EE 565 - Machine Learning I
Project 3 Report

Jorge A. Garcia

***Abstract*—In this report, the perceptron model is implemented and training using online and batch methods is implemented to build a linear classifier. Experimentation of the optimization routines Gradient Descent, Newton's method and the Levenberg-Marquardt method are explored and tested with different cost functions.**

***Index Terms*—Machine Learning, Data Science**

I. Introduction

The perceptron model marks the beginning of neural networks, which are modern and powerful models capable of solving complex problems. The perceptron is a linear classifier, and its performance on the Double Moon and Gaussian XOR data sets will be explored using online and batch training routines to find the optimal weights for the model. In further sections, the problem of optimization, crucial for the learning aspect of neural networks, is discussed. Implementation of Gradient Descent, Newton's method and the Levenberg-Marquardt method on three different cost functions is done, and strategies to find optimal learning rates and other alternatives such as line searches are implemented.

II. Perceptron Model

A function called "perceptron" is implemented under a file "perceptron.py". This function is repeatedly used in the following sections.

III. Online Perceptron Learning

A class "PerceptronClassifier" is implemented in Python and used to classify the data sets in the following sections. The class implements the online fitting algorithm to find the optimal weights to classify a data set using the previously defined "perceptron" function. The Double Moon data set with $N = 500$, $r = 1$, $w = 0.6$ and $d \in \{0.5, 0, -0.5\}$ is used. All models are trained for a single epoch.

A. Convergence due to initialization: Null Weights

The perceptron classifier is fitted and evaluated with thirty different Double Moon distributions. An initial guess of $\vec{w}_0 = (0, 0)$ is used for the weights. The accuracy of prediction is measured after every weight update (every iteration through a data point), and the average accuracy per weight update iteration across the 30 trials is calculated. This measurement of average accuracy per iteration is shown as the top curve in figures 1-3, one for each possible d value. The bottom plot in these Figures shows the decision regions the model converges to for one of the data distributions, with the predicted class of the data overlain. Incorrect predictions are represented as red stars.

Figure 1 shows the resulting average accuracy and a decision region for the case $d = 0.5$. The model achieves perfect accuracy, which is expected for this particular distribution as it is a perfectly linearly separable problem. The weights quickly converge, roughly reaching 100% accuracy at $\sim$80 iterations.

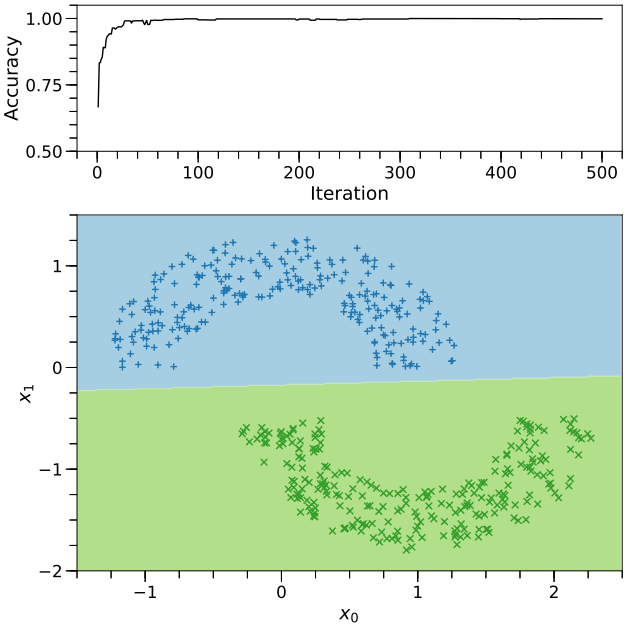

Fig. 1. Double Moon distribution with $d = 0.5$. (Top) Average model accuracy per weight update across 30 distributions using online learning. (Bottom) Decision region and predictions for the data points.

Figure 2 shows the resulting plots for the case $d = 0$. This data set may or may not be linearly separable depending on the data-generating seed (points may or may not overlap at $x_2 = 0$), with the probability of finding an ideal linear decision boundary decreasing with the number of data points. In the case of the data set shown in the figure, the perceptron model achieves a final average accuracy of 96.09%, converging to this accuracy at $\sim$480 iterations. The data distribution shown may be linearly separable, but the model seems to be slightly skewed towards the green class. This is likely from the random sorting of the data as the weights are updated, and given another epoch it could be possible to minimize the error further, especially given how late into the training this accuracy was achieved.

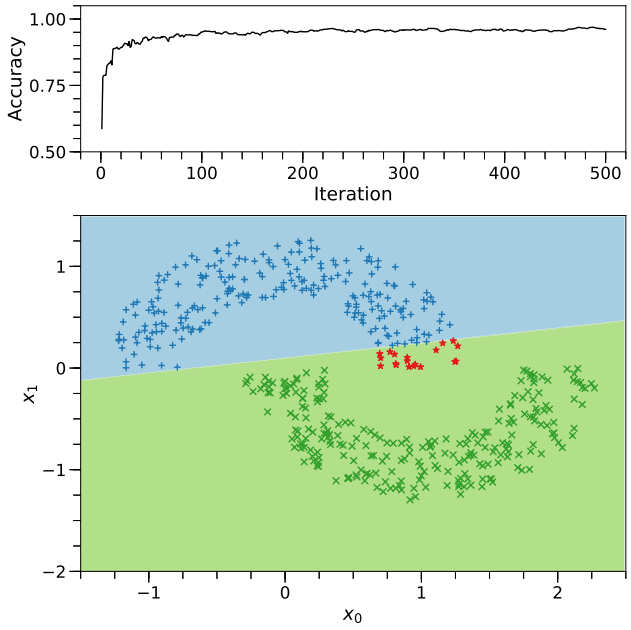

Fig. 2. Double Moon distribution with $d = 0$. (Top) Average model accuracy per weight update across 30 distributions using online learning. (Bottom) Decision region and predictions for the data points.

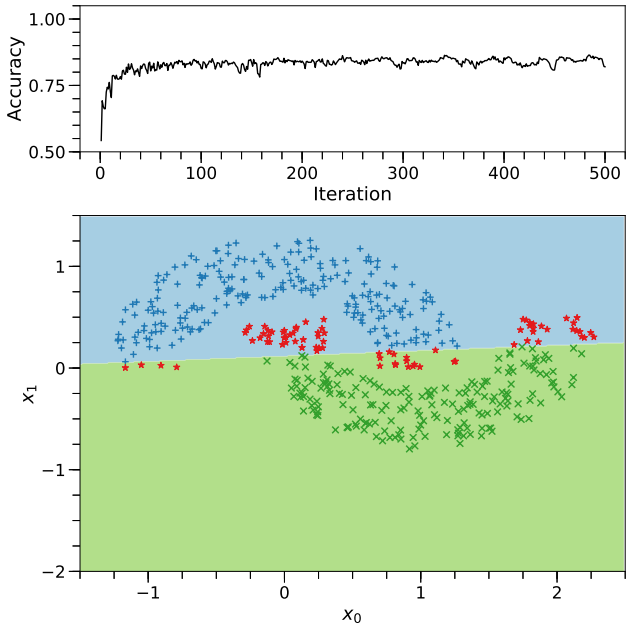

Fig. 3. Double Moon distribution with $d = -0.5$. (Top) Average model accuracy per weight update across 30 distributions using online learning. (Bottom) Decision region and predictions for the data points.

Finally, Figure 3 contains the results for the case $d = -0.5$. This is not a linearly separable problem due to the intertwining of the two classes, and as such the model cannot define a perfect decision boundary. The average accuracy of the perceptron is of 82.03% at the last iteration. Further training may or may not slightly increase the accuracy, given the volatility seen in the accuracy curve.

B. Convergence due to initialization: Uniform Distribution

The previous setting of 30 different Double Moon distributions is repeated, except initialization of the starting weights is drawn from a uniform distribution, such that $\vec{w}_0 \in [-0.5, 0.5]$. The accuracy at each weight update is measured, and the average accuracy across the 30 trials is plotted against the iterations in Figure 4. The initial accuracy is lower, which makes sense since the optimal decision regions are closer to $\vec{w}s = (0, 0)$ as seen in Figures 1 - 3. As the weights are updated, the increase in accuracy follows a more linear trend instead of the more hyperbolic behavior seen before. Given this slower increase in accuracy, the resulting accuracy after a single accuracy is considerably lower than before. More epochs for training would be required in order to achieve the optimal model performance.

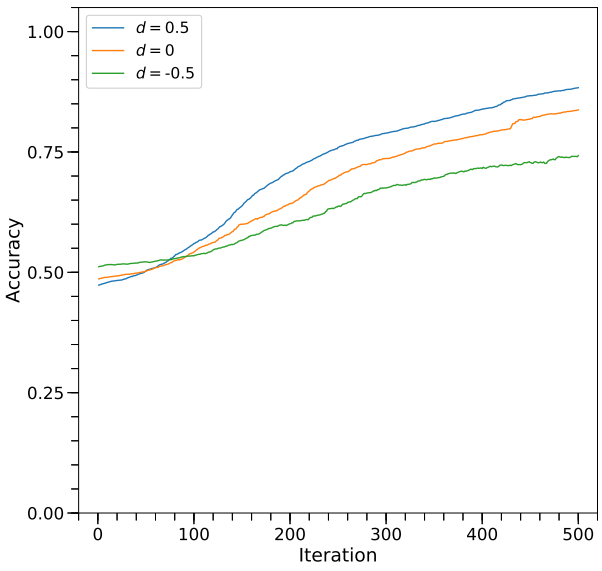

Fig. 4. Average accuracy per weight update across 30 distributions using randomly initialized weights $\vec{w}_0 \in [-0.5, 0.5]$.

IV. Batch Perceptron Learning

The "PerceptronClassifier" class used before also contains a batch training routine, and is the one used for the following section. Two different distributions are used: the Double Moon distribution with $N = 500$, $r = 1$, $w = 0.6$ and $d \in \{0.5, 0, -0.5\}$, and the Gaussian XOR distribution with $N = 500$, $\mu = 0$ and $\sigma = 0.5$ are used. For all data sets used, the model is trained for 100 epochs, the weights are initialized from a uniform distribution $\vec{w}_0 = [-5, 5]$.

A. Double Moon Data Set

A random Double Moon data set with $d = 0$ is generated and fitted for. The resulting decision region from the model is shown in Figure 5. The weights found are able to minimize

the loss to zero, allowing the model to perfectly fit for the training data set.

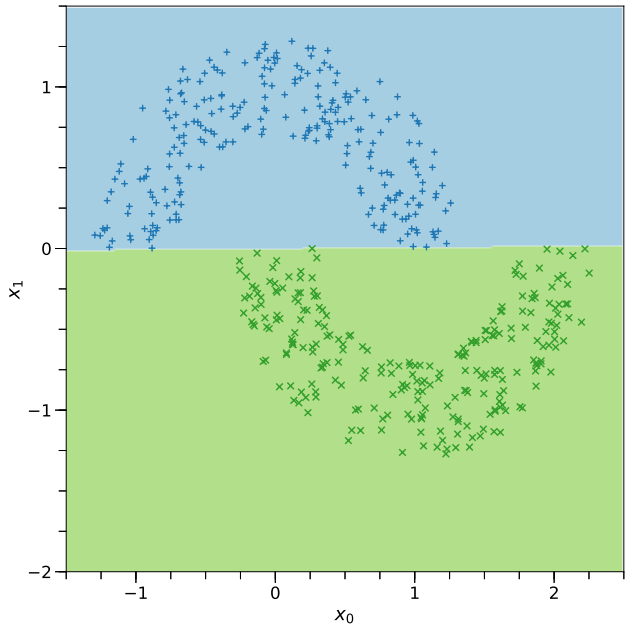

Fig. 5. Possible decision region using batch training from a Double Moon distribution with $d = 0$.

Further evaluation of the batch routine is done by generating 30 Double Moon data sets and fitting a perceptron model to them. The accuracy of a model is evaluated after every epoch, and the average accuracy per epoch is calculated across the 30 trials. The resulting average accuracy and cost for the cases of $d = 0.5, 0, -0.5$ are shown in Figure 6. The perceptron performs exceptionally well in the cases of $d = 0.5, 0$, being able to achieve perfect performance within $\sim$15 and $\sim$20 epochs respectively. This is reflected in the cost plot, as the models achieve the ideal minimal cost of $J = 0$. The model does not perform as well for the case $d = -0.5$, which is expected since it is not a linearly separable problem, achieving a final accuracy of $\sim 85\%$.

B. Gaussian XOR Data Set

The perceptron model is now used on the Gaussian XOR data set. A possible decision region for this data set is shown in Figure 7. This is in an impossible data set for the perceptron to classify as it is not linearly separable. The model decides to classify the entirety of the data set as a single class due to the slight imbalance of there being more green "x" than blue "+" present.

This is further explored in Figure 8, showing the average accuracy and loss across 30 different Gaussian XOR data sets as done before. The model "learns" at first, as seen in the lowering cost in the first 5 iterations, but shortly after stagnates and remains fairly constant. While there is some optimization in loss, this did not translate into an improvement in accuracy, as the model ended up guessing through the entirety of training.

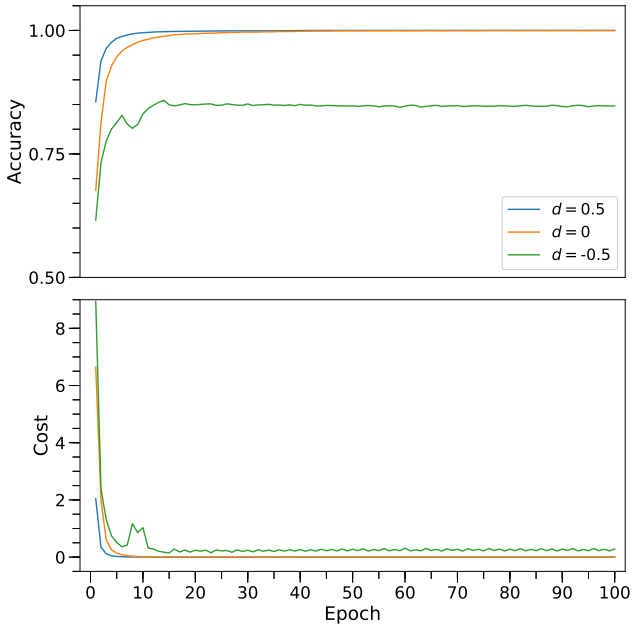

Fig. 6. Average accuracy (top) and average cost (bottom) per epoch across 30 data sets for Double Moon distributions using various d-parameters.

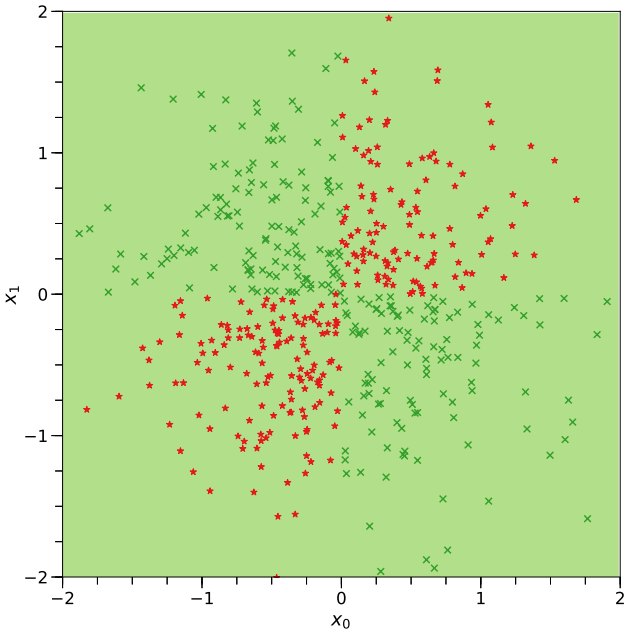

Fig. 7. Possible decision region using batch training from a Gaussian XOR data set.

V. Gradient Descent with a Quadratic Cost Function

The following section is based off the Gradient Descent code provided by Dr. Sandoval, with some modifications in order to address the needs of the project. The routine is limited to a maximum of 50 iterations in case convergence is not achieved.

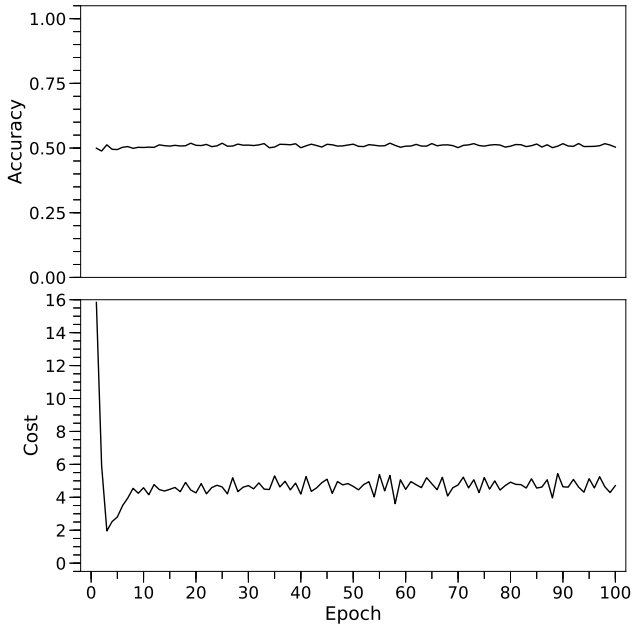

Fig. 8. Average accuracy (top) and average cost (bottom) per epoch across 30 data sets for Gaussian XOR distributions.

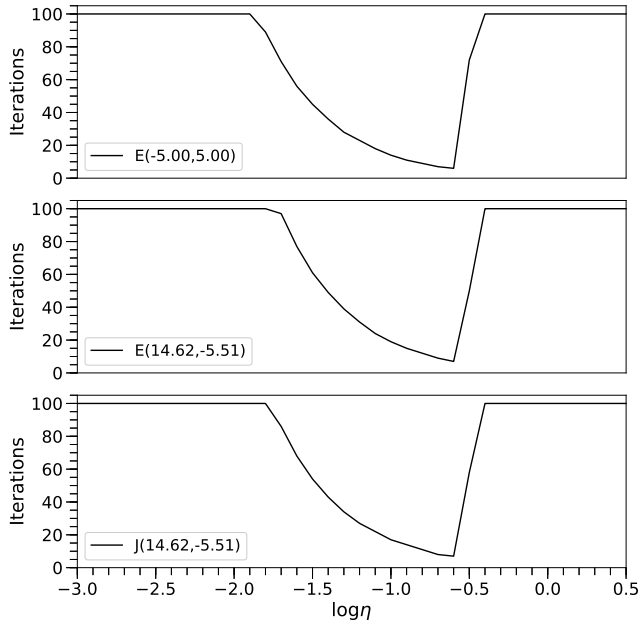

Fig. 9. Quadratic cost function: (Top) Iterations until weight error $E(\vec{w})$ convergence from initial guess $\vec{w}_0 = (-5, 5)$. (Middle) Iterations until weight error $E(\vec{w})$ convergence from random normal initial guess. (Bottom) Iterations until cost $J(\vec{w})$ convergence from random normal initial guess.

A. Impact of Learning Rate on Convergence

To determine the best learning rate for the given cost function, the performance of gradient descent is observed when determining convergence based off of the weight error or from the evaluated cost. Once a predetermined threshold was reached, set to be $E(\vec{w}) \leq \epsilon_E = 0.5$ and $J(\vec{w}) \leq \epsilon_J = 0.5$, the weight-updating loop would break, and the number of iterations undergone recorded. Learning rates are explored in logarithmic space as they tend to be very small numbers and noticeable difference in performance is only seen when approaching magnitudes of order difference between them. The result of the three cases analyzed are shown in Figure 9.

The first case uses an initial weight guess of $\vec{w}_0 = (-5, 5)$, and uses the weight error to measure convergence, seen at the top of Figure 9. The quickest convergence is found to be at a learning rate $\eta = 10^{-0.6} \approx 0.25$, converging in 6 iterations. A second case using an initial guess drawn from a normal distribution $\vec{w}_0 \in \mathcal{G}(\mu = 0, \sigma = 3)$ is used, and the resulting plot can be seen in the middle of Figure 9. Using these weights, the weight error still converges the quickest when using a learning rate of $\eta \approx 0.25$, stopping at 7 iterations.

These same random weights are used again, but this time using the value of the cost function $J(\vec{w})$ instead to determine convergence, seen at the bottom of Figure 9. The same learning rate of $\eta \approx 0.25$ is found to converge the quickest at 7 iterations. This is a promising result, as it indicates that minimization of the cost function $J(\vec{w})$ can be used as a proxy of the weight error $E(\vec{w})$ between a given set of weights and the optimal weights $\vec{w}^*$. Convergence is decided using $J(\vec{w})$ since in a real setting, the optimal weights $\vec{w}^*$ cannot be known and thus the weight error $E(\vec{w})$ cannot be determined.

B. Comparison of Static vs Adaptive Learning Rate

An Exact Line Search routine is implemented and compared to the performance of the previously found optimal learning rate. The Line Search calculates the cost from a change in weights using learning rates given $\ln(\eta) \in [-3, 0]$. Figure 10 shows the location in weight space at each iteration for both methods. For this particular cost function, both approaches have similar performance. The optimal learning rate converges to the minimum in 24 iterations, whereas the adaptive learning rate reaches it in 23. The quadratic cost function is a concave problem though, and as such is an easy function to optimize.

VI. Gradient Descent with Rosenbrock's Cost Function

Using the same routine as before albeit with Rosenrbock's cost function, and an initial weight guess of $\vec{w}_0 = (-0.5, 1)$. The location in weight space of each update is explored, using learning rates of $\eta \in \{10^{-4}, 10^{-3}, 10^{-2}\}$ and an Exact Line Search. The resulting trajectories are seen in Figure 11. The routine is limited to a maximum of 50 iterations in case convergence is not achieved.

Using a fixed learning rate is not the preferable approach for this cost function, as the two smaller learning rates converge far from the optimal weights, and the larger learning rate of $\eta = 0.01$ completely diverges. In contrast, the Exact Line Search reaches the iteration limit of 50 but is very close to the optimal weights of $\vec{w}^* = (1, 1)$, with a difference $\Delta\vec{w} = (0.007, 0.014)$ between the optimal and found weights. For this cost function, an adaptive learning rate would the preferred approach to solve for the optimal weights.

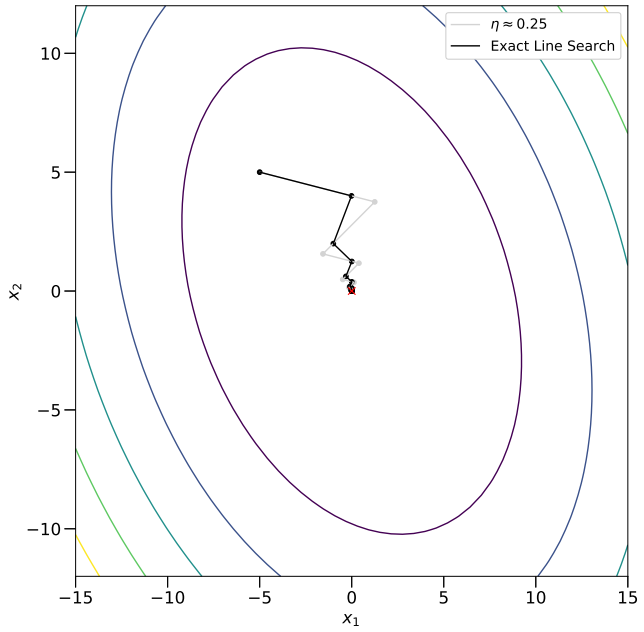

Fig. 10. Comparison of weight update trajectories between the optimal learning rate and an Exact Line Search for the quadratic cost function using Gradient Descent.

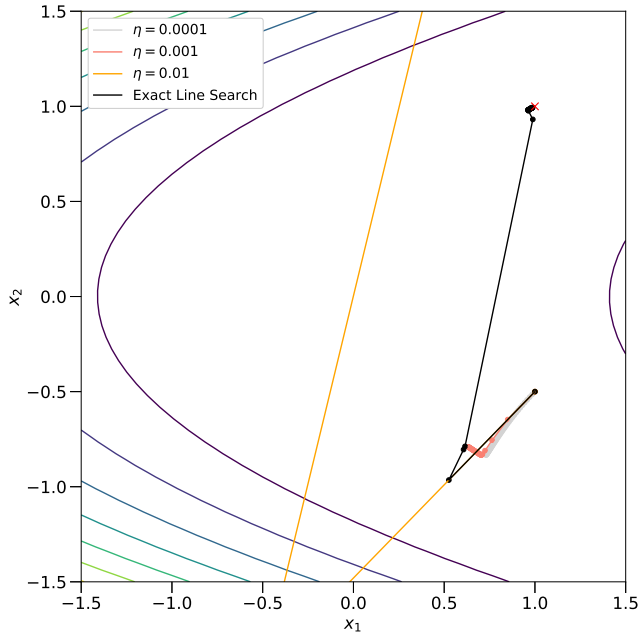

Fig. 11. Comparison of weight update trajectories between different learning rates and an Exact Line Search for Rosenbrock's cost function using Gradient Descent.

VII. Gradient Descent with Himmelblau's Cost Function

The process is repeated, but now with Himmelblau's cost function. The initial weight guess used for this section was drawn from a normal distribution, $\vec{w}_0 \in \mathcal{G}(\mu = 0, \sigma = 3)$. The routine is limited to a maximum of 50 iterations in case convergence is not achieved.

A. Performance of Gradient Descent

Initial exploration of optimization of the cost function is done using learning rates obtained from defining them in logarithmic space $\ln(\eta) \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$. The resulting trajectories in weight space can be seen in Figure 12.

The two smallest learning rates fail to converge to the optimal weight, as the update size is too small and the maximum number of iterations is not enough for these learning rates. With a learning rate of $\eta = 10^{-2}$, the optimal weight $\vec{w}^* = (-3.78, -3.28)$ is found after 16 iterations. Too big of a learning rate though, and the algorithm will diverge completely.

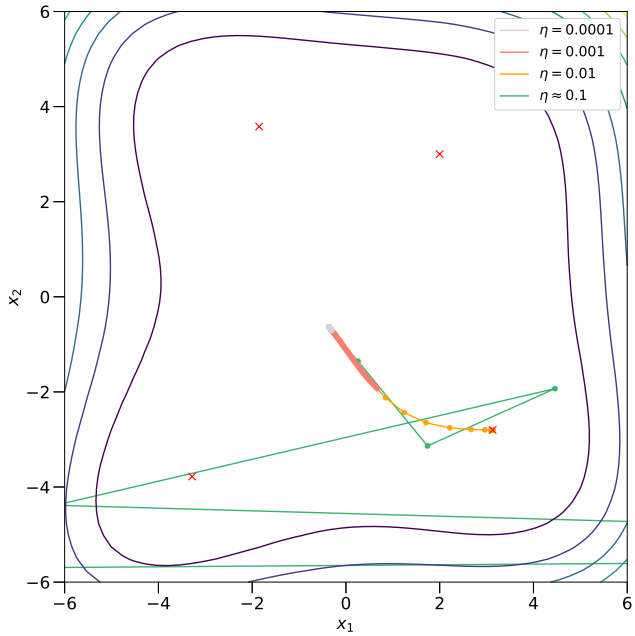

Fig. 12. Comparison of weight update trajectories between different learning rates for Himmelblau's cost function using Gradient Descent.

B. Cost Evaluation due to Weight Updates

The associated cost per weight update from the previous learning rates can be seen in Figure 13. The smaller learning rates have very slow decays, and could converge to a minimum given more iterations to update. In comparison, the cost for the learning rate of $\eta = 10^{-2}$ quickly reaches the minimum cost and remains there. The cost for the largest learning rate instantly shoots up and diverges after the first iteration, hinting at it being a bad choice for this cost function.

C. Optimal Learning Rate

The same process used in Section V-A is implemented to find the best learning rate for this cost function. Learning rates are explored in logarithmic space, such that $\ln(\eta) \in$

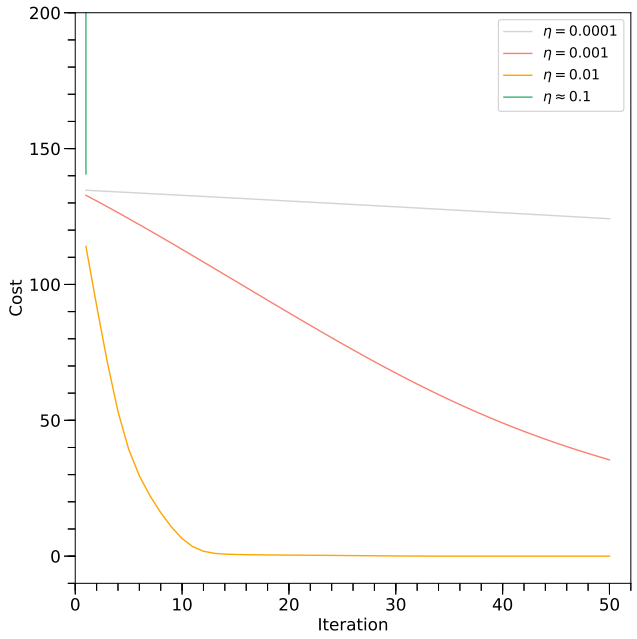

Fig. 13. Cost per weight update for different learning rates for Himmelblau's cost function using Gradient Descent.

$[-4, -1.5]$, and using the same set of initial weights. Convergence is determined as achieving a cost $J(\vec{w}) \leq \epsilon_J = 0.5$. The results are shown in Figure 14. A learning rate of $\eta = 10^{-1.8} \approx 0.016$ results in the quickest convergence, reaching the threshold within 9 iterations. The learning rate of $\eta = 10^{-2}$ used in the previous part was close to this optimal learning rate, and hence why it was able to converge within the allotted maximum iterations.

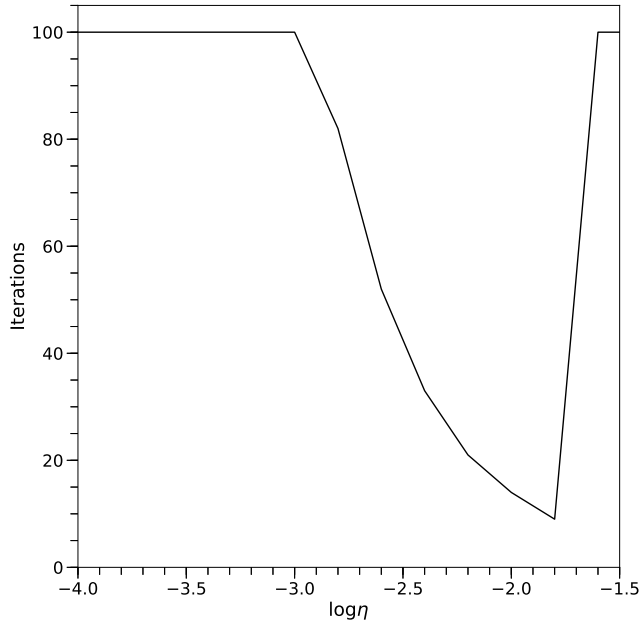

Fig. 14. Iterations until cost $J(\vec{w})$ convergence for Himmelblau's cost function using Gradient Descent.

VIII. Newton's Method with Quadratic Cost Function

Newton's method is now used to find the optimal weight for the quadratic function, implementing the Hessian of the cost function into the weight update:

$$\Delta\vec{w} = -\boldsymbol{H}^{-1}\vec{g} \tag{1}$$

A random, normally distributed initial weight guess $\vec{w}_0 \in \mathcal{G}(\mu = 0, \sigma = 3)$ is used. Three different learning rates of $\eta \in \{1, 0.1, 0.01\}$ are experimented with, and the resulting trajectories in weight space are shown in Figure 15. The failure to converge of the smallest learning rate, in contrst to the other two, suggests that second-order iterative descent methods benefit from larger learning rates. A learning rate of $\eta = 1$ instantly updates towards the optimal weights $\vec{w}^* = (0, 0)$.

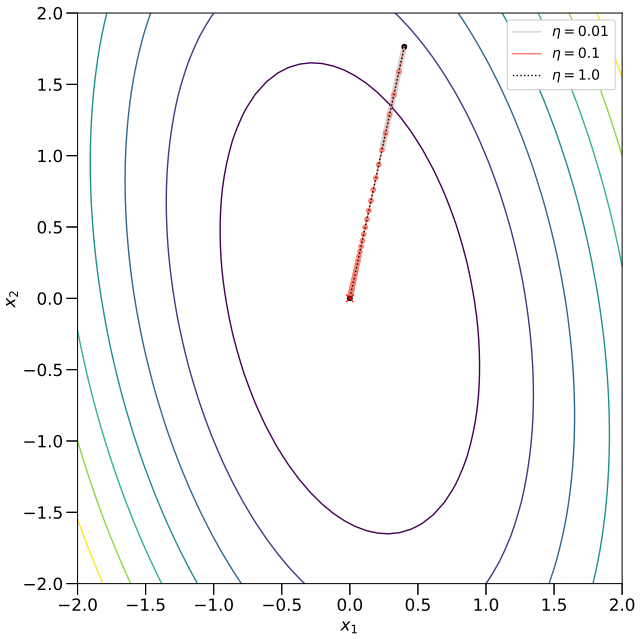

Fig. 15. Comparison of weight update trajectories between different learning rates for the quadratic cost function using Newton's method.

IX. Newton's Method with Himmelblau's Cost Function

Newton's method is now implemented to optimize for Himmelblau's cost function. Using random initial weights from the normal distribution as in the section before, and the same learning rates are used as well. Figure 16 shows the trajectories in the weight space for this function. Again, low learning rates fail converge, and a learning rate of $\eta = 1$ achieves the quickest convergence, reaching the optimal weights in 13 steps. What is interesting to note is the fact that the second-order term first takes the weights somewhere distant in space to find the optimal path to the minimum solution.

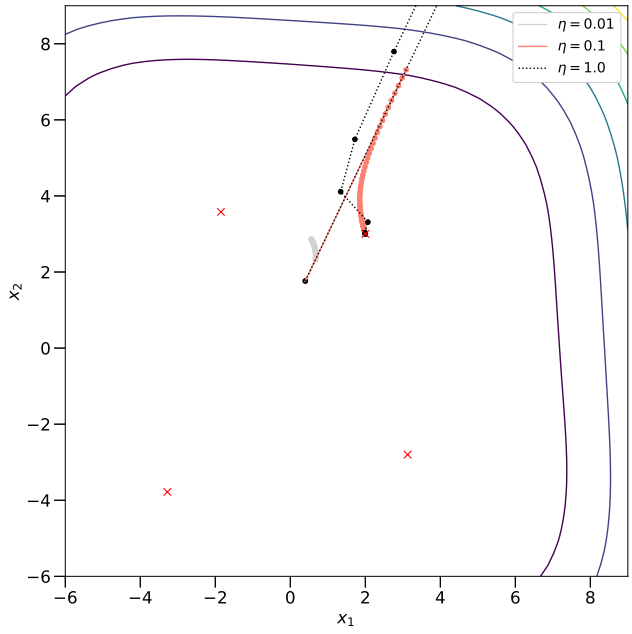

Fig. 16. Comparison of weight update trajectories between different learning rates for Himmelblau's cost function using Newton's method.

X. Levenberg-Marquardt Method with Himmelblau's Cost Function

The Levenberg-Marquardt method essentially adds a regularization term λ to the Hessian of $J(\vec{w})$L

$$\Delta\vec{w} = (\mathbf{H} + \lambda\mathbf{I})^{-1}\vec{g} \tag{2}$$

The initial weight guess from the previous section is used again in order to draw comparison, and a learning rate of $\eta = 1$ is used throughout. The regularization term is now varied, with $\lambda \in \{0, 1, 10, 100\}$, and the resulting trajectories in weight space can be seen in Figure 17. Given $\lambda = 0$, this results in the same update as Newton's method seen in the section before. As λ increases, the path taken is shortened, although there is not much of a change in iterations for convergence. For $\lambda = 0, 1, 10$ it took 13, 12 and 13 iterations respectively to find the optimal solution. Large values of λ result with a weight update approximating that of Gradient Descent, as seen in the case of $\lambda = 100$. For this particular function, the Levenberg-Marquardt method did not offer much advantage over Newton's method.

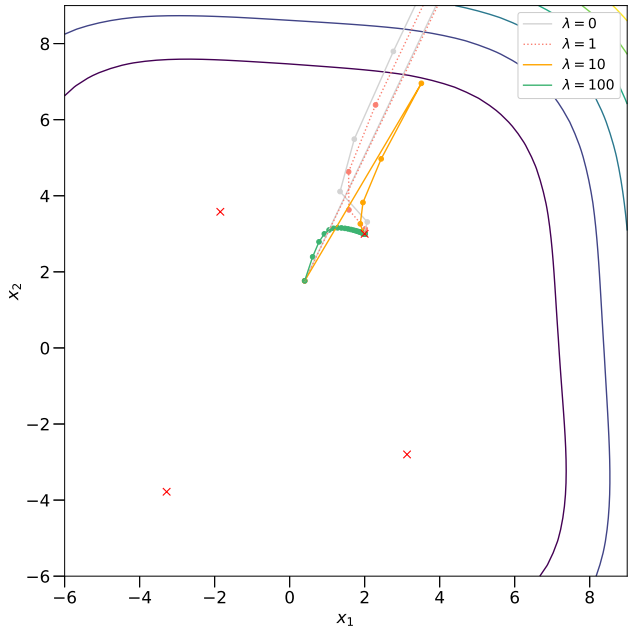

Fig. 17. Comparison of weight update trajectories between different learning rates for Himmelblau's cost function using the Levenberg-Marquardt method.

XI. Conclusion

In this report, the perceptron model was used to define a linear classifier for the Double Moon and Gaussian XOR data sets using online and batch training routines. Optimization techniques were discussed, and strategies for finding optimal learning rates and alternative methods such as line searches were implemented.