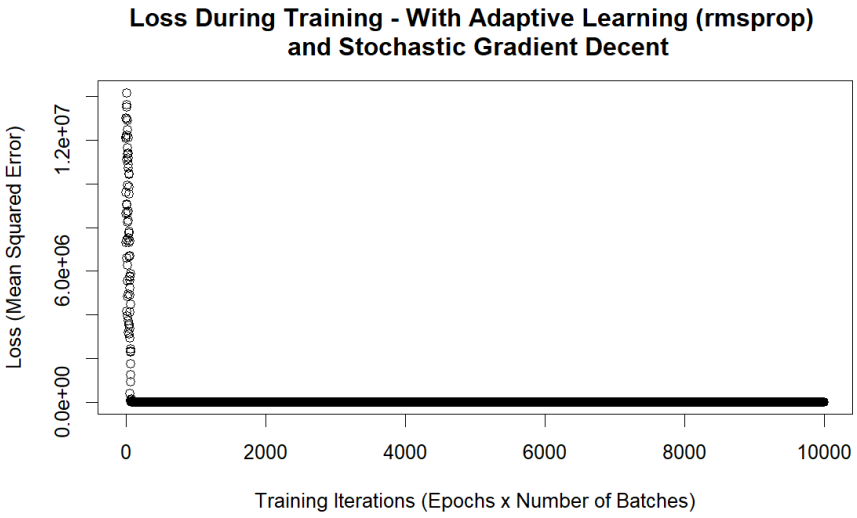

Figure 3: Training Loss for Multiple Predictors with Adaptive Learning Rate and Stochastic Gradient Descent

The final neural network had a single layer with 25 units, an adaptive learning rate method of rmsprop with $\rho = 0.9$, and training over 1,000 iterations. It was then tested on both simulated data and built-in datasets from R. The resulting loss was then compared to the final loss from other versions of the neural network created. The results are summarized in the table below. The last loss values for the iris, MT cars, air quality datasets with the final neural network model were 0.565, 66.484, and 132.820 respectively.

Table 1: Final loss across different code versions and datasets

	Gradient Descent		**Stochastic Gradient Descent (10 batches)**	
	Fixed	**Adaptive**	**Fixed**	**Adaptive**
Simulated Training Data (p=5, n=750)	53.33168	29.36071	193.9947	16.80547
Simulated Test Data (p=5, n=250)	106.5133	49.23754	257.1862	24.79511
Iris Data (p=2, n=150)	3.095503	3.118004	3.122788	0.5649942

For each dataset, the minimum loss occurred when the neural network used stochastic gradient descent and an adaptive learning rate. Regardless of the type of descent, the simulated data worked better with the adaptive learning rate over a fixed learning rate. Figure 4 shows the loss throughout the training iterations that led to the final loss of 16.85 shown in the table. When considering both types of learning rates, stochastic gradient descent worked better overall. For the iris dataset, the loss was consistent between gradient descent with fixed learning rate,

gradient descent with adaptive learning rate, and stochastic gradient descent with fixed learning rate. However, there was a decrease in the final loss when stochastic gradient descent with adaptive learning rate was used.

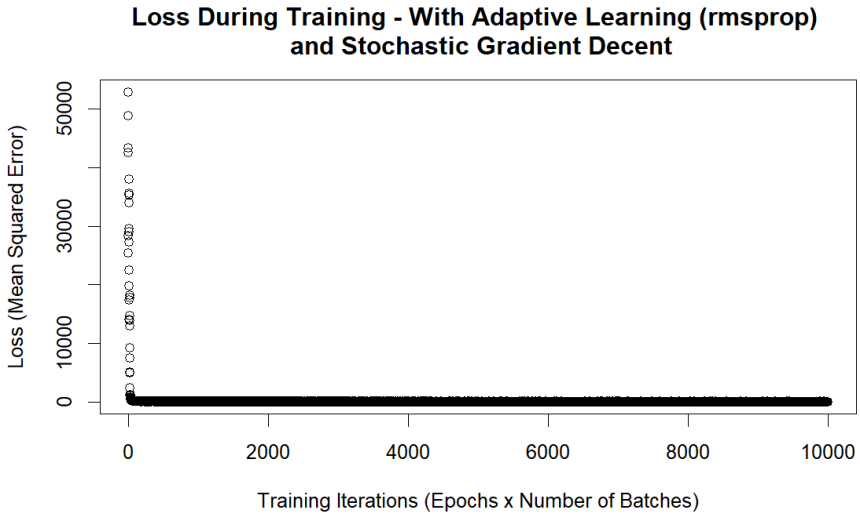

Figure 4: Training Loss for Simulated Data with Stochastic Gradient Descent and Adaptive Learning Rate with Final Hyperparameters

Discussion

Based on the analysis, there were some key results that should be noted. First, vectorization was an integral piece in reducing run time. With the use of matrix algebra in R, versions of the neural network which required intensive training took less time than if loops were used. For example, the version with stochastic gradient descent with adaptive learning rate used 10,000 overall iterations based on the number of iterations and number of batches defined.

Another result was that stochastic gradient descent underperformed compared to regular gradient descent when a fixed learning rate was used. A potential reason for this was that the defined learning rate of 0.1 was too large. It was chosen based on results using the full training dataset, but with the stochastic method, there are less observations per iteration leading to more bias. A smaller learning rate would lead to smaller updates to the parameters, resulting in more stability when working with batches that may contain influential outliers. After the initial results, a learning rate of 0.01 was used which resulted in a training loss of 1.82 and test loss of 2.99 for the simulated data which showed that this solution could work.

Then, it follows that the best performing model was the one that included an adaptive learning rate and stochastic gradient descent. The implementation of an adaptive learning rate aimed to dynamically adjust the learning rate at each iteration for each weight parameter, where rmsprop utilized a moving average parameter to accomplish this. We used a relatively large weight of $\rho = 0.9$, which is the default value in the Keras and PyTorch libraries. Currently, the rho weight is

not optimal and would benefit from a hyperparameter search. Because of this, the loss still took a considerable number of iterations, 1,000 epochs, to reach a plateau.

Of all the datasets analyzed during the project, the iris dataset had the smallest loss by the end of the neural network training. One potential reason for this was that the dataset was less complex compared to the simulated data since it had a small number of variables and a small number of observations. Another possible reason was that the distribution of the input variables was different from the uniform distribution used to simulate the data. Also, the iris data was not split into train and test sets, so overfitting may have occurred which led to the small loss.

The last key result that should be noted was the drop in performance between the train and test simulated datasets across all versions of the neural network. This could be due to overfitting where the neural network performed well on the data it was trained on but cannot generalize to new data. A potential fix for this would be to utilize a validation dataset to see what combination of the number of units in the single layer and number of training iterations best generalizes to unseen data.

Limitations

One consistent limitation was the time constraint of the project. A considerable amount of time was spent vectorizing the backward pass. This required not only a full understanding of the matrix algebra but also extensive trial and error in the coding stages. The process involved testing the accuracy of every addition made to the code using simple simulated data sets, then increasing the complexity to test the overall performance. Overall, a majority of the project timeline was dedicated to debugging code relating to matrix calculations and dimensions.

Another closely linked restraint was the runtimes of the code. Our final model took a considerable number of epochs for the loss to converge to a value near zero. This issue arose after increasing the number of predictors and introducing an adaptive learning rate. While without vectorization, the code would be too computationally intensive, the addition of the more complex methodology led to much longer runtimes.

Next Steps

In the future, to optimize the overall performance, hyperparameter tuning or penalization should be utilized. Due to the signs of overfitting in the neural network that used stochastic gradient descent, the overall model would benefit from some form of regularization. Adding an L2 penalty works by shrinking the values of the less important weights. Before this can be done, the input features must be scaled to ensure that the penalty affects the weights uniformly. Overlooking this step could lead to underfitting and general problems with the predictions. Moreover, the lambda parameter will have to be tuned to ensure that the model contains the ideal number of hidden layer units. This parameter controls how much importance is given to previous iterations' estimates. Similarly, the number of epochs should be chosen carefully. If the training

goes through too many iterations, then it runs the risk of attributing too much importance to the random noise and irrelevant fluctuations in the data.

A large number of units is preferred in regularization techniques like one mentioned above. However, this can cause computational inefficiencies. To address this, the use of another adaptive learning rate, Adaptive Moment Estimation (Adam) should be explored (Kingma). Similar to the Root Mean Square Propagation, this technique adjusts the fixed learning rate based on the moving average of the second-order gradients. However, it also incorporates another smoothing constant and takes the moving average of the first-order gradients, analogous to the concept of momentum used in stochastic gradient descent (McCulloch).

Additionally, the neural net should be tested on more real-world data to assess the flexibility of the algorithm. For instance, Gross Domestic Product (GDP) and other economic factors, such as employment rates and inflation, can be compiled and used to predict the likelihood of a recession. Due to the format of the current data, each feature would need to be downloaded and merged together to create the full data set. The target of the model would be GDP due to its significance as a recession indicator and domain that falls within the range of the neural network constructed in this project. This implementation would demonstrate the ability of the model to handle applied, high-dimensional data and provide insights into macroeconomic trends.

References

Adagrad optimizer in deep learning. (2020, November 25). GeeksforGeeks. https://www.geeksforgeeks.org/intuition-behind-adagrad-optimizer/.

Anderson, Edgar (1935). The irises of the Gaspe Peninsula, *Bulletin of the American Iris Society*, **59**, 2–5.

Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A. (1983) *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, Part II, 179–188. doi:10.1111/j.1469-1809.1936.tb02137.x.

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391–411.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An Introduction to Statistical Learning: With Applications in R*. Springer US. https://doi.org/10.1007/978-1-0716-1418-1.

Kingma, D. P., & Ba, J. (2017). *Adam: A method for stochastic optimization* (arXiv:1412.6980). arXiv. https://doi.org/10.48550/arXiv.1412.6980.

McCulloch, R. (2025) *Neural Networks*. Arizona State University. https://www.rob-mcculloch.org/2025_cs/webpage/notes/nnet.pdf

McCulloch, R. (2025) *Optimization*. Arizona State University. https://www.rob-mcculloch.org/2025_cs/webpage/notes/opt_23.pdf

Neural Networks: What are they and why do they matter? (n.d.). SAS. https://www.sas.com/en_us/insights/analytics/neural-networks.html.

R Core Team (2024). _R: A Language and Environment for Statistical Computing_. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/.

RMSprop—PyTorch 2.7 Documentation. (n.d.). https://docs.pytorch.org/docs/stable/generated/torch.optim.RMSprop.html.

Santosh, K., Das, N., & Ghosh, S. (2022). Chapter 2 - Deep learning: A review. In K. Santosh, N. Das, & S. Ghosh (Eds.), *Deep Learning Models for Medical Imaging* (pp. 29–63). Academic Press. https://doi.org/10.1016/B978-0-12-823504-1.00012-X.

What is a Neural Network? (2021, October 6). IBM. https://www.ibm.com/think/topics/neural-networks.