NLS for GPS Localization, Custom Two-Layer Neural Network

Matthew Sun

Abstract

The first purpose of this assignment was to formulate a GPS localization problem as a nonlinear least squares problem and then solve it using the MATLAB function "lsqnonlin()". The residual errors between each satellite's Cartesian coordinate and their respective reading were used to construct the objective function. The function was implemented with the default settings and an augmented setting that specifies the use of the Levenberg-Marquardt algorithm. Each setting was tested with three different initial estimates. It was found that every combination of settings and initial estimates outputted very different solutions. The second purpose of the assignment was to use a two-layer neural network to perform a nonlinear binary classification on the Fisher's Iris data set based on sepal length and width by finding an optimal weight vector. It was also compared with the MATLAB built-in neural network. The results were evaluated based on their accuracies and confusion matrices.

1 Introduction

1.1 Nonlinear Least Squares - GPS Localization

The objective of the first part of this assignment was to formulate a Global Positioning System (GPS) localization as a nonlinear least squares (NLS) problem to estimate a receiver location based on the Cartesian coordinates of 6 satellites and their respective pseudoranges. The motivation for formulating GPS localization as an NLS problem is that GPS works by having a receiver estimate their location based on very accurate information sent from satellites. This estimation process is a minimization of errors of readings with nonlinear models which suits the kind of problem that NLS models.

This GPS localization has the following components. y_i is the pseudorange of the i^{th} satellite which is found by multiplying the time delay between the message being sent and received with the speed of light. $\vec{x}_i$ is the 3D location of the i^{th} satellite in Earth-centered Earth-focused Cartesian coordinates. Lastly, $g_i(\vec{w})$ is the actual distance between from the receiver to the i^{th} and is defined as $g_i(\vec{w}) := ||\vec{w} - \vec{x}_i||$. $g_i(\vec{w})$ can expanded to

$$g_i(\vec{w}) = (\vec{w}^T\vec{w} - 2\vec{x}_i^T\vec{w} + \vec{x}_i^T\vec{x}_i)^{\frac{1}{2}} \tag{1}$$

The gradient of $g_i(\vec{w})$ can be found by applying the chain rule to Equation (1)

$$\underline{\nabla} g_i(\vec{w}) = \frac{1}{2}(\vec{w}^T\vec{w} - 2\vec{x}_i^T\vec{w} + \vec{x}_i^T\vec{x}_i)^{\frac{-1}{2}} \tag{2}$$

$$= [\vec{w} - \vec{x}_i]^T/||\vec{w} - \vec{x}_i|| \tag{3}$$

Thus, the Jacobian matrix for m satellites can be elegantly written as

$$J_{\vec{g}}(\vec{w}) = \begin{bmatrix} [\vec{w} - \vec{x}_1]^T/||\vec{w} - \vec{x}_1|| \\ [\vec{w} - \vec{x}_2]^T/||\vec{w} - \vec{x}_2|| \\ \vdots \\ [\vec{w} - \vec{x}_m]^T/||\vec{w} - \vec{x}_m|| \end{bmatrix} \tag{4}$$

The residual errors of the distances are defined as

$$\vec{r}(\vec{w}) := \begin{bmatrix} g_1(\vec{w}) - y_1 \\ g_2(\vec{w}) - y_2 \\ \vdots \\ g_m(\vec{w}) - y_m \end{bmatrix} \tag{5}$$

This problem can then be solved using the Gauss-Newton or Levenberg–Marquardt algorithm by minimizing the residual error described in Equation (5) using the Jacobian matrix in Equation (4). The problem was attempted with different initial estimates and the scientific question was "How do different starting points affect the conversion behaviour of the algorithm?". Each trial was evaluated on whether it reached a local minimizer or not, and if it did, the number of iterations needed.

1.2 Linear Algebra for Neural Networks

The objective of the second part of the assignment was to train a two-layer neural network in the forward direction to perform a binary classification on the Fisher's Iris data. More specifically, to separate the "setosa" species from the others. This network will contain a hidden layer with 2 neurons and an output layer with 1 neuron. The motivation for this was that the classification to separate "setosa" from the two other species would not be possible with only 1 hyperplane meaning the classification is not linearly separable and requires more than 1 decision hyperplane. The data in this problem is a 150×2 matrix where the two variables are the width and length of the sepals. Each observation (row) also comes with a label $y \in \{0, 1\}$ for classification and is contained in vector $\vec{y}$.

A classification function is required to quantize the outputs of each layer which is how the network will assign a class to each observation. For this implementation, the classification function was chosen to be the Heaviside step function, which is defined as

$$H(v) := \begin{cases} 0 & \text{if } v < 0 \\ 1 & \text{if } v \geqslant 0 \end{cases} \tag{6}$$

A neural network will involve the solving extensions of linear equations such as $A\vec{w} = \vec{c}$ where the goal is to solve for $\vec{w}$ and the other components are given. The Kronecker vectorization theorem can be used to solve for the matrix extensions of simple linear equations $AW = C$ and can be used to write the linear responses for each layer. The Kronecker product, denoted with the operator symbol $\otimes$, for some matrix $A \in \mathbb{R}^{m \times n}$ and $C \in \mathbb{R}^{p \times q}$ is a block diagonal matrix $M \in \mathbb{R}^{mp \times nq}$ with partitions

$$m_{ij} := a_{ij} C \tag{7}$$

The vectorization of a matrix is the columns of the matrix "stacked" on top of one another. For some matrix $A \in \mathbb{R}^{m \times n}$, the vectorization is

$$\text{vec}(A) := \begin{bmatrix} \vec{a}_1 \\ \vec{a}_2 \\ \vdots \\ \vec{a}_n \end{bmatrix} \tag{8}$$

The Kronecker vectorization theorem [HJ91] states that for any given matrices A, M, C and some unknown matrix W, the matrix equation $AWM = C$ is equivalent to

$$[M^T \otimes A]\text{vec}(W) = \text{vec}(C) \tag{9}$$

The Hadamard product, denoted as $\odot$, for matrices $A \in \mathbb{R}^{m \times n}$ and $C \in \mathbb{R}^{m \times n}$ is a matrix $M \in \mathbb{R}^{m \times n}$ with entries

$$m_{ij} := a_{ij} c_{ij} \tag{10}$$

This is also known as the entry-wise product and this extends to a general class of Hadamard operators that act in this entry-wise fashion. The Hadamard product has a useful property that for compatible matrices $A \in \mathbb{R}^{m \times n}$, $C \in \mathbb{R}^{m \times n}$, $M \in \mathbb{R}^{p \times q}$, and $W \in \mathbb{R}^{p \times q}$, the Hadamard product of the Kronecker product is equal to the the Kronecker products of the Hadamard products

$$[A \otimes C] \odot [M \otimes W] = [A \odot M] \otimes [C \odot W] \tag{11}$$

The scientific question for this part of the assignment was "How accurately can this custom network classify the given data?". The results were evaluated based on the accuracy of the classification compared

to the actual labels. In addition to this, a confusion matrix was created for the classification to further analyze the classification. The classifications from this network were also compared with those of a built-in MATLAB network.

2 Methods

2.1 Levenberg-Marquardt Algorithm

The Levenberg-Marquardt algorithm (LMA) is a method used to solve nonlinear least squares problems. LMA is a combination of the Gauss-Newton algorithm (GMA) and the steepest descent method. It titters between the aforementioned methods based on a hyperparameter. The motivation for LMA was that GMA often failed to converge if the initial estimate was "too far" away from a local minimizer or if the Jacobian matrix at any point was rank-deficient. Kenneth Levenberg found these problems in 1944 [Lev44] and were solved in 1963 by Donald Marquardt [Mar63].

GMA is a scaled form of steepest descent and is based on the first-order approximation using the Taylor series of the actual distance between a satellite and the receiver from the GPS localization problem $g_i(\vec{w})$. The approximation at some point $\vec{w}_0$ is

$$g_i(\vec{w}) \approx g_i(\vec{w}_0) + \underline{\nabla} g_i(\vec{w}_0)[\vec{w} - \vec{w}_0] \tag{12}$$

The approximated residual $q_i(\vec{w})$ for each reading y_i is then defined as

$$q_i(\vec{w}) := g_i(\vec{w}_0) + \underline{\nabla} g_i(\vec{w}_0)[\vec{w} - \vec{w}_0] - y_i \tag{13}$$

To formulate this as an NLS problem, the approximated residual errors are gathered into a function with a vector output $\vec{q}(\vec{w}) = J_{\vec{g}}(\vec{w}_0)\vec{w} - [J_{\vec{g}}(\vec{w}_0)\vec{w}_0 - (r_i(\vec{w}_0))]$ where $r_i(\vec{w}_0) = q_i(\vec{w}_0) - y_i$ and organized into an objective function

$$f(\vec{w}) := ||\vec{q}(\vec{w})||^2 \tag{14}$$

Recall that the goal of this NLS problem is to find a minimizer $\vec{w}^*$ to the objective function $f(\vec{w})$. Consider when $\vec{q}(\vec{w}^*) = 0$, that is to say, the approximated residual errors have been perfectly minimized

$$||\vec{q}(\vec{w})||^2 = 0 \tag{15}$$

$$J_{\vec{g}}(\vec{w}_0)\vec{w}^* = [J_{\vec{g}}(\vec{w}_0)\vec{w}_0 - (r_i(\vec{w}_0))] \tag{16}$$

Equation (16) has the normal equation as its solution

$$[[J_{\vec{g}}(\vec{w}_0)]^T J_{\vec{g}}(\vec{w}_0)]\vec{w}^* = [J_{\vec{g}}\vec{w}_0]^T[J_{\vec{g}}(\vec{w}_0)\vec{w}_0 - (r_i(\vec{w}_0))] \tag{17}$$

$$\vec{w}^* = \vec{w}_0 - [[J_{\vec{g}}(\vec{w}_0)]^T[J_{\vec{g}}(\vec{w}_0)]]^{-1}[J_{\vec{g}}(\vec{w}_0)]^T\vec{r}(\vec{w}_0) \tag{18}$$

Substitute $\vec{w}_0$ with the estimate at the current estimate $\vec{w}_k$ and $\vec{w}^*$ with $\vec{w}_{k+1}$ then the step for each iteration of the GMA is

$$\vec{w}_{k+1} = \vec{w}_k - [[J_{\vec{g}}(\vec{w}_k)]^T[J_{\vec{g}}(\vec{w}_k)]]^{-1}[J_{\vec{g}}(\vec{w}_k)]^T\vec{r}(\vec{w}_k) \tag{19}$$

Observe that the matrix that scales the steepest descent portion of the iteration step is guaranteed to be symmetric and positive definite if the Jacobian matrix is full-rank. The solution proposed in the LMA is the addition of a non-negative scalar coefficient $\lambda \geqslant 0$ that scales an identity matrix which "damps" the effect of the GMA solution. Each iteration in the LMA is written as

$$\vec{w}_{k+1} = \vec{w}_k - [[J_{\vec{g}}(\vec{w}_k)]^T[J_{\vec{g}}(\vec{w}_k)] + \lambda I]^{-1}[J_{\vec{g}}(\vec{w}_k)]^T\vec{r}(\vec{w}_k) \tag{20}$$

When $\lambda = 0$, the LMA is no different than the GMA. As $\lambda \to \infty$, the LMA scaling matrix is overtaken by the λI component and it becomes equivalent to the steepest descent method.

In MATLAB, the GPS localization problem was formulated by first creating an anonymous function that finds the residual errors between each of the 6 satellites' Cartesian coordinates and their pseudo-range and returns it as a vector with 6 entries. This NLS problem was solved using the MATLAB function "lsqnonlin()" with its default settings. It was also tested with a change to its algorithm parameter that specified the function to use LMA. Other than this change, other parameters including λ were kept as the default options. The satellite locations were loaded as a 6×3 matrix which differs slightly from what was described in Section 1.1 since each coordinate here is a row object. The pseudoranges were loaded as a 6-entry vector. This function finds the minimum of the sum of squares of the inputted residual errors function to estimate a receiver location. It also requires an initial estimate which in this case was chosen to be the mean of each dimension of the satellites' Cartesian coordinates, the center of the earth [0; 0; 0], or the Cartesian coordinates of Kingston, Ontario. An estimated minimizer was found for each of these initial estimates and the results were evaluated by converting the Earth-centered Cartesian coordinates to a global coordinate system (GCS) form (i.e. latitude, longitude, and altitude) using the World Geodetic System 1984 (WGS84). The purpose of this was for ease of interpretation since the GCS coordinates can be searched on any popular map service to visualize the results better. This conversion was done using the "ecef2lla()" function from the Aerospace Toolbox in MATLAB. Lastly, the results from each starting point were compared with each other based on whether each attempt was able to reach a local minimizer or not, and if it did, the number of iterations needed.

2.2 Two-Layer Neural Network for Supervised Learning

A layered two-layer neural network was used to classify the 150 observations in Fisher's Iris data set based on the length and width of the sepals. In addition to these two variables, each observation is labelled as I. setosa, I. virginica, and/or I. versicolor. This neural network includes one hidden layer, an input layer for the incoming data, and an output layer. The hidden layer includes two neurons. The output layer has one neuron. Each neuron is also augmented with a scalar bias term of 1. Suppose a problem has m observations and n variables. The data can be gathered into a design matrix $X \in \mathbb{R}^{m\times(n+1)}$. Each row i of X will be an augmented matrix with a bias term $\underline{x}_i = [\vec{x}^T \ 1]$ where $\vec{x} \in \mathbb{R}^n$ is a data vector. The goal will be to find the augmented weight vector $\vec{w} \in \mathbb{R}^n$ that optimally classifies the data. The inner product is the product between the design matrix and the augmented weight vector $\vec{u} = X\vec{w}$ and has derivative $\frac{\partial \vec{u}}{\partial \vec{w}} = X$. The necessary data was stored in a global variable with attributes that contain the data which is a 150×3 augmented data matrix, the label vector, and the number of layers in the network. The initial weight vector was chosen to be

$$w_0 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ -1 \\ -1 \\ 3 \\ -1 \\ -1 \\ 7 \end{bmatrix} \tag{21}$$

First consider how a single neuron in this network could perform steepest descent to minimize its residual errors. The activation function for the neurons in this network was chosen to be the "sigmoid" function which guarantees that each output is bound in an open interval of $(0, 1)$. This, for a vector argument $\vec{u} \in \mathbb{R}^{n+1}$, is written as

$$\vec{\phi}(\vec{u}) = \begin{bmatrix} 1/(1+e^{-u_1}) \\ 1/(1+e^{-u_2}) \\ \vdots \\ 1/(1+e^{-u_{n+1}}) \end{bmatrix} \tag{22}$$

Consider the mapping for a scalar input to scalar output $\phi(u_i)$. Its derivative would be

$$\psi(u_i) = \ \phi(u_i)(1 - \phi(u_i)) \tag{23}$$

Since the i^{th} entry of $\vec{\phi}(\vec{w})$ depends only on the i^{th} entry of $\vec{u}$, the Jacobian of $\vec{\phi}(\vec{w})$ in respect to $\vec{u}$ is

$$J_{\vec{\phi}} = \text{diag}(\psi(u_i)) \tag{24}$$

Let $y_i \in \{0, 1\}$ be the label of observation i. These labels can be gathered into a vector $\vec{y}$. For the purpose of later formulating an objective function as a sum of squares, the residual error vector $\vec{r}$ needs to be found. For this implementation, the output of a neuron requires a classification function $q : \mathbb{R} \to \{0, 1\}$ to separate the outputs based on their activation values. Function q was chosen to be the Heaviside function which for any scalar value returns 1 if the input is greater than or equal to 0 and 0 otherwise. The residual error for the vector function $\vec{q}$ is

$$\vec{r}_{\vec{q}}(\vec{q}(\vec{\phi}(\vec{u}))) := \vec{y} - \vec{q}(\vec{\phi}(\vec{u})) \tag{25}$$

The Jacobian of the residual vector with respect to $\vec{q}$ is

$$J_{\vec{r}} = -I \tag{26}$$

With the residual error vector, the objective function f to be optimized is

$$f(\vec{r}) := \frac{1}{2}\vec{r}^T\vec{r} \tag{27}$$

The derivative of the objective function with respect to $\vec{r}$ is

$$\frac{\partial f}{\partial \vec{r}} = \vec{r}^T \tag{28}$$

The gradient (1-form) of the objective function at some weight vector $\vec{w}_0$ can be found by using the Chain rule. By multiplying the derivatives that were found previously, the gradient of f can be found

$$\underline{\nabla} f(\vec{w}_0) = -\vec{r}^T J_{\vec{\phi}} X \tag{29}$$

This gradient can be used in steepest descent to find a hyperplane for a single neuron. To extend this into multiple layers, first consider layer 1, the output layer. This network was used to perform a binary supervised classification so the output layer only requires one neuron. The equations discussed earlier can be rewritten with a leading subscript to denote what layer each symbol is describing. Thus the required components for the output layer can be rewritten as

$$\begin{aligned}
{}_1u({}_1\vec{w}) &= {}_1\underline{x}\,{}_1\vec{w} \\
\frac{\partial\,{}_1\phi}{\partial\,{}_1u} &= {}_1\psi({}_1\vec{w}) \\
\frac{\partial\,{}_1u}{\partial\,{}_1\vec{w}} &= {}_1x \\
\frac{\partial\,{}_1\phi}{\partial\,{}_1\vec{w}} &= {}_1\psi({}_1\vec{w})\,{}_1\underline{x}
\end{aligned}$$

For the hidden layer, these components will have to be derived for two neurons. The linear responses of the neurons can be gathered into a 1-form ${}_2u = [{}_2u_1 \;\; {}_2u_2]$ where the ending subscripts indexes each neuron in the layer. Likewise, the weights can be gathered into ${}_2W = [{}_2\vec{w}_1 \;\; {}_2\vec{w}_2]$. The linear response can then be written as ${}_2u({}_2W) = \underline{x}\,{}_2W$. Using the Kronecker vectorization, the previous equation can be rewritten to vectorize the linear response. Let ${}_2D = I \otimes \underline{x}$ and ${}_2\vec{u} = [{}_2\underline{u}]^T$, the previous equation is equivalent to

$${}_2\vec{u} = {}_2D\,{}_2\vec{w} \tag{30}$$

The derivative of the linear response vector with respect to the weights is $\frac{\partial\,{}_2\vec{u}}{\partial\,{}_2\vec{w}} = {}_2D$. Like in the case of a single neuron the activation values will be gathered into ${}_2\vec{\phi}$. The output of the hidden layer will be the activation values with an appended 1 for the bias value. So the data vector for layer 1 is

$${}_1\vec{x}({}_2\vec{u}) := \begin{bmatrix} {}_2\vec{\phi} \\ 1 \end{bmatrix} \tag{31}$$

The data vector of layer 1 can also be transposed such that ${}_1\underline{x}({}_2\vec{u}) = {}_1\vec{x}^T$. The Jacobian for the activation function of the hidden layer is

$$J_{{}_2\vec{\phi}} = \text{diag}({}_2\psi({}_2u_i)) \tag{32}$$

By applying the Chain rule and using the previously found derivatives, the derivative of the hidden layer's activation function with respect to the weight vector is

$$\frac{\partial\, {}_2\vec{\phi}}{\partial\, {}_2\vec{w}} = \left[J_{{}_2\vec{\phi}}\right] {}_2D \tag{33}$$

The derivative of the hidden layer's input data with respect to the weight vector only requires the addition of a row of 0s below since the derivative of the constant bias terms will be 0. This derivative is

$$\frac{\partial\, {}_2\vec{x}}{\partial\, {}_2\vec{w}} = \begin{bmatrix} \left[J_{{}_2\vec{\phi}}\right] {}_2D \\ \underline{0} \end{bmatrix} \tag{34}$$

What remains is the combination of the two layers. This starts with stacking the weights of both layers into one weight vector

$$\vec{w} := \begin{bmatrix} {}_1\vec{w} \\ {}_2\vec{w} \end{bmatrix} \tag{35}$$

The linear response of the network is ${}_1u(\vec{w})$ which can be expanded to

$$\begin{aligned} {}_1u(\vec{w}) &= {}_1\underline{x}({}_2\vec{w})\, {}_1\vec{w} && (36)\\ &= {}_1w^T \left[{}_1\underline{x}({}_2\vec{w})\right]^T && (37)\\ &= {}_1w^T\, {}_1\vec{x}({}_2\vec{w}) && (38) \end{aligned}$$

The derivative of the linear response of the network needs to be found with respect to both the output layer's and the hidden layer's weights. The derivative with respect to the weights of the output layer is

$$\frac{\partial\, {}_1u}{\partial\, {}_1\vec{w}} = {}_1\underline{x}({}_2\vec{w}) \tag{39}$$

The derivative with respect to the weights of the output layer is

$$\frac{\partial\, {}_1u}{\partial\, {}_2\vec{w}} = {}_1\vec{w}^T \begin{bmatrix} \left[J_{{}_2\vec{\phi}}\right] {}_2D \\ \underline{0} \end{bmatrix} \tag{40}$$

The combination of this results in the gradient of ${}_1u$

$${}_1\underline{\nabla}u = \left[\frac{\partial\, {}_1u}{\partial\, {}_1\vec{w}}, \frac{\partial\, {}_1u}{\partial\, {}_2\vec{w}}\right] \tag{41}$$

The output of the network is the activation of the output layer ${}_1\phi({}_1u)$ and its derivative with respect to ${}_1u$ is ${}_1\psi({}_1u)$. The residual error of the network is $r(\vec{w}) = y - q({}_1\phi({}_1u))$ where y is the label and q is the classification. The derivative of the residual error with respect to q is

$$\frac{\partial r}{\partial q} = -1 \tag{42}$$

The objective function was formulated as a sum of squares $f(\vec{w}) = \frac{1}{2}(r(\vec{w}))^2$. The objective function's derivative with respect to r is

$$\frac{\partial f}{\partial r} = r(\vec{w}) \tag{43}$$

To formulate these components for multiple data vectors, the residual errors need to be gathered into $\vec{r}$. The gradient for the objective function for one observation with respect to the weight vector can be found using the Chain rule and the components previously calculated

$$\underline{\nabla} f(\vec{w}) = -r(\vec{w}) \begin{bmatrix} {}_1\underline{x}({}_2\vec{w}) & {}_1\vec{w}^T \begin{bmatrix} \left[J_{{}_2\vec{\phi}}\right] {}_2D \\ \underline{0} \end{bmatrix} \end{bmatrix} \tag{44}$$

In the code the gradient of the objective was found for all observations in one computation, not one by one. However, the component $\frac{\partial_{\,1}u}{\partial_{\,2}\vec{w}}$ was calculated for each observation and then stacked on top of one another. The data inputs of the first layer are the activation values from the hidden layer since for this implementation, the activation values from the hidden layer do not go through the classification function. So the data inputs for the output layer can be stacked into a matrix ${}_1\Phi({}_2\vec{w})$. Lastly, to make sure the residual error component's dimensions match that of the others it will be multiplied with $-I$ which in the code is done by diagonalizing the residual error vector. This matrix is written as

$$G_f = -r(\vec{w})[I] \begin{bmatrix} & \begin{bmatrix} {}_1\vec{w}^T \begin{bmatrix} [J_{\,{}_2\vec{\phi}_1}]\,{}_2D_1 \\ \underline{0} \end{bmatrix} \\ {}_1\vec{w}^T \begin{bmatrix} [J_{\,{}_2\vec{\phi}_2}]\,{}_2D_2 \\ \underline{0} \end{bmatrix} \\ \vdots \\ {}_1\vec{w}^T \begin{bmatrix} [J_{\,{}_2\vec{\phi}_m}]\,{}_2D_m \\ \underline{0} \end{bmatrix} \end{bmatrix} \\ {}_1\Phi({}_2\vec{w}) & \end{bmatrix} \tag{45}$$

The net gradient for this network will be the sum of G_f along each column which will result in a 1×9 row matrix which can then be passed along to use for steepest descent. The optimization method chosen for this network was a fixed step size steepest descent. Each iteration of this method, where s is the given step size and $\vec{d_k} = -[\text{sum}(G_f)]^T$ is the step direction, for each iteration k is

$$\vec{t}_{k+1} = \vec{t}_k + s\vec{d_k} \tag{46}$$

In this implementation, the step size was chosen to be 0.01, the estimate was set to converge if the norm of the gradient is less than or equal to 10^{-3}, and the maximum number of iterations was set to be 5000. The results from this network were evaluated based on the confusion matrix when evaluated against the actual labels and they were compared with that of the built-in MATLAB neural network.

3 Results

Table 1: Estimates from MATLAB function "lsqnonlin()" for each of the three initial estimates, using the default and LMA settings. All outputs are in Cartesian coordinates, (x, y, z) in metres, with the center of the Earth as the origin and reported to one decimal place.

Settings \ Initial Estimate	**Mean of Satellite Coordinates**	**Origin (Center of the Earth)**	**Kingston, ON**
Default Settings	(14131448.6, 13301800.9, 17048043.5)	(7804302.5, 3681377.5, 20943982.7)	(18713879.6, 742092.5, 13778468.7)
LMA setting	(9231367.4, 4899043.4, 24461707.9)	(12360953.9, 2850848.3, 23382466.6)	(14516076.2, 1862032.7, 22210192.9)

Table 2: Confusion matrix for binary classification done by custom two-layer neural network

	Predicted Other	**Predicted Setosa**
Other	97	3
Setosa	3	47

Table 3: Confusion matrix for binary classification done by a built-in MATLAB neural network (note that the results for this network vary slightly from run to run)

	Predicted Other	**Predicted Setosa**
Other	98	2
Setosa	3	47

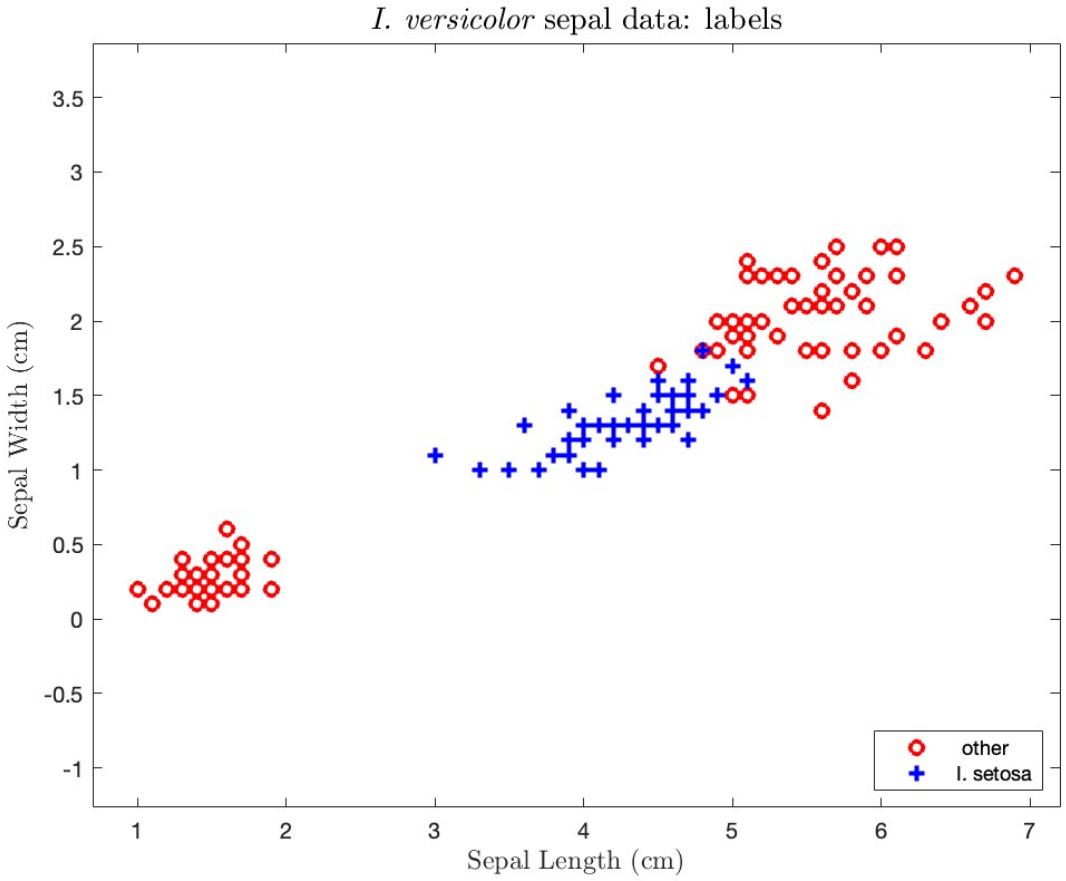

Figure 1: Scatter plot displaying the Fisher's Iris data with the original labels. Sepal length in cm on the X axis and sepal width in cm on the Y axis.

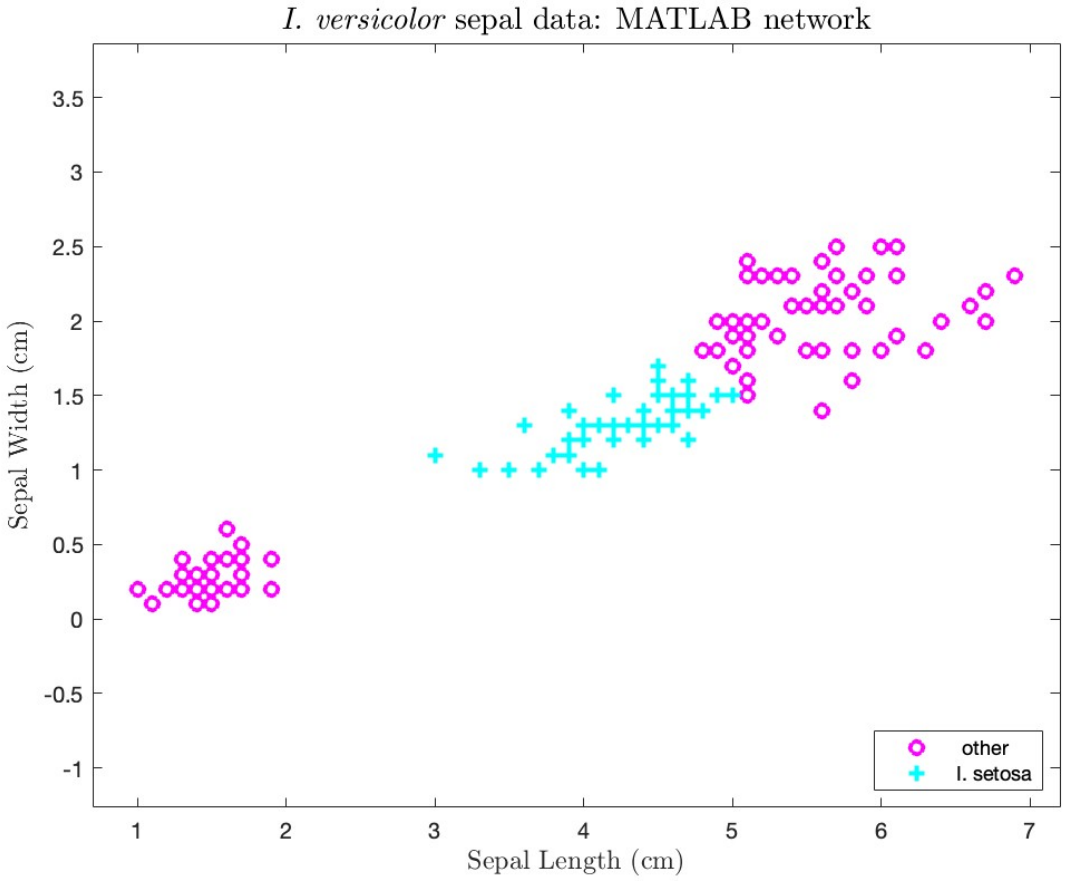

Figure 2: Scatter plot displaying the MATLAB built-in neural network's binary classification. Sepal length in cm on the X axis and sepal width in cm on the Y axis. (note that the results for this network vary slightly from run to run)

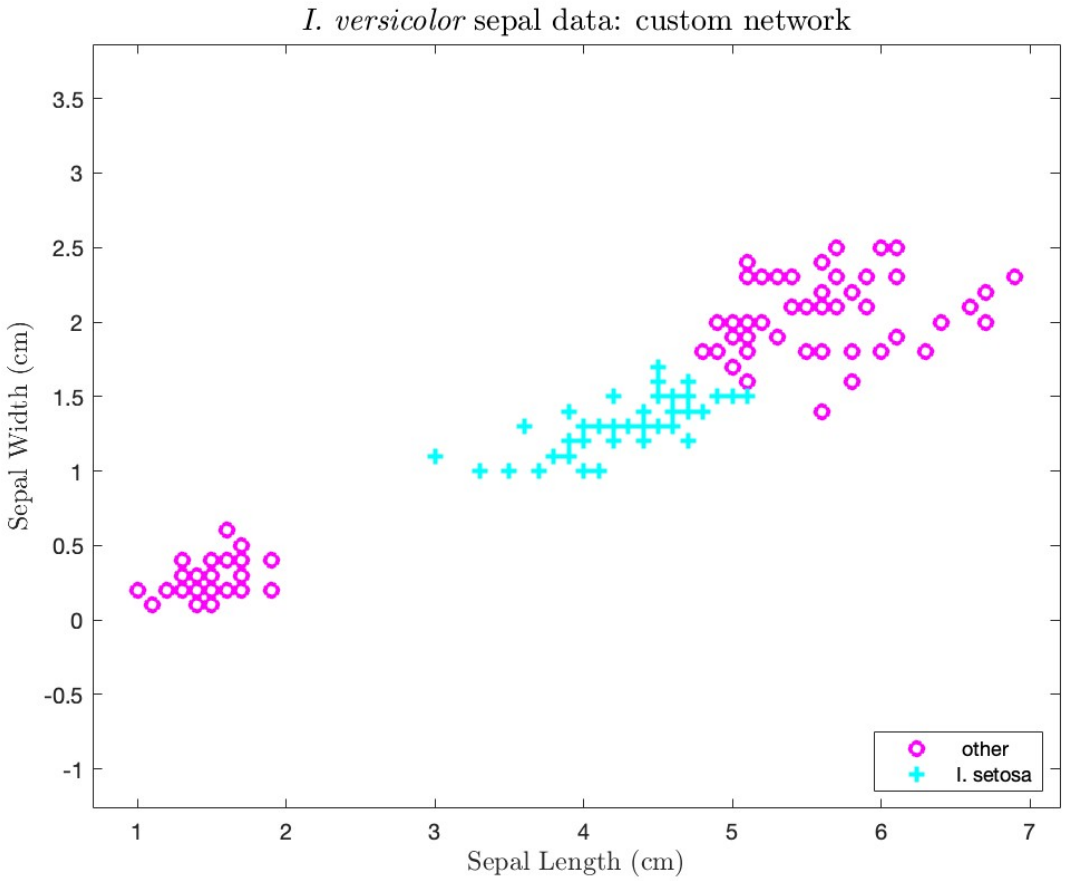

Figure 3: Scatter plot displaying the custom neural network's binary classification. Sepal length in cm on the X axis and sepal width in cm on the Y axis.

4 Discussion

4.1 GPS Localization

Using the default settings for function "lsqnonlin()", running the algorithm with the first initial estimate which is means of the satellites' coordinates managed to arrive at a local minimizer of (14131448.6, 13301800.9, 17048043.5) after 1 iteration (in GCS coordinates this is 41.3, 43.3, 19462776.7). Using the second initial estimate which is the origin, the algorithm stopped prematurely after 75 iterations at (7804302.5, 3681377.5, 20943982.7) since it had reached the maximum number of function evaluations which by default is 100 times the number of variables so 300 in this case (in GCS coordinates this is 67.6, 25.3, 16292078.4). When using the Cartesian coordinates of Kingston, Ontario as the initial estimate, the algorithm stopped prematurely at (18713879.6, 742092.5, 13778468.7) after 75 iterations (this in GCS coordinates is 36.4, 2.3, 16880313.0).

Changing "algorithm" setting for function "lsqnonlin()" to specify the use of LMA, running the algorithm with the first initial estimate managed to arrive at a local minimizer of (9231367.4, 4899043.4, 24461707.9) after 8 iterations (in GCS coordinates this is 66.9, 28.0, 20240584.4). Using the second initial estimate, the algorithm arrived at a local minimizer of (12360953.9, 2850848.3, 23382466.6) after 11 iterations (in GCS coordinates this is 61.6, 13.0, 20240269.1). When using the third initial estimate, the algorithm arrived at a local minimizer of (14516076.2, 1862032.7, 22210192.9) after 7 iterations (in GCS coordinates this is 56.7, 7.3, 20235200.0).

The GCS coordinates of all the results show varying estimates for the receiver's location. Intuitively, the minimizer found using the first initial estimate with the default settings seems like the most correct one as one would expect the receiver to be in the "center" of all the satellites. This attempt also required the least number of iterations which points to the first initial estimate being the most appropriate one. The GCS coordinates of this minimizer indicate that the receiver is 19462776.7m above a small municipality in the country of Georgia, which is well above the atmosphere of the Earth. This should be expected since this is an unconstrained optimization problem so there is no restriction that states the receiver must be on Earth. Of course, it is also perfectly plausible that the receiver is actually in low to medium Earth orbit. The GCS coordinates of the minimizer found using the first initial estimate with LMA indicate the receiver is 20240584.4m above a remote location in northern Finland. This demonstrates the differences between the algorithms for this problem since the same initial estimate produced very distant minimizers. Using LMA with the second and third initial estimates also produced GCS coordinates in northern Europe all with similar altitude values. So there is also a case to be made for the minimizer found using LMA starting at the first initial estimate since even using initial estimates that are very far from this region gave a coordinate above northern Europe. Another approach is to look strictly at the objective function evaluation at the estimate. For "lsqnonlin()" with default settings, starting at the first initial estimate, the objective function value was 1.5379e+15. Specifying the use of "lsqnonlin()" resulted in objective function values of 5.0224e+12 for all initial estimates. These values also strengthen the case for the LMA setting.

It was observed that results for changes in the initial estimate had a significant effect on the output of the two settings. For the implementation with default settings, it was found that the algorithm did not converge for the second and third initial estimates. Considering that using the first initial estimate arrived at a local minimizer in just 1 iteration with a step norm (magnitude) of 10, which is a relatively small step norm, it can be inferred that the second and third initial estimates were too far away from a local minimizer. The documentation from MathWorks [Mat] states that the default algorithm used in "lsqnonlin()" is the trust-region-reflective algorithm (TRRA). The detailed workings of TRRA are outside the scope of this assignment but what is important to understand is that TRRA will only look for solutions within the neighbouring region of the initial estimate so even if both algorithms were given the same initial estimate, due to the nature of the algorithms, their convergence behaviour could be drastically different or they could both converge at distinct local minimizes. The default number of maximum function evaluations terminated TRRA prematurely and when this limit was raised to 1000, TRRA reached a local minimizer of (10183097.4, 4206606.7, 24227489.2) for the second initial estimate after 146 iterations and for the third initial estimate

reached a local minimizer of (21966888.6, 1722669.2, 14905746.9) after 117 iterations. These coordinates point to regions above Sweden and Algeria respectively which significantly differed from the location found starting at the first initial estimate. This means that there is a reachable local minimum for every initial estimate but the local minimizers are still quite different from each other. From these observations, it can be said that for this specific GPS localization problem, choosing an initial estimate that is far away from an actual local minimizer significantly reduced the performance of "lsqnonlin()" with its default algorithm.

Specifying the use of LMA for "lsqnonlin()" resulted in successful convergence for all initial estimates in relatively few iterations. The difference in nature between the two algorithms has led the minimizers starting at the same initial estimate to be very different. What is also interesting is that for the worse initial estimates, the origin and Kingston, LMA converged much faster than TRRA which points to an advantage of speed for this specific problem. A reason for this may be that LMA is not bound to the neighbouring region so using its combination of the Gauss-Newton and steepest descent method, it is able to take much larger steps in earlier iterations compared to TRRA. There is numerical evidence for this as it was observed that earlier iterations of TRRA had small step norms. For example, the step norm at iteration 7 for TRRA was 640 whereas the attempts by the LMA were in the range of approximately 10^7 to 10^8.

For the given GPS localization problem, which was formulated as an NLS problem, it was found that changes to the initial estimate had a large impact on the optimization done by the MATLAB function "lsqnonlin()" for both the default TRRA and for LMA. Provided a larger maximum number of function evaluations, TRRA was able to converge for all initial estimates although it was far slower for the second and third initial estimates and the coordinates differed more compared to LMA. LMA found very different minimizers than TRRA but was faster for the second and third initial estimates despite them being much farther away from their respective minimizers. It was also found that the minimizers found from LMA were closer to each other.

4.2 Two-Layer Neural Network for Binary Classification

The custom two-layered neural network did not converge for the chosen hyperparameters. After 5000 iterations of fixed step size steepest descent, it arrived at a weight vector of

$$\vec{w}^* = \begin{bmatrix} 2.1443 \\ -2.6836 \\ -0.0500 \\ -1.1517 \\ -2.8297 \\ 6.4619 \\ -2.6927 \\ -1.6956 \\ 6.9871 \end{bmatrix} \tag{47}$$

Although it did not converge, it was able to classify the observations as "setosa" (negative class) or not (positive class) with 96% accuracy. The confusion matrix showed that there were 97 true positives, 47 true negatives, 3 false negatives, and 3 false positives. Where the error occurs to the right of Figure 1 where observations with different labels exist very closely together, there are even two observations that seem to overlap each other. One would expect one of the hidden neurons to produce a hyperplane separating the data in the bottom left from the data to the right, but the other neuron's hyperplane would be trickier since the region to the right is more crowded with no clear binary classification. Slight variations in the decision hyperplane drawn in this region would result in varying classifications. It is expected that both the built-in neural network and the custom one would make mistakes in this region. It is difficult to evaluate the differences in the results between the builtin network and the custom one since the classification from the builtin networks seems to vary slightly from run to run. When there are differences between the built-in and the custom networks, it is in this region where observations of different labels are grouped closely.

Further exploration was done by varying the hyperparameters of the neural network. Since steepest descent did not converge, the maximum number of iterations was raised to 30000. It was observed that this change produced a different weight vector but the exact same confusion matrix. It was also noticed the weight vector after 5000 iterations was not significantly different from the one after 30000 considering the large increase in iterations. The step size was increased to 0.02 to see if it could improve the results. This change made no difference after 5000 iterations but after 30000 iterations, there was one less false negative in the confusion matrix, raising the accuracy of the classification to $96.\overline{6}\%$. Depending on the use case, it may be worth it to extract these extra improvements but if efficiency is of higher priority like in the case of a large language model where there is a massive amount of data involved, it is unlikely that these small changes in accuracy would be worth the additional computational costs. The same could not be said for a use case like crucial medical diagnostics where accuracy is of the utmost importance. The use case will also determine what kind of improvements should be made to the network. To relate back to the confusion matrix, if one were to make changes to say decrease the false positive or negative rate, the nature of the problem should be seriously considered. For example, if a neural network was built to detect spam emails, it would be hugely detrimental to accidentally increase the false positive rate (marking a legitimate email as spam).

It was conjectured that since fixed step size steepest descent did not converge, other optimization algorithms would be worth exploring. The optimization was attempted with Armijo backtracking. The initial step size was chosen to be 0.5, the convergence criteria remained the same, and the maximum number of iterations was set to 100. It was observed that for this neural network, using Armijo backtracking was essentially infeasible since it seems to backtrack the step size indefinitely and thus could not even complete a single iteration. Perhaps tuning the hyperparameters of the backtracking steepest descent method could produce a result of similar quality to that of the fixed step size method, but it seems that the backtracking process takes too heavy of a toll on the efficiency of the optimization step. This demonstrates that even for a simple neural network like the one implemented and a relatively small data set like Fisher's Iris, a simple fixed step size method could be more effective in practice than a more complex optimization method. Determining an effective method in machine learning is often about empirical success, so although a generalized statement cannot be made about the effectiveness of fixed step size methods, it has in this problem, produced satisfactory results that Armijo backtracking could not.

For the given binary classification problem on the Fisher's Iris data set, it was found that the custom network was able to classify "setosa" from the other species with 96% accuracy, although the fixed step size steepest descent was not able to converge after 5000 iterations. The custom network had 3 false positives and 3 false negatives The built-in MATLAB network produced similar results although they varied from run to run. It was found that both networks made errors near the region of the data where data with different labels were very closely grouped. The maximum number of iterations for fixed step size was increased but the accuracy only rose slightly; it was stated that depending on the use case, the extra computational cost may be worthwhile. Armijo backtracking was also implemented but it was determined that it was not a viable method for this problem.

References

[Lev44] Kenneth Levenberg. "A METHOD FOR THE SOLUTION OF CERTAIN NON-LINEAR PROBLEMS IN LEAST SQUARES". In: *Quarterly of Applied Mathematics* 2.2 (1944), pp. 164–168. ISSN: 0033569X, 15524485. URL: http://www.jstor.org/stable/43633451 (visited on 10/31/2023).

[Mar63] Donald W. Marquardt. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters". In: *Journal of the Society for Industrial and Applied Mathematics* 11.2 (1963), pp. 431–441. ISSN: 03684245. URL: http://www.jstor.org/stable/2098941 (visited on 10/31/2023).

[HJ91] Roger A. Horn and Charles R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, 1991. DOI: 10.1017/CBO9780511840371.

[Mat] MathWorks. *Solve nonlinear least-squares (nonlinear data-fitting) problems - MATLAB lsqnonlin.* URL: https://www.mathworks.com/help/optim/ug/lsqnonlin.html?s_tid=doc_ta#buul744-1.