Unconstrained Optimization Using Newton's Method and of Artificial Neural Networks

Part A) Unconstrained Optimization with Newton's Method

Abstract

Purpose: In this section of the assignment the goal is to gather results of how Newton's Method performs when looking for local minimizers on vector argument functions. To gain a basis in which results can be reliably compared, two other methods, fixed step size line search and line search with backtracking, were implemented. As well we use the same step size and starting point for each method and function.

Methods: As both fixed step size and backtracking line search have been extensively covered in lecture and were analyzed in the previous assignment, they will be more briefly covered. The focus is on Newton's method. This optimization method was implemented using MATLAB with equations derived and studied from the course material.

Results: The results of each optimization method are summarized into one table and are supported visually by 3-D plots of the functions with the estimated minimizers. It is clear that Newton's Method does provide a different way of finding local minimizers compared as the results do not match the other two methods.

Conclusions: From the values obtained in results and referencing the corresponding plots it can be noted that Newton's Method can work very well, but in some cases does not. As with most optimization methods this can be understood that there are certain conditions which affect the performance of Newton's Method, something that will be further explained in the relevant section.

Introduction

The goal of these experiments is to compare a newly introduced optimization method to two that have been repeatedly seen throughout this course. This new method is Newton's Method or more specifically Damped Newton's Method. The two methods for providing a comparable basis are fixed step size line search and backtracking line search.

Both variations of line search provide a reliable way of searching for local minimizers given appropriate conditions, that is a convex problem space. Computationally, issues can arise with these two methods when the partial derivatives composing the gradient have disproportionate ratios. An example of this was given in class, lecture 8, leading to the inspiration of studying Newton's Method.

Newton's Method provides a way of scaling the gradient descent, or steepest descent direction in a manner that is considerate of the difference in variable scales of the objective function. This scaling matrix could be selected by hand via inspection of the variables $w_1, \dots, w_n$ but this would be tedious. Instead we consider positive definite matrices, to preserve the convexity of our objective function. For Newton's Method this scaling matrix is the hessian of our objective function.

To view this in action a script was written in MATLAB that implements Damped Newton's Method. Damped implies that a step size is being selected through backtracking and Armijo's Condition.

Methods

As mentioned, both line search methods were implemented in Assignment 1 for this course, and thus a detailed explanation of their workings will be bypassed. Their implementation is seen in the attached MATLAB script, a2q1_20051918.

To ensure reproducibility the initial point and backtracking values should be noted as well as the step sizes for fixed step size line search and backtracking line search.

$$\vec{w}_0 = [-1.2, 1], \alpha = 0.5, \beta = 0.5$$

$$sfixed = 0.001, sline = 0.1$$

As well the three objective functions that will be used for testing these methods are…

$$f_1(\vec{w}) = \big((\mathrm{w}_1 + 1.21) - 2(\mathrm{w}_2 - 1)\big)4 + 64(\mathrm{w}_1 + 1.21)(\mathrm{w}_2 - 1) \quad (1)$$

$$f_2(\vec{w}) = 2{\mathrm{w}_2}^3 - 6{\mathrm{w}_2}^2 + 3{\mathrm{w}_1}^2\mathrm{w}_2 \quad (2)$$

$$f_3(\vec{w}) = 100(w_2 - {w_1}^2)^2 + (1 - w_1)^2 \quad (3)$$

Each optimization method will return the following three pieces of information for the three selected objective functions; the local minimum, the local minimizer and number of iterations.

In the written code both line search methods are called first returning their results to the console.

The implementation of Damped Newton's Method is found in the MATLAB function newtondamped. It takes five mandatory and two optional parameters. These are a function pertaining to the objective function and its gradient, the hessian of the objective function, initial point, the two backtracking values α and β, and optionally the maximum number of iterations and the gradient norm limit. For simplicity of focusing on the workings of this optimization method the objective function's gradient and hessian were both given.

The beginning of the function newtondamped was given by Dr. Ellis, these conditionals pertain to setting the optional parameters if provided.

Proceeding this section is where the many initial values are set prior to iteration. These initial values are set with the parameters that were passed and for the functions they are evaluated at the initial point $\vec{w}_0$.

In lecture it is defined that the descent direction for Newton's Method is the following…

$$\vec{d} = -\frac{\nabla^2 f(\vec{w}_k)}{\nabla f(\vec{w}_k)} \qquad (4)$$

Using nested while loops in MATLAB, with the outer loop checking for the norm gradient and iteration limits while the inner performs backtracking utilizing Armijo's Condition which is…

$$f(\vec{w}_k + s\vec{d}) > f(\vec{w}_k) + \alpha s|\nabla f(\vec{w}_k)|\vec{d} \qquad (5)$$

Once a satisfactory step size is chosen the algorithm moves our current point that distance in the appropriate direction given by Eq. 1.

$$\vec{w}_k = \vec{w}_k + s\vec{d}.$$

When either of the limit conditions is satisfied the while loop is broken out of and the relevant values returned, most importantly the local minimizer.

Results

Table 1: Displays the return values of each optimization method for the three given objective functions. The returned values displayed are the number of iterations and the local minimizers.

Method	**f_1**		**f_2**		**f_3**	
	Iter.	$\vec{w}^*$	Iter.	$\vec{w}^*$	Iter.	$\vec{w}^*$
Fixed	482	(-0.21, 0.50)	1486	(0.00, 2.00)	32076	(1.00, 1.00)
Backtracking	50000	(-0.21, 0.51)	26	(0.00, 2.00)	50000	(-1.02, 1.08)
Damped Newton's	200	(-1.21, 1.00)	200	(-1.20, 1.00)	18	(1.00, 1.00)

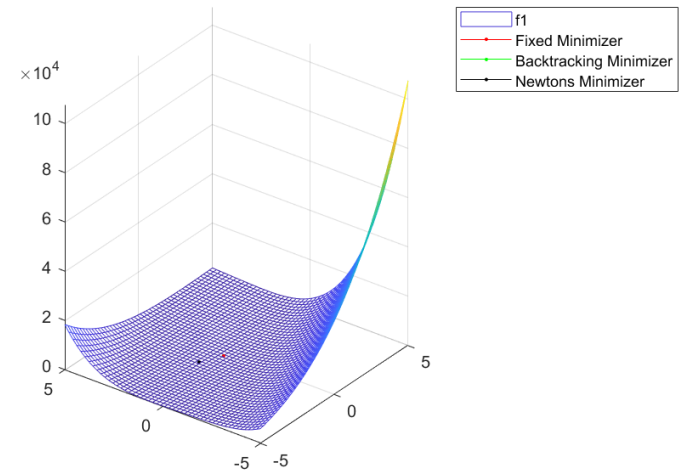

Figure 1: Shows the objective function $f_1(\vec{w})$ along with the minimizers found by the optimization methods. Note that the minimizer found by fixed line search and Backtracking line search near directly overlap, this is seen in Table 1.

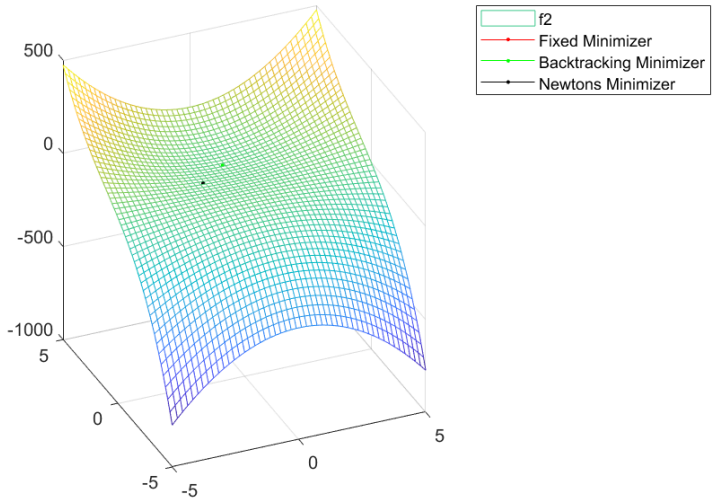

Table 2: Shows the objective function $f_2(\vec{w})$ along with the minimizers found by the optimization methods. Note that the minimizer found by fixed line search and Backtracking line search directly overlap, this is seen in Table 1.

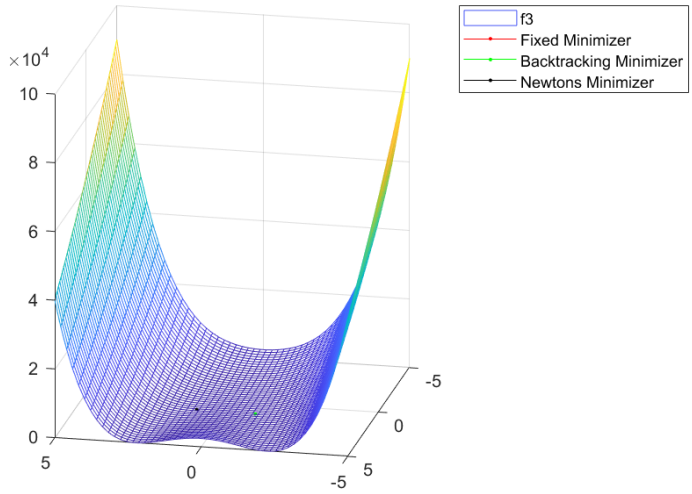

Table 3: Shows the objective function $f_3(\vec{w})$ along with the minimizers found by the optimization methods. Note that the fixed line search minimizer and the Newton's Method minimizer overlap as they are in the same place. This is seen in Table 1.

Discussion

From *table 1* and *figures 1-3* we can see that Newton's Method in comparison to other optimization methods we have seen in course work is indeed a viable choice. Visually we see the minimizers generated by the MATLAB function newtondamped occur in what appear to local minimizers of the function except for $f_2(\vec{w})$ where it appears to have found a local maximum due to a change of concavity.

Another result that is noteworthy comes from *table 1* and the number of iterations, specifically between Newton's Method and the line search methods. For $f_1(\vec{w})$ and $f_3(\vec{w})$ Newton's Method requires noticeably fewer iterations compared to backtracking line search which could lead us to believe it is a more efficient algorithm, except for $f_2(\vec{w})$ it requires significantly more than backtracking but still less than fixed line search. To understand why this is the case we should inspect the objective functions, Eq's 1,2 & 3, and their gradients, which are found in the given a2q1funs file, more closely. Let us choose to look at $f_1(\vec{w})$, its gradient is given as…

$$\underline{\nabla} f_1(\vec{w}) = [4((w_1 + 1.21) - 2(w_2 - 1))^3 + 64(w_2 - 1), -8(-2w_2 + w_1 + 3.21)^3 + 64(w_1 + 1.21)]$$

From this gradient it is clear that when expanded the two components of the gradient that w_1 and / or w_2 have disproportionate factors. This is why Newton's Method converges exceptionally well in this case as scaling by the hessian accounts for this.

Whereas if we look at $f_2(\vec{w})$ and its gradient…

$$\underline{\nabla} f_2(\vec{w}) = [6w_1w_2, 6{w_2}^2 - 12w_2 + 3{w_1}^2]$$

The variables w_1 and w_2 and their factors have much more parity thus Newton's Method tends to act as a median between the two methods. Albeit with a notably higher convergence rate then in $f_1(\vec{w})$ and $f_3(\vec{w})$.

This allows us to come to an important conclusion about Newton's Method for optimizing convex objective functions, that is for objective functions with multiple variables $w_1, \dots, w_n$ with disproportionally scaled values in components of the gradient $[\frac{df(\vec{w})}{dw_1}, \dots, \frac{df(\vec{w})}{dw_n}]$ the more efficiently Newton's Method will perform comparatively to previously seen non-scaled descent methods.

Part B) Training the weights of a Two-Layer Neural Network Using Optimization

Abstract

Purpose: The second portion of the assignment pertains to Neural Networks, specifically one implemented with two-layers. One can self-check that a single pass through of this system type does not provide reliable results. Using the provided data, the weights associated with each artificial neuron in the network can be trained by optimization methods studied in class. The provided data models a XOR dataset.

Methods: In MATLAB, a neural network was implemented with initial data and set up provided by Dr. Ellis. By modifying the algorithm in the course notes to fit a two-layer neural network where the hidden layer has two artificial neurons with a sigmoid transfer function and the output layer a single artificial neuron with linear transfer function, the code demonstrates how the gradient of the network objective function is computed and used to optimize the network.

Results: The output of our script are the trained weights for the neural network. These weights can then be partitioned to show the classification of the example XOR dataset visually. This figure is in the accompanying section of the report. To help convince that this is indeed an optimization problem plots of the weight vectors are shown, including how they change iteratively.

Conclusions: By comparing the initial weights of the system and how after optimization these weights can be better used to classify the XOR data. This implies that the derived method of optimizing a network from course notes can equivalently be seen as training a network. To see how this concept can be further improved, a varying learning rate is considered similar to backtracking in line searching.

Introduction

The goal of part two is to learn how to implement a simple linear / sigmoid neural network and compute the gradient of network objective function. To accomplish this, a script will be written in MATLAB that is based off a skeleton file given by Dr. Ellis. A neural network is an acyclic directed graph where the artificial neurons are simply computational nodes that take some input and produce a corresponding output. This allows it to be applied to estimation or classification problems.

The data that is loaded in the MATLAB code is 2-D data for the XOR relation and its corresponding labels. The aim is to optimize or train our neural network on the provided data to see how well it can classify the data.

An inspiration for studying neural networks comes directly from us humans. With one of if not the most developed cognitive system on the planet the ability to even mimic a portion of our brain's capability would be tremendous for the progress in computation.

To truly view if neural networks and artificial neurons can be treated as an optimization problem the MATLAB script will be further broken down in the following section describing the individual parts.

Methods

Before delving into the core parts of the neural network, it is important to briefly describe the main function. To begin, a large portion of the main function involves set up of the data, weights and calls necessary for the script to work smoothly. For a better understanding of this it is recommended to look at the MATLAB script.

The part to study begins when the call to function annfun is made. This function only takes one parameter wvec, the initial weights used for the neural network.

$$_{12}\vec{w}_0 = [1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1]^T$$

The first lines in annfun pertain to initializing instances of global variables. These global variables are stored in ANNDATA and include the original XOR data, number of layers and labels for the XOR data.

Following further set up, the first computation in the neural network is computing the hidden responses from the hidden layer where the transfer function is sigmoidal.

$$_1\phi(\ _1\hat{x}^T\ _1\hat{w}_i), i = 1, \dots, l \qquad (1)$$

These responses are given as long row vectors.

$$ {}_2\hat{x}({}_1\widehat{w}) = \begin{bmatrix} {}_1\phi({}_1\hat{x}^T\ {}_1\widehat{w}_1) \\ {}_1\phi({}_1\hat{x}^T\ {}_1\widehat{w}_2) \\ 1 \end{bmatrix} \qquad (2) $$

Where ${}_1\phi(u)$ is a sigmoid transfer function described in the below equation.

$$ {}_1\phi(u) = \frac{1}{1+e^{-u}} \qquad (3) $$

Proceeding Eq. 2 finding the result of the output transfer function is simple as the artificial neuron implements a linear transfer function.

$$ {}_2u({}_{12}\widehat{w}) = [\,{}_2\hat{x}({}_1\widehat{w})]^T\ {}_2\widehat{w} \qquad (4) $$

The predictive classification of the elements in ${}_2u({}_{12}\widehat{w})$, is Heaviside step function of the transfer function.

$$ {}_2\phi\big({}_2u({}_{12}\widehat{w})\big) = \begin{cases} 0\ if\ u < 0 \\ 1\ if\ u \geq 0 \end{cases} \qquad (5) $$

The residual vector, which is necessary for computing the gradient is given as…

$$ r({}_{12}\widehat{w}) = \vec{y} - {}_2\phi\big({}_2u({}_{12}\widehat{w})\big) \qquad (6) $$

Where $\vec{y}$ is the vector of labels loaded in at the beginning of the script. Finally, the objective of the network can be computed being…

$$ f(\vec{r}) = \frac{1}{2}\vec{r}^T\vec{r} \qquad (7) $$

Finding the differential responses for the hidden layer requires computing.

$$ {}_1\psi({}_1u_l) = \phi(u_l)\cdot(1-\phi(u_l)) \qquad (8) $$

Modelling the differential responses of the hidden layer in one matrix are best done using the ψ matrix defined in the course notes. This is a block diagonal matrix as seen below.

$$ \Psi({}_1\vec{u}) = \begin{bmatrix} \psi({}_1u_1) & 0 & 0 \\ 0 & \psi({}_1u_2) & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad (9) $$

In the MATLAB script this matrix setup is seen in the psimat variable. The following matrix is also needed and is obtained via a Kronecker Product.

$$ xhatkron = [\begin{bmatrix} I \\ 0 \end{bmatrix} \otimes \hat{x}^T] \qquad (10) $$

Where the identity matrix is of dimensions $[l, l]$ and l is the number of layers. Using Eq's. 8 & 9, the Jacobian of the hidden layers is computed, scaled by the weight associated with layer two. In the lecture notes for class 11, a method for computing this is given. The variable set to this computation is the one in the MATLAB script.

$$wJmat = {}_2\hat{w} \cdot \Psi({}_1\vec{u}) \cdot xhatkron \quad (11)$$

In the MATLAB a diagonal matrix is created from the product of ${}_2\hat{w} \cdot \Psi({}_1\vec{u})$ in Eq. 10 to preserve the separation of calculations pertaining to individual neurons.

With Eq. 10 it is also possible to now form the gradient of each data vector, that is forming a matrix which is the result of differentiating the residual error.

$$\underline{\nabla} f({}_{12}\hat{w}) = -r({}_{12}\hat{w}) \cdot I \cdot [\,{}_2\hat{x}^T \quad wJmat] \quad (12)$$

Finally summing column-wise, results in a $[1, number\ of\ initial\ weights]$ vector which is the gradient of the network that is passed back to the fixed steepest descent optimization method.

Optionally, it was asked that a varying step size descent method could be investigated. This was implemented in the function steepvary found in the MATLAB script. Very few changes are made between that function and steepfixed. The main change to note is that we implement Armijo's condition by adding two new parameters, alpha and beta, these are common parameters when performing backtracking for on the step size.

Results

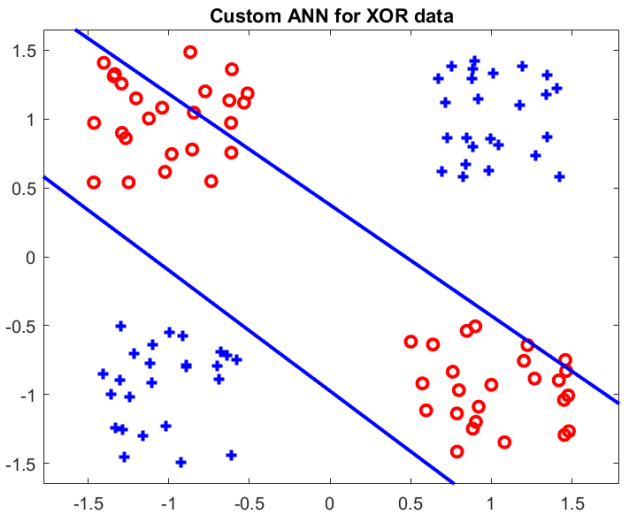

Figure 1: Plotting the weight vectors associated with the hidden layer artificial neurons on the XOR data as classification planes. These weight vectors are optimized using fixed steepest descent.

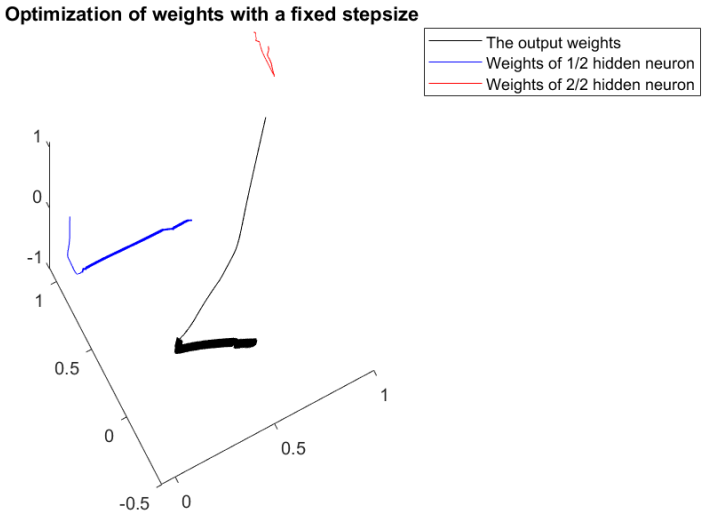

Figure 2: Plotting the change in weights as they are optimized over iterations. Done with a fixed step size steepest descent method.

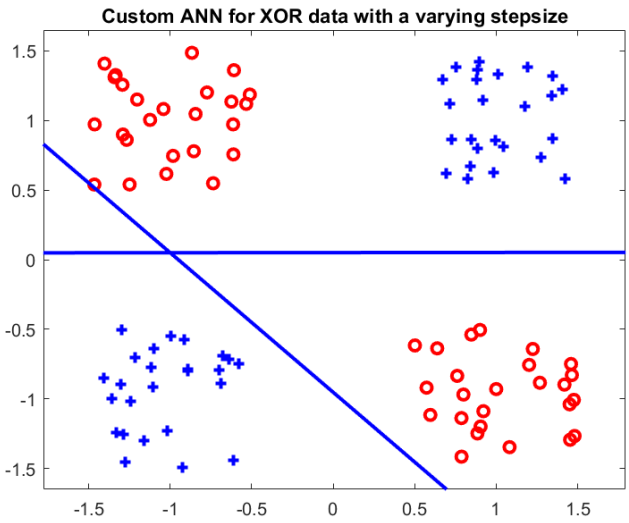

Figure 3: Plotting the weight vectors associated with the hidden layer artificial neurons on the XOR data as classification planes. These weight vectors are optimized using a varying step size steepest descent.

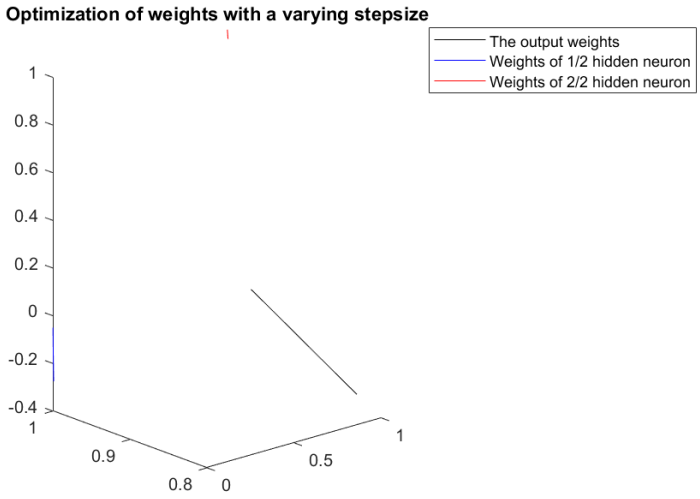

Figure 4: Plotting the change in weights as they are optimized over iterations. Done with a varying step size steepest descent method.

Discussion

The results for the core ideas tested in part two are rather straight forward. Beginning by plotting the initial XOR dataset generated by our script we see their original labels associated with the data, this is seen in Figure 1. After running the fixed steep descent method which in turn repeatedly calls the annfun function that finds the gradient of the network allowing us to optimize the weight arguments based on a number of iterations limit or norm gradient limit.

Plotting the classified XOR data after being passed through the artificial neural network, we see the labels match their original classifications and that the hidden layer weights near perfectly act as visual classifiers for the XOR data.

To help visualize how the weights converge to satisfactory values we plotted them using the MATLAB function plot3. This is shown in Figure 2. As the steepest descent method we implement is in fact a fixed step size version we notice that for the blue and black “tracks” that there seems to be oscillation around a certain part of each. This is the same problem as seen in scalar optimization with fixed steepest descent that we oscillate around a local minimum. This could be improved upon with a varying step size.

As an extra task we were permitted to attempt this problem except with a step size which varies. This was implemented with Armijo’s Condition as found in the steepvary function. Some interesting results arose from attempting this. First off, by referring to Figure 3, we see that this varying step size method produces hidden weight vectors which more appropriately classify the XOR dataset. This result seems intuitive but when we look at the number of steps this method took, which happened to be 5000, this should catch our attention as this is notably higher than the fixed step size method. Looking at Figure 4 this result becomes even more odd as the weight “tracks” seem relatively straight with very little to no oscillation about a local minimum as seen in Figure 2. We can conclude this must arise due to a small step size, we set it to 0.001 and although the varying step size method results in a higher level of accuracy it requires numerous more small steps as the steepest descent method must now satisfy Armijo’s Condition which will decrease the step size on certain iterations.

Throughout this part of the assignment, the concepts of partial derivatives, networks composed of artificial neural networks and gradients are fundamental. As they are seen in a non-traditional topic such as neural networks this assignment really tested the understanding of these topics.