ECE 174 Linear and Nonlinear Optimization
Prof. Piya Pal

Mini Project 2: Training Neural Networks using Non Linear Least Squares

December 4, 2022

Conner Hsu (A16665092)

Contents

1 Introduction

In this report, I will program a neural network to approximate different non-linear functions. In particular, the the network will approximate a non-linear function, $g(\mathbf{x})$ that maps from $\mathbb{R}^3 \to \mathbb{R}$. The function that will model this neural network will be given by the expression,

$$\begin{aligned} f_{\mathbf{w}}(\mathbf{x}) = & w_1\phi(w_2x_1 + w_3x_2 + w_4x_3 + w_5) \\ & + w_6\phi(w_7x_1 + w_8x_2 + w_9x_3 + w_{10}) \\ & + w_{11}\phi(w_{12}x_1 + w_{13}x_2 + w_{14}x_3 + w_{15}) \\ & + w_{16} \end{aligned}$$

where $\mathbf{w}$ represents a vector in $\mathbb{R}^{16}$ that will paremeterize this network and ϕ is the tanh function. We are particularly interested in determining $\mathbf{w}$ such that $f_{\mathbf{w}}$ best approximates the function $g(\mathbf{x})$. Let $\mathbf{x}_i \in \mathbb{R}^3$ where $i = 1, 2, \ldots, N$. Let

$$y_i = g(\mathbf{x}_i)$$

We can quantify the error between the approximation, $f_{\mathbf{w}}(\mathbf{x})$ and $g(\mathbf{x})$ using a sum of squared errors.

$$\sum_{i=1}^{N} \left(f_{\mathbf{w}}(\mathbf{x}_i) - y_i\right)^2$$

For brevity in later sections, let

$$r_i(\mathbf{w}) = f_{\mathbf{w}}(\mathbf{x}_i) - y_i$$

and let

$$\mathbf{r}(\mathbf{w}) = \begin{bmatrix} r_1(\mathbf{w}) & r_2(\mathbf{w}) & \ldots & r_N(\mathbf{w}) \end{bmatrix}^T$$

$\mathbf{r}(\mathbf{w})$ represents the error of neural network for every test point. In addition to minimizing $\mathbf{r}(\mathbf{w})$, it would also be of interest to minimize $\mathbf{w}$, thus the following loss function should be minimized,

$$l(\mathbf{w}) = \sum_{i=1}^{N} r_i(\mathbf{w})^2 + \lambda||\mathbf{w}||_2^2 = \left|\left|\begin{bmatrix} \mathbf{r}(\mathbf{w}) \\ \sqrt{\lambda}\mathbf{w} \end{bmatrix}\right|\right|_2^2 = ||\mathbf{h}(\mathbf{w})||_2^2$$

where λ represents how much we care that $||\mathbf{w}||_2^2$ get minimized.

2 Levenberg-Marquardt algorithm

In order to determine the weights, $\mathbf{w}$ we will use the Levenberg-Marquardt algorithm. The algorithm will repeat the following steps for $k = 1, 2, \ldots, k_{\max}$. $k_{\max}$ is determined when a certain stopping criterion is met. Choose $\gamma_1 > 0$. The values of $\mathbf{w}_1$ will be drawn from a uniform distribution.

1. Linearize $\mathbf{h}(\mathbf{w})$ at $\mathbf{w} = \mathbf{w}_k$, i.e. compute the first order Taylor series approximation of $\mathbf{h}(\mathbf{w})$.

$$\mathbf{h}(\mathbf{w}) \approx \hat{\mathbf{h}}(\mathbf{w}, \mathbf{w}_k) = \mathbf{h}(\mathbf{w}_k) + \mathrm{D}\mathbf{h}|_{\mathbf{w}=\mathbf{w}_k}(\mathbf{w} - \mathbf{w}_k)$$

2. Compute the next iterate. Set $\mathbf{w}_{k+1}$ to the minimizer of

$$||\hat{\mathbf{h}}(\mathbf{w}, \mathbf{w}_k)||_2^2 + \gamma_k||\mathbf{w} - \mathbf{w}_k||_2^2$$

3. Analyze the next iterate.
   - If $l(\mathbf{w}_{k+1}) < l(\mathbf{w}_k)$, use $\mathbf{w}_{k+1}$ as the next iterate and set $\gamma_{k+1} = 0.8\gamma_k$.
   - Otherwise, increase γ and use the current iterate as the second iterate, i.e. $\gamma_{k+1} = 2\gamma_k$ and $\mathbf{w}_{k+1} = \mathbf{w}_k$.

In the proceeding subsections, details will be provided on how to complete each of these steps and what stopping criterion will be used.

2.1 Linearizing $\mathbf{h}(\mathbf{w})$

Linearizing $\mathbf{h}(\mathbf{w})$ is of interest because doing so allows us to formulate the problem as a linear least squares problem, making it much easier to compute the next iterate. In order to linearize $\mathbf{h}(\mathbf{w})$, D$\mathbf{h}$ must be computed.

$$\mathrm{D}\mathbf{h} = \begin{bmatrix} \mathrm{D}\mathbf{r} \\ \sqrt{\lambda}\mathrm{D}\mathbf{w} \end{bmatrix}$$

Determining D$\mathbf{r}$:

$$\mathrm{D}\mathbf{r} = \begin{bmatrix} (\nabla_{\mathbf{w}} r_1(\mathbf{w}))^T \\ (\nabla_{\mathbf{w}} r_2(\mathbf{w}))^T \\ \vdots \\ (\nabla_{\mathbf{w}} r_N(\mathbf{w}))^T \end{bmatrix} = \begin{bmatrix} (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}_1))^T \\ (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}_2))^T \\ \vdots \\ (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}_N))^T \end{bmatrix}$$

Each row, $(\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}_i))^T$ can be individually computed like so,

$$\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}_i) = \begin{bmatrix} \frac{\partial f_{\mathbf{w}}}{\partial w_1} \\ \frac{\partial f_{\mathbf{w}}}{\partial w_2} \\ \vdots \\ \frac{\partial f_{\mathbf{w}}}{\partial w_{16}} \end{bmatrix} = \begin{bmatrix} \phi(w_2 x_1 + w_3 x_2 + w_4 x_3 + w_5) \\ w_1 \dot{\phi}(w_2 x_1 + w_3 x_2 + w_4 x_3 + w_5) x_1 \\ w_1 \dot{\phi}(w_2 x_1 + w_3 x_2 + w_4 x_3 + w_5) x_2 \\ w_1 \dot{\phi}(w_2 x_1 + w_3 x_2 + w_4 x_3 + w_5) x_3 \\ w_1 \dot{\phi}(w_2 x_1 + w_3 x_2 + w_4 x_3 + w_5) \\ \phi(w_7 x_1 + w_8 x_2 + w_9 x_3 + w_{10}) \\ w_6 \dot{\phi}(w_7 x_1 + w_8 x_2 + w_9 x_3 + w_{10}) x_1 \\ w_6 \dot{\phi}(w_7 x_1 + w_8 x_2 + w_9 x_3 + w_{10}) x_2 \\ w_6 \dot{\phi}(w_7 x_1 + w_8 x_2 + w_9 x_3 + w_{10}) x_3 \\ w_6 \dot{\phi}(w_7 x_1 + w_8 x_2 + w_9 x_3 + w_{10}) \\ \phi(w_{12} x_1 + w_{13} x_2 + w_{14} x_3 + w_{15}) \\ w_{11} \dot{\phi}(w_{12} x_1 + w_{13} x_2 + w_{14} x_3 + w_{15}) x_1 \\ w_{11} \dot{\phi}(w_{12} x_1 + w_{13} x_2 + w_{14} x_3 + w_{15}) x_2 \\ w_{11} \dot{\phi}(w_{12} x_1 + w_{13} x_2 + w_{14} x_3 + w_{15}) x_3 \\ w_{11} \dot{\phi}(w_{12} x_1 + w_{13} x_2 + w_{14} x_3 + w_{15}) \\ 1 \end{bmatrix}$$

where $\mathbf{x}_i = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$. Now, determining D$\mathbf{w}$:

$$\mathrm{D}\mathbf{w} = \begin{bmatrix} (\nabla_{\mathbf{w}} w_1)^T \\ (\nabla_{\mathbf{w}} w_2)^T \\ \vdots \\ (\nabla_{\mathbf{w}} w_{16})^T \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} = I_{16}$$

Now, $\mathbf{h}(\mathbf{w})$ can be linearized around a fixed point, $\mathbf{w}_k$, using a 1st order Taylor series expansion:

$$\mathbf{h}(\mathbf{w}) \approx \hat{\mathbf{h}}(\mathbf{w}, \mathbf{w}_k) = \mathbf{h}(\mathbf{w}_k) + \mathrm{D}\mathbf{h}|_{\mathbf{w}=\mathbf{w}_k}(\mathbf{w} - \mathbf{w}_k)$$

2.2 Computing the next iterate

In order to compute the next iterate, the following minimization problem must be solved for.

$$\begin{aligned} \min_{\mathbf{w}\in\mathbb{R}^{16}} \left(||\hat{\mathbf{h}}(\mathbf{w}, \mathbf{w}_k)||_2^2 + \gamma_k ||\mathbf{w} - \mathbf{w}_k||_2^2 \right) &= \min_{\mathbf{w}\in\mathbb{R}^{16}} \left|\left| \begin{bmatrix} \hat{\mathbf{h}}(\mathbf{w}, \mathbf{w}_k) \\ \sqrt{\gamma_k}(\mathbf{w} - \mathbf{w}_k) \end{bmatrix} \right|\right|_2^2 \\ &= \min_{\mathbf{w}\in\mathbb{R}^{16}} \left|\left| \begin{bmatrix} \mathbf{h}(\mathbf{w}_k) + \mathrm{D}\mathbf{h}|_{\mathbf{w}=\mathbf{w}_k}(\mathbf{w} - \mathbf{w}_k) \\ \sqrt{\gamma_k}(\mathbf{w} - \mathbf{w}_k) \end{bmatrix} \right|\right|_2^2 \\ &= \min_{\mathbf{w}\in\mathbb{R}^{16}} \left|\left| \begin{bmatrix} \mathrm{D}\mathbf{h}|_{\mathbf{w}=\mathbf{w}_k} \\ \sqrt{\gamma_k} I_{16} \end{bmatrix} \mathbf{w} - \begin{bmatrix} \mathrm{D}\mathbf{h}|_{\mathbf{w}=\mathbf{w}_k}\mathbf{w}_k - \mathbf{h}(\mathbf{w}_k) \\ \sqrt{\gamma_k}\mathbf{w}_k \end{bmatrix} \right|\right|_2^2 \end{aligned}$$

Let

$$A = \begin{bmatrix} \mathrm{D}\mathbf{h}|_{\mathbf{w}=\mathbf{w}_k} \\ \sqrt{\gamma_k} I_{16} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathrm{D}\mathbf{h}|_{\mathbf{w}=\mathbf{w}_k} \mathbf{w}_k - \mathbf{h}(\mathbf{w}_k) \\ \sqrt{\gamma_k} \mathbf{w}_k \end{bmatrix}$$

Now, the minimization problem can be written as

$$= \min_{\mathbf{w} \in \mathbb{R}^{16}} ||A\mathbf{w} - \mathbf{b}||_2^2$$

Note that A is full rank because $N(A) = N(D\mathbf{h}|_{\mathbf{w}=\mathbf{w}_k}) \cap N(\sqrt{\lambda_k} I_{16})$ and $N(\sqrt{\lambda_k} I_{16}) = \{\mathbf{0}\}$. This means that this minimization problem has the solution,

$$\mathbf{w} = (A^T A)^{-1} A^T \mathbf{b}$$

The above formula can be used to compute $\mathbf{w}_{k+1}$.

2.3 Stopping criterion

I came up with 4 stopping criterion. The program will stop training if any of the following conditions are met.

1. If the loss, $l(\mathbf{w}_k)$ is small enough.
2. If the RMS error, $\sqrt{\frac{1}{N}||\mathbf{r}(\mathbf{w})||_2^2}$ is small enough.
3. If the change in the loss, $\nabla_{\mathbf{w}} l(\mathbf{w}_k)$ is small enough.
4. If the quantities above don't change for a certain number of iterations.

I will elaborate on the theory behind stopping criterion (3). According to calculus, a local minimum will occur when the derivative of the function is equal to zero. Since we are interested in finding the $\mathbf{w}_k$ that gives a minimum of $l(\mathbf{w})$ (the loss function), computing $\nabla_{\mathbf{w}} l(\mathbf{w})$ is of interest to determine if a given $\mathbf{w}_k$ is at a local minimum of $l(\mathbf{w})$.

$$\begin{aligned}
\nabla_{\mathbf{w}} l(\mathbf{w}_k) &= \nabla_{\mathbf{w}} ||\mathbf{h}(\mathbf{w}_k)||_2^2 \\
&= \nabla_{\mathbf{w}} \left(\sum_{i=1}^{N} r_i(\mathbf{w}_k)^2 + \lambda ||\mathbf{w}_k||_2^2 \right) \\
&= \sum_{i=1}^{N} 2 r_i(\mathbf{w}_k) \nabla_{\mathbf{w}} r_i(\mathbf{w}_k) + \lambda \nabla_{\mathbf{w}} ||\mathbf{w}_k||_2^2 \\
&= 2(\mathrm{D}\mathbf{r}|_{\mathbf{w}=\mathbf{w}_k})^T \mathbf{r}(\mathbf{w}_k) + \lambda \nabla_{\mathbf{w}} ||\mathbf{w}_k||_2^2 \\
&= 2(\mathrm{D}\mathbf{r}|_{\mathbf{w}=\mathbf{w}_k})^T \mathbf{r}(\mathbf{w}_k) + \lambda \nabla_{\mathbf{w}} (w_1^2 + w_2^2 + \cdots + w_{16}^2) \\
&= 2(\mathrm{D}\mathbf{r}|_{\mathbf{w}=\mathbf{w}_k})^T \mathbf{r}(\mathbf{w}_k) + \lambda \begin{bmatrix} 2w_1 \\ 2w_2 \\ \vdots \\ 2w_{16} \end{bmatrix} \\
&= 2(\mathrm{D}\mathbf{r}|_{\mathbf{w}=\mathbf{w}_k})^T \mathbf{r}(\mathbf{w}_k) + 2\lambda \mathbf{w}_k
\end{aligned}$$

If $\nabla_{\mathbf{w}} l(\mathbf{w}_k) = \mathbf{0}$, then $\mathbf{w}_k$ is at a local minimum. To make this work as a stopping criteria we can say that if $\nabla_{\mathbf{w}} l(\mathbf{w}_k)$ is close to being zero, then $\mathbf{w}_k$ is close to being a local minimum, i.e.

$$||\nabla_{\mathbf{w}} l(\mathbf{w}_k) - \mathbf{0}||_2 < L$$

where L is some arbitrarily chosen constant.

In my actual implentation of the Levenberg-Marquardt, however, I mainly used criterion (2) and criterion (4). The distinction between criterion (1) and (2) isn't that much, so I could've easily used (1) instead. However, criterion (4) is particularly important because I noticed that many initializations would have no hope of meeting the the other criterion or would improve too slowly. Criterion (4) helps with these cases by cutting them off earlier, saving me some time.

3 Training and testing data

Training and testing data sets will be generated by drawing from a gaussian distribution such that each input data point, $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$, satisfies the following condition:

$$\max\{|x_1|, |x_2|, |x_3|\} \leq \Gamma$$

where Γ is some positive real number. I will use $N = 500$ data points for training and $N_T = 100$ data points for testing.

For all the training data used in the proceeding sections, $\Gamma = 1$. However, to test the robustness of the neural network, different Γ will be used for testing the neural network.

4 Initialization parameters

There are multiple initialization parameters for training this neural network. This section will summarize all of these parameters and specify what values will be used for each throughout the report.

- λ : specifies how much we care about minimizing $||\mathbf{w}||_2^2$. This parameter will be set to $\lambda = 10^{-5}$ unless otherwise specified.
- Initial weights, $\mathbf{w}_1$: the weights for the first iteration of the Levenberg-Marquardt algorithm. This parameter will typically be initialized by drawing from a uniform distribution in the range of $[-0.1, 0.1)$. For the sake of brevity, a lot of the figures will say that $\mathbf{w}_1 \in [-0.1, 0.1)$ to indicate the range of the uniform distribution it was drawn from.
- Trust factor, γ_1 : specifies how much we initially care about minimizing $||\mathbf{w}_{k+1} - \mathbf{w}_k||_2^2$. This will be set to $\gamma_1 = 10^{-5}$ unless otherwise specified.

In the proceeding sections, a single $\mathbf{w}_1$ was generated and used for all the initializations for the neural network.

5 Results for $g(\mathbf{x}) = x_1 x_2 + x_3$

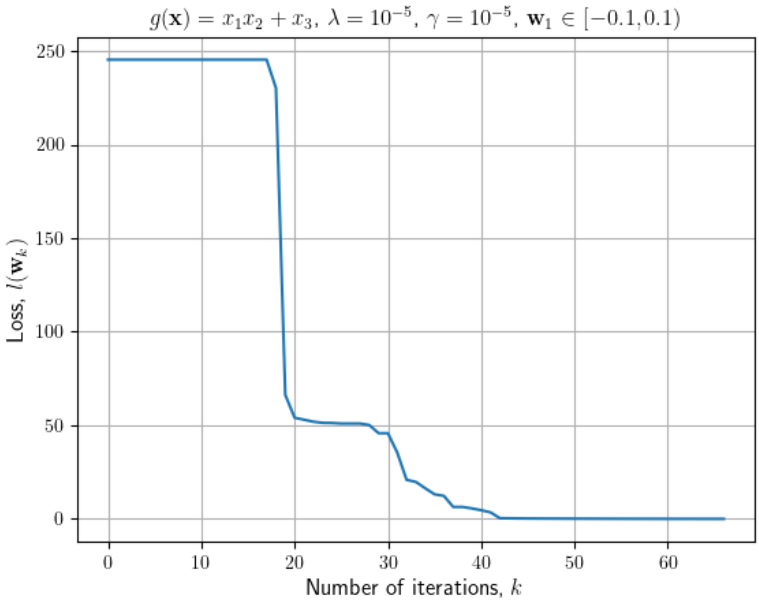

Figure 1: Training loss, $l(\mathbf{w}_k)$ with respect to the number of training iterations, k.

The final loss and RMS error for the initialization used in Figure 1 is 0.025 and 0.0051, respectively.

5.1 Final loss versus varied initializations

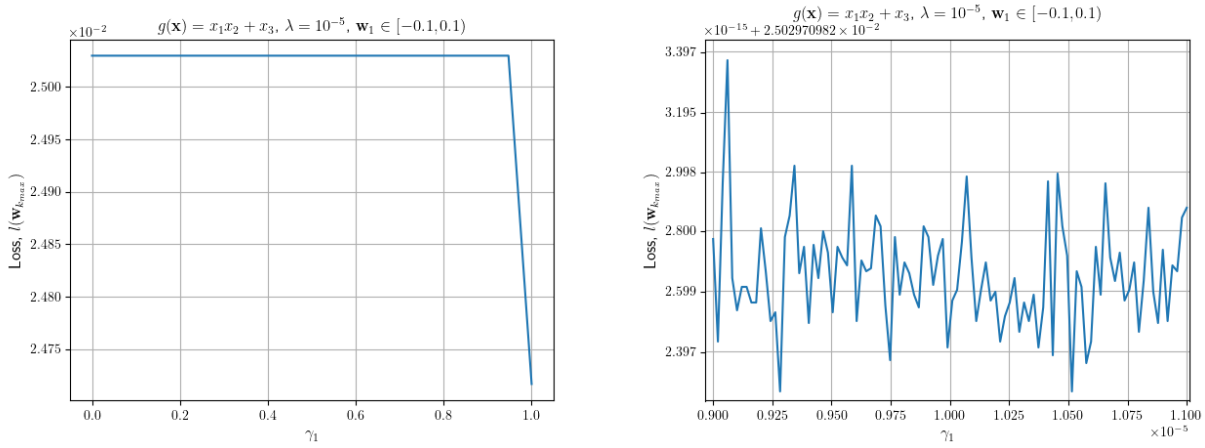

Figure 2: Final training loss with respect to different γ_1.

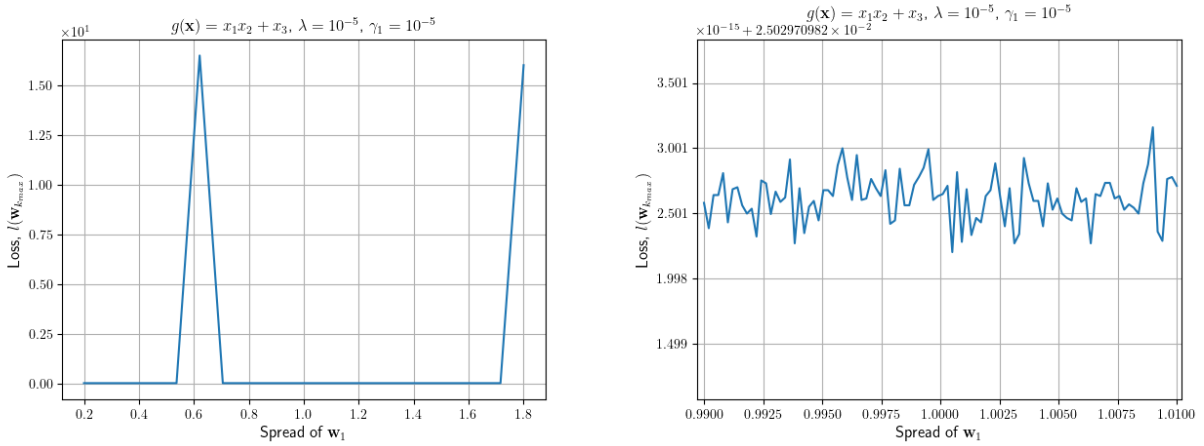

Figure 3: Final training loss with respect to different $\mathbf{w}_1$.

"Spread of $\mathbf{w}_1$" refers to the range of the uniform distribution used to generate $\mathbf{w}_1$ relative to the default range $[-0.1, 0.1)$. Multiply the value on the x-axis by 0.1 to get the new uniform distribution range. e.g if the x-axis reads 1.1, that means $\mathbf{w}_1$ is generated between $[-0.11, 0.11]$.

For both Figures 2 and 3, the zoomed in plots on the left show that it is difficult to say whether or not changing γ_1 or $\mathbf{w}_1$ has much of an effect on the final training loss. From the zoomed out plots, we can see that the final loss stays relatively constant for a large range of γ_1 and $\mathbf{w}_1$.

5.2 Final RMS error versus varied initializations

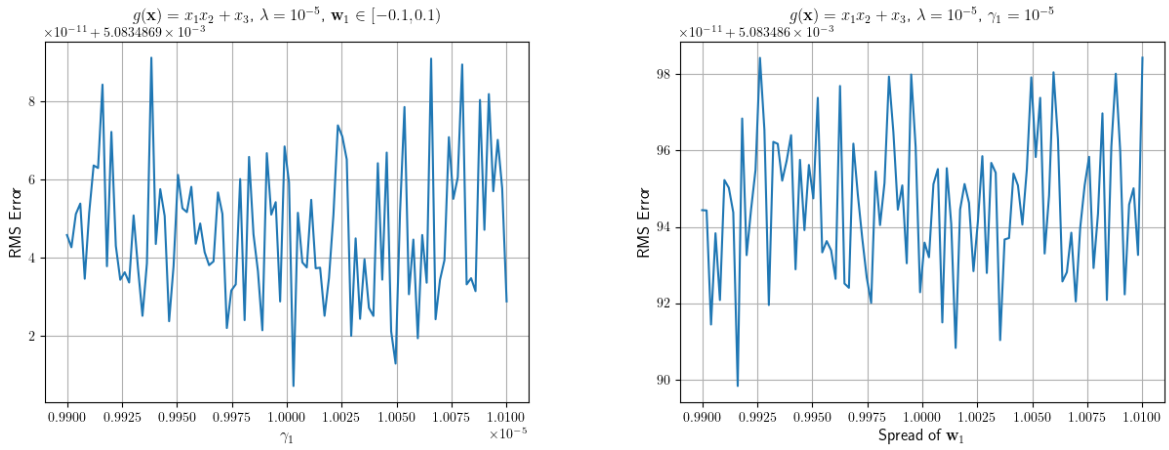

Figure 4: Training RMS error with respect to different γ_1 and $\mathbf{w}_1$.

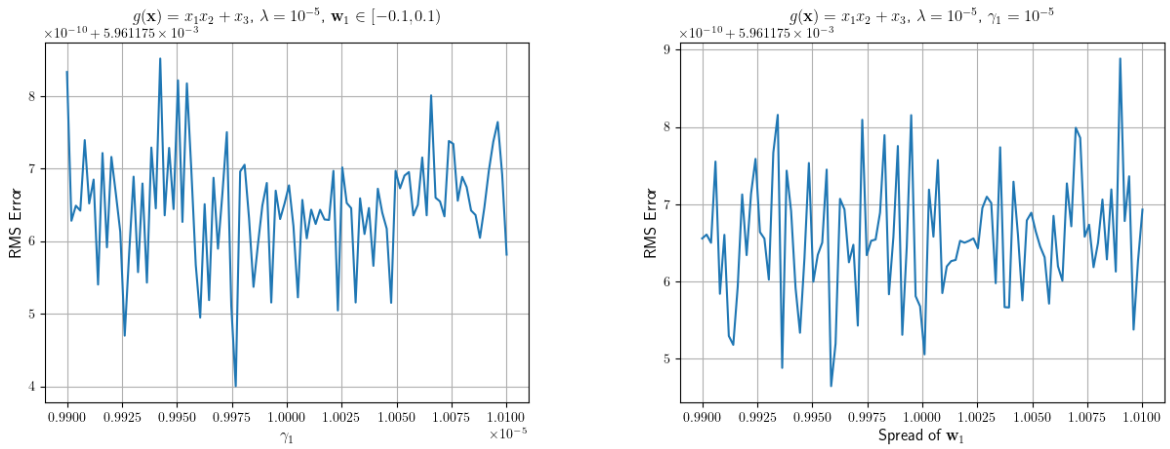

Figure 5: Testing RMS error with respect to different γ_1 and $\mathbf{w}_1$.

It is difficult to say if changing γ_1 or $\mathbf{w}_1$ has much of an effect on the RMS error. Perhaps a large range of γ_1 and $\mathbf{w}_1$ would need to be tested to see if there is a trend.

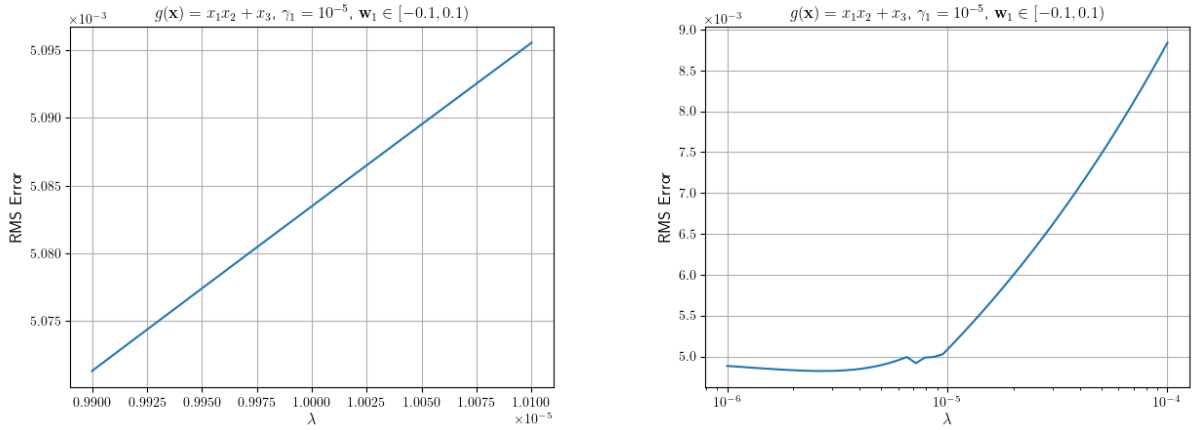

Figure 6: Training RMS error with respect to different λ.

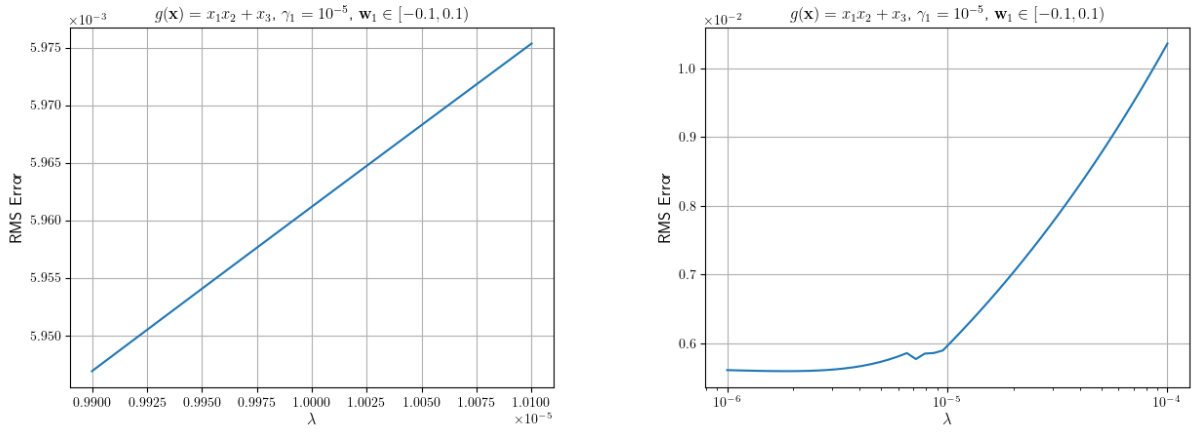

Figure 7: Testing RMS error with respect to different λ.

Figures 6 and 7 show that decreasing λ actually improves the RMS error for both the training and testing data set.

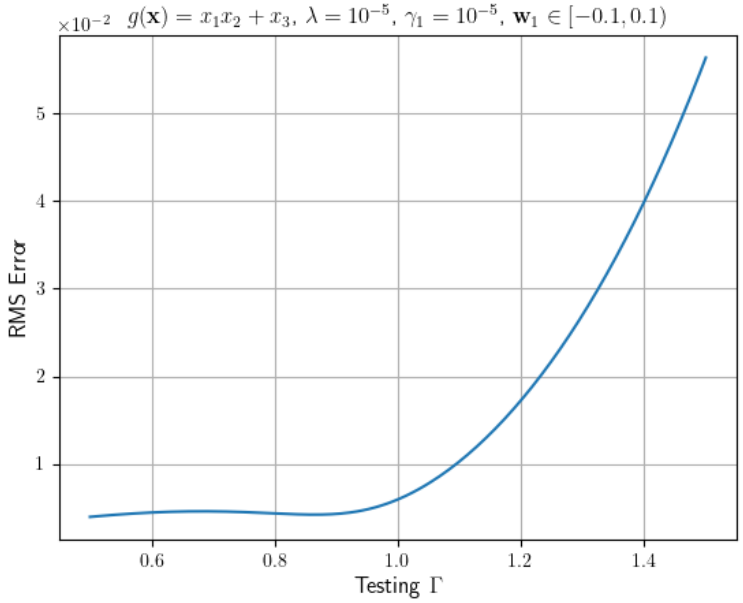

Figure 8: Testing RMS error with respect to different Γ for testing data.

Figure 8 shows that when the range of the testing data is extended beyond the range of the data that the network was trained on, the RMS error increases.

6 Results with noisy training data for $g(\mathbf{x}) = x_1x_2 + x_3$

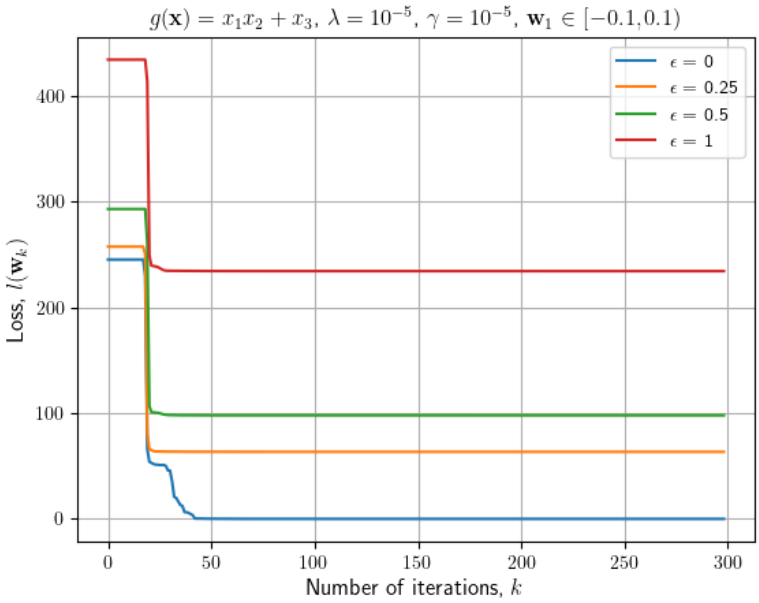

Figure 9: Training loss, $l(\mathbf{w}_k)$ with respect to the number of training iterations, k.

Obviously, increasing the noise in the training data will make the neural network perform much worse. It would be interesting to know if adjusting the initialization parameters in a particular way would make the network more robust to increased noise levels.

6.1 Final loss versus varied initializations

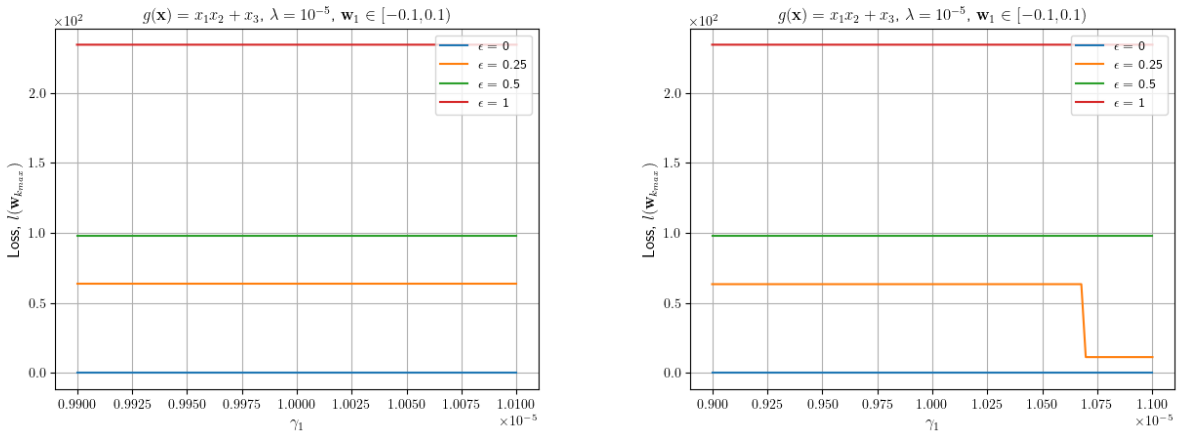

Figure 10: Final training loss with respect to different γ_1. Left plot is more zoomed in and higher resolution than the right plot.

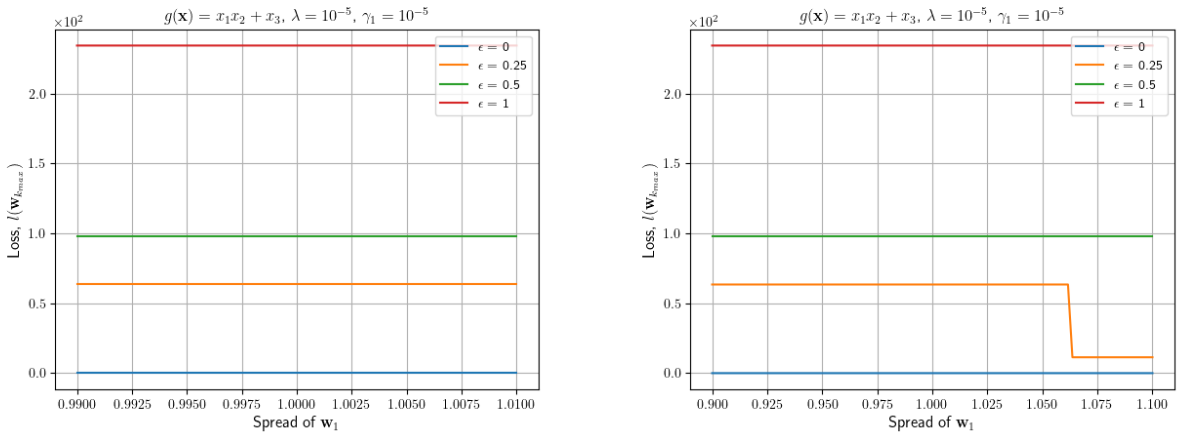

Figure 11: Final training loss with respect to different $\mathbf{w}_1$. Left plot is more zoomed in and higher resolution than the right plot.

Based on Figures 10 and 11, it seems that changing γ_1 and the spread of $\mathbf{w}_1$ doesn't help much with improving the loss when there is noise.

6.2 Final RMS error versus varied initializations

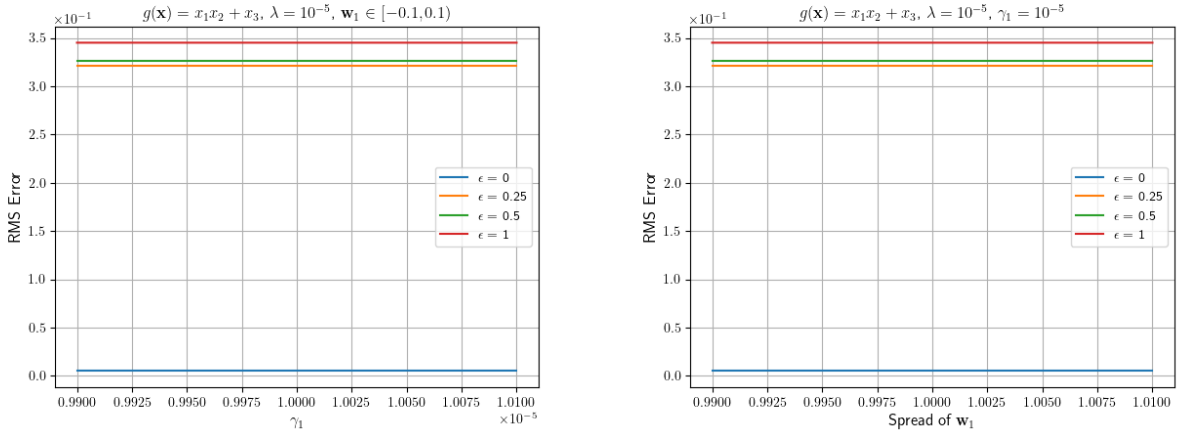

Figure 12: Training RMS error with respect to different γ_1 and $\mathbf{w}_1$.

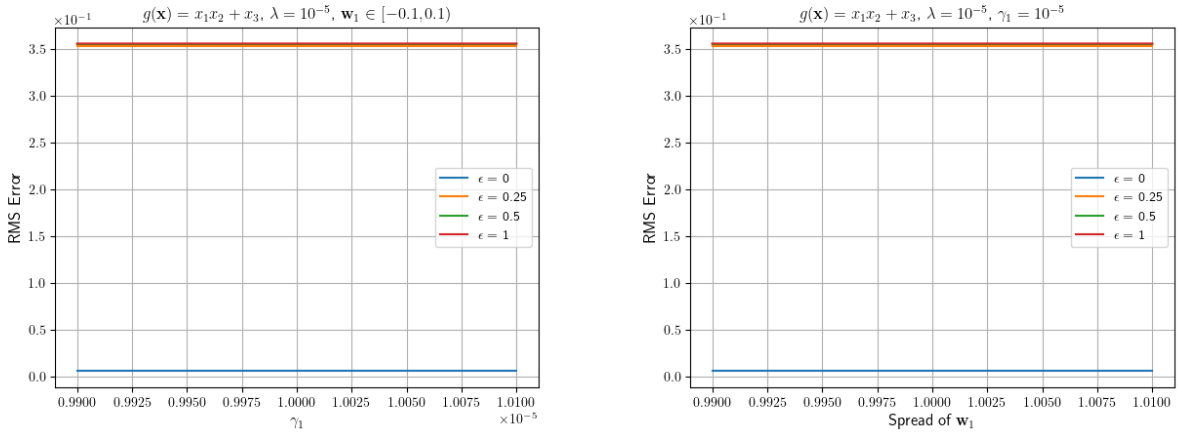

Figure 13: Testing RMS error with respect to different γ_1 and $\mathbf{w}_1$.

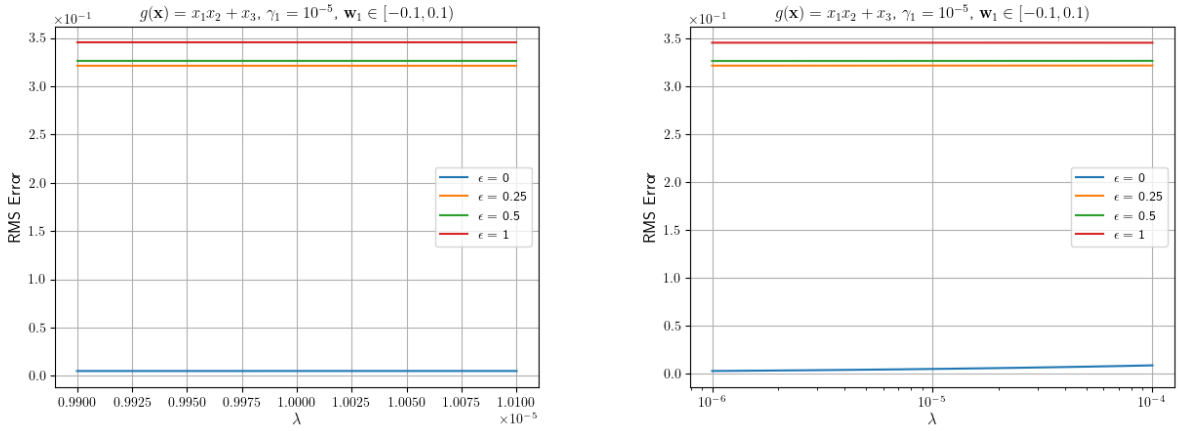

Figure 14: Training RMS error with respect to different λ

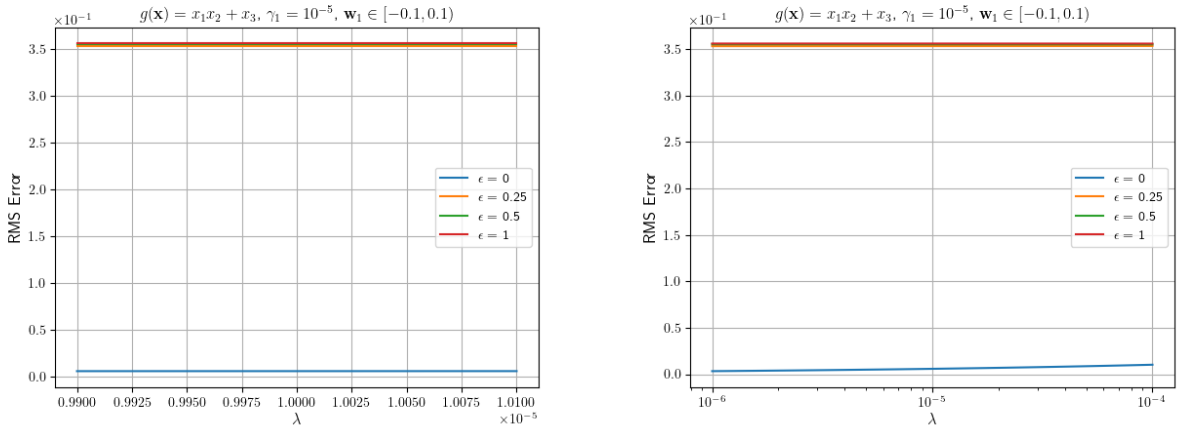

Figure 15: Testing RMS error with respect to different λ

Based on Figures 12 to 15, it seems that changing γ_1, $\mathbf{w}_1$, and λ doesn't help much with improving the loss when there is noise.

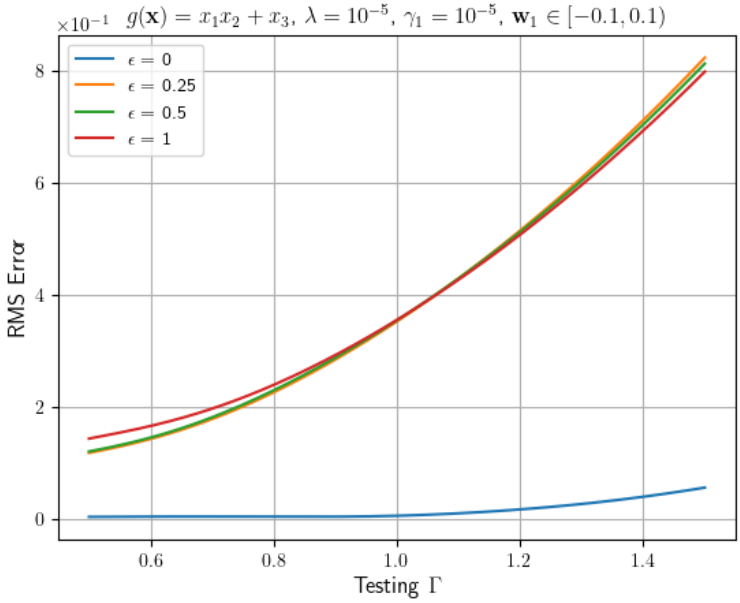

Figure 16: Testing RMS error with respect to different Γ for testing data.

7 Results for $g(\mathbf{x}) = 69\sin(x_1) + x_2x_3$

To produce the results in this section, all that needed to be done was define a new function for $g(\mathbf{x})$ and input it into the program. The values for the stopping criterion needed to be modified however to get quicker run times. In general, the neural network is worse at generalizing for $g(\mathbf{x}) = 69\sin(x_1) + x_2x_3$ so the bar needs to stopping criterion needs to be loosened up a little.

The results shown below are would bring us to similar conclusions as before. Changing $\mathbf{w}_1$ and γ_1 doesn't

have much effect on the final training loss or RMS error.

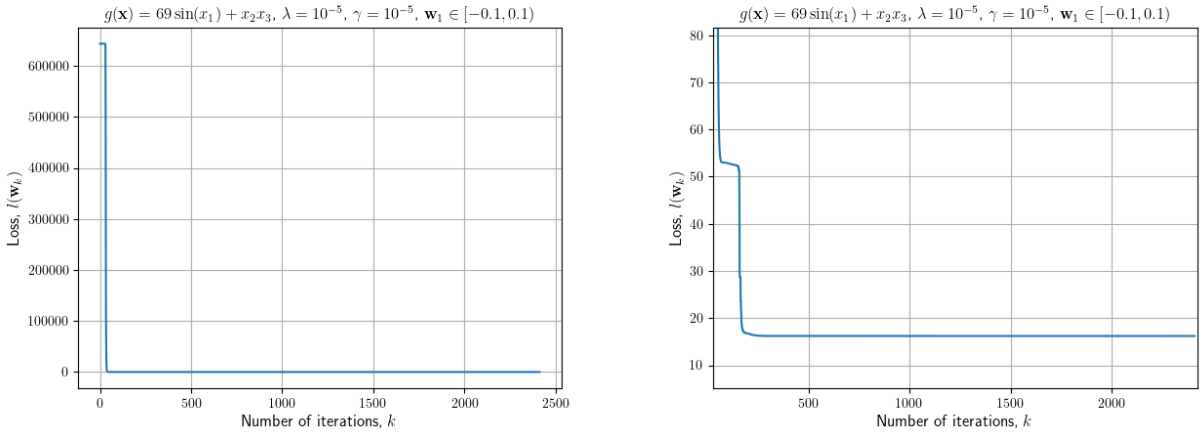

Figure 17: Training loss, $l(\mathbf{w}_k)$ with respect to the number of training iterations, k.

7.1 Final loss versus varied initializations

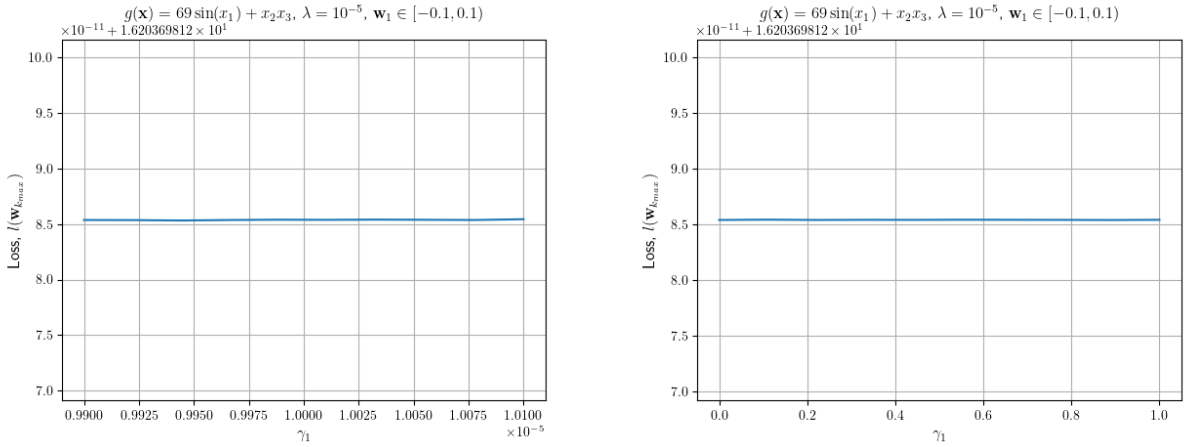

Figure 18: Final training loss with respect to different γ_1. Left plot is more zoomed in and higher resolution than the right plot.

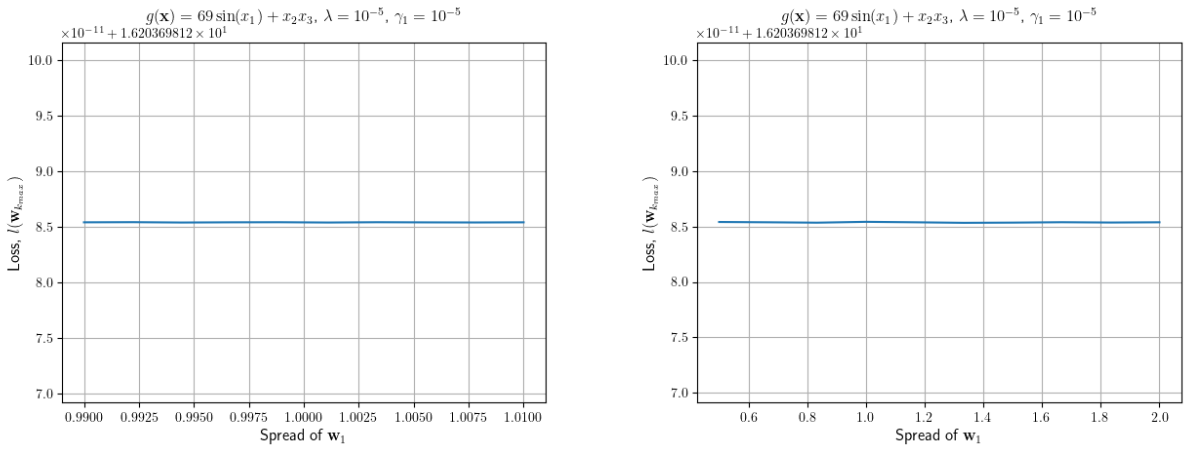

Figure 19: Final training loss with respect to different $\mathbf{w}_1$. Left plot is more zoomed in and higher resolution than the right plot.

7.2 Final RMS error versus varied initializations

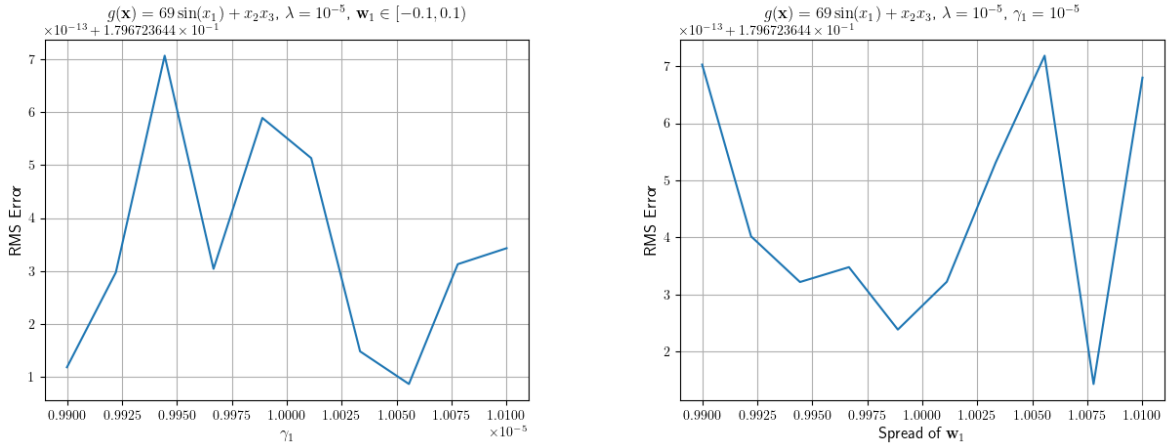

Figure 20: Training RMS error with respect to different γ_1 and $\mathbf{w}_1$.

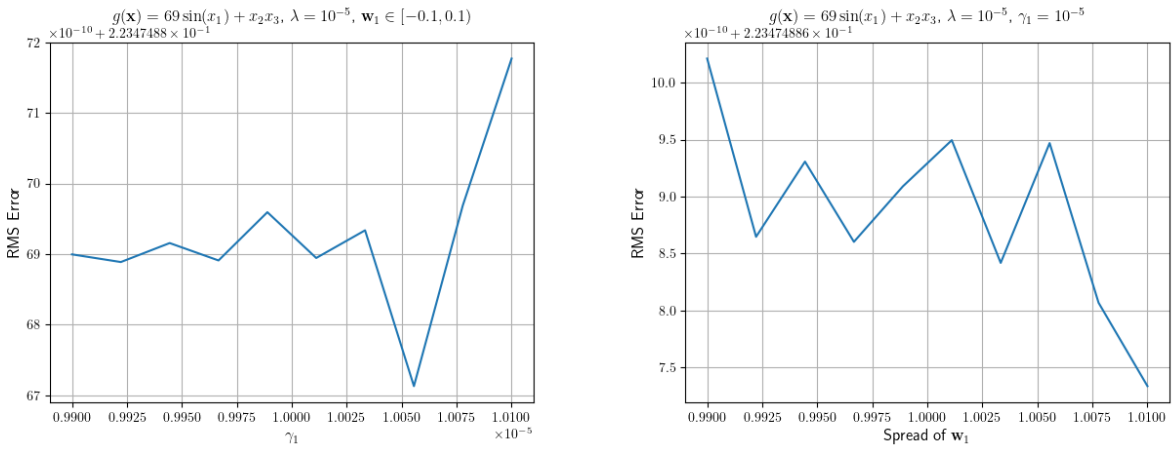

Figure 21: Testing RMS error with respect to different γ_1 and $\mathbf{w}_1$.

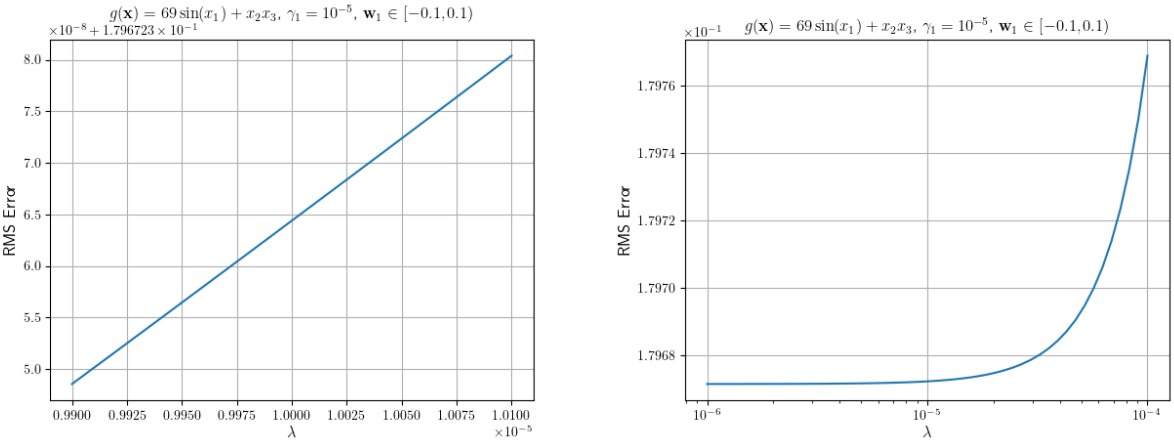

Figure 22: Training RMS error with respect to different λ.

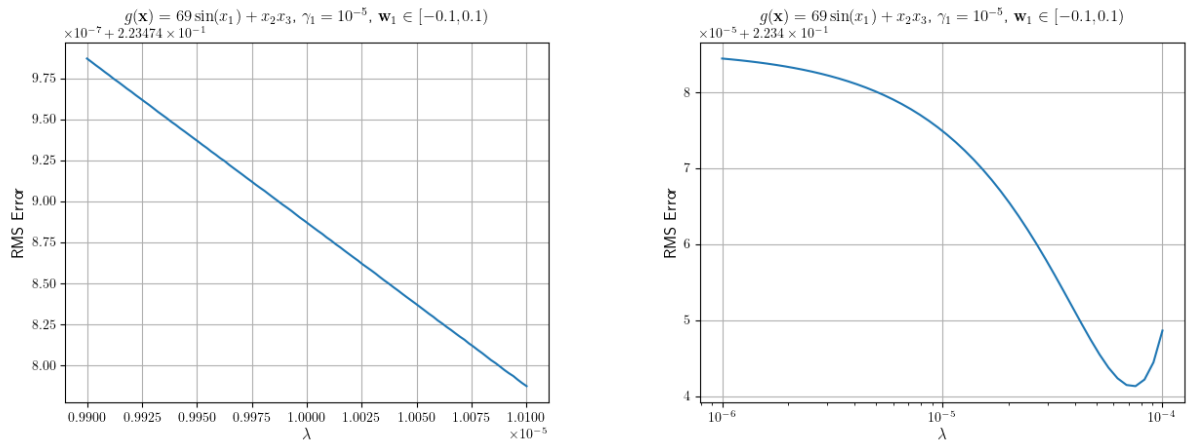

Figure 23: Testing RMS error with respect to different λ.

Interestingly, Figures 22 and 23 show the opposite results. Increasing λ increases RMS error for the training data, but decreases it for the testing data.

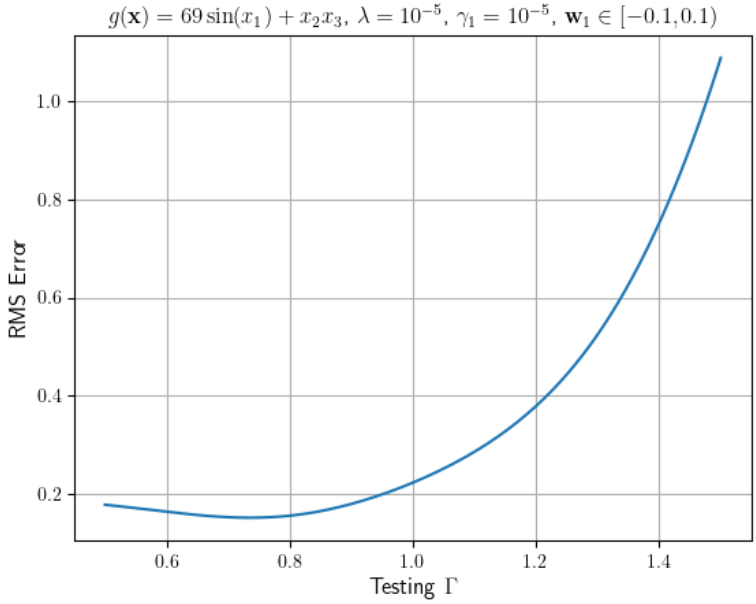

Figure 24: Testing RMS error with respect to different Γ for testing data.

8 Source code

```
##############################################
#          ECE 174 MINI PROJECT 2
##############################################
# AUTHOR: Conner Hsu
# PID: A16665092
```

```
##########################################
import os.path
from pathlib import Path
import sys

import numpy as np
import math

import matplotlib.pyplot as plt
import time

def save_np(path_name, arr):
    if '/' not in path_name:
        np.save(path_name, arr)
        return
    last_slash = path_name.rindex('/')
    path = path_name[0:last_slash+1]
    name = path_name[last_slash:]
    if not os.path.exists(path):
        os.makedirs(path)
    np.save(path_name, arr)
def list_latex(list):
    mean = np.average(list)
    result = ''
    for n in list:
        if n >= 1000:
            result += '{:.2e}'.format(n)
        else:
            result += '{:.2f}'.format(n)
        result += ' & '
    if mean >= 1000:
        result += '{:.2e}'.format(mean)
    else:
        result += '{:.2f}'.format(mean)
    return result  + ' \\\\'
def precision_truncate(list, order=1):
    list2 = (np.array(list)*order).astype('int64').astype('float64')/order
    end = np.where(list2==list2[-1])[0][0]
    return list[:end]

# Creates N sample input and output data points for g where g maps from R
    ^3 to R
def get_data(g, N, name, range=1, noise=0, new=False, save=True):
    filename = name + str(N) + '.npy'

    if not new and Path(filename).is_file():
        dataX = np.load(filename)
        print('Loaded', filename)
    else:
        dataX = np.random.uniform(-1, 1, size=(N, 3))
        if save:
            save_np(filename, dataX)
    dataX = dataX*range

```

```
    dataY = np.apply_along_axis(g, 1, dataX).T
    if noise != 0:
        filename_noise = name + '_noise' + str(N) + '.npy'
        if not new and Path(filename_noise).is_file():
            dataNoise = np.load(filename_noise)
        else:
            dataNoise = np.random.uniform(-1, 1, size=dataY.shape)
            if save:
                save_np(filename_noise, dataNoise)
        dataY += dataNoise*noise

    return dataX, dataY

class Trainer:
    phi = lambda x: np.tanh(x)
    phidot = lambda x: 1-np.square(np.tanh(x))
    def add_bias(A):
        return np.append(A, np.ones((A.shape[0], 1)), axis=1)
    def get_rms_error(w, dataX, dataY):
        N = dataX.shape[0]
        return math.sqrt(1/N*np.linalg.norm(Trainer.f(w, dataX)-dataY)**2)
    def f(w, dataX):
        x = Trainer.add_bias(dataX)
        temp = np.array([w[1:5], w[6:10], w[11:15]])
        phiInputs = np.matmul(x, temp.T)
        phiOutputs = Trainer.phi(phiInputs)

        phiOutputs = np.append(phiOutputs, np.ones((phiOutputs.shape[0],1)
            ), axis=1)
        output = np.matmul(phiOutputs, np.array([w[0], w[5], w[10], w
            [15]]))
        return output

    def __init__(self, w1, trainX, trainY, lambda0=1e-5, gamma1=1e-5):
        self.trainX = trainX
        self.trainY = trainY
        self.lam = lambda0
        self.w1 = w1
        self.gamma1 = gamma1

    def r(self, w):
        return Trainer.f(w, self.trainX) - self.trainY
    def Dr(self, w):
        x = Trainer.add_bias(self.trainX)
        temp = np.array([w[1:5], w[6:10], w[11:15]])
        phiInputs = np.matmul(x, temp.T)

        phiRows = Trainer.phi(phiInputs)

        phidotOutputs = Trainer.phidot(phiInputs)
        temp = np.diag([w[0], w[5], w[10]])
        temp = np.matmul(phidotOutputs, temp)
        temp = np.repeat(temp, 4, axis=1)
        phidotRows = np.multiply(temp, np.tile(x, (1, 3)))
```

```
111
112         result = (phiRows[:,0:1], phidotRows[:,:4],
113                   phiRows[:,1:2], phidotRows[:,4:8],
114                   phiRows[:,2:3], phidotRows[:,8:],
115                   np.ones((self.trainX.shape[0], 1)))
116
117         return np.concatenate(result, axis=1)
118     def Dw(self, w):
119         return np.identity(w.shape[0])
120     def h(self, w):
121         return np.concatenate((self.r(w), math.sqrt(self.lam)*w))
122     def Dh(self, w):
123         return np.concatenate((self.Dr(w), math.sqrt(self.lam)*self.Dw(w))
                )
124     def l(self, w):
125         return np.linalg.norm(self.h(w))**2
126     def get_next_iterate(self, w, gamma):
127         Dh = self.Dh(w)
128
129         b = np.matmul(Dh, w) - self.h(w)
130         b = np.concatenate((b, math.sqrt(gamma)*w))
131
132         A = np.concatenate((Dh, math.sqrt(gamma)*np.identity(16)))
133         pinvA = np.matmul(np.linalg.inv(np.matmul(A.T, A)), A.T)
134
135         return np.matmul(pinvA, b)
136     def get_optimality(self, w):
137         optimalityCondResidual = np.linalg.norm(2*np.matmul(self.Dr(w).T,
                self.r(w)) + 2*self.lam*self.w)
138         return optimalityCondResidual
139
140     def print_init(self):
141         print('='*110)
142         print('lambda=' + '{:.2e}'.format(self.lam),
143               '\t gamma1=' + '{:.2e}'.format(self.gamma1),
144               '\t w1=', str(self.w1[:2])[:-1], '...', str(self.w1[-2:])
                    [1:])
145         print('='*110)
146
147     def train(self, g, display=False):
148         wk = self.w1; gammak = self.gamma1
149
150         losses = []; loss = self.l(self.w1)
151         rms_error = Trainer.get_rms_error(wk, self.trainX, self.trainY)
152
153         stagnate = 0; k = 1; start_time = time.time()
154         while stopping_criterion(loss, rms_error, stagnate, time.time()-
                start_time, k):
155             losses.append(loss)
156
157             wk_next = self.get_next_iterate(wk, gammak)
158             loss_next = self.l(wk_next)
159
160             if loss_next < loss:
```

```
                wk = wk_next
                loss = loss_next
                gammak = 0.8*gammak
            else:
                gammak = 2*gammak
                stagnate += 1

            k+=1
            rms_error = Trainer.get_rms_error(wk, self.trainX, self.trainY
                )
            if display:
                print(k, '\t', '{:.2f}'.format(time.time()-start_time),
                        '\t Loss:', '{:.8f}'.format(loss),
                        '\t RMS Error:', '{:.8f}'.format(rms_error),
                        #'\t Gamma:', '{:.2e}'.format(gammak),
                        end=' '*15+'\r' )

        if display:
            print()

        return wk, losses

# 3a, make it plot multiple lines for multiple noises
def plot_loss_wrt_k(g, N, w1, noise_levels=[0]):
    plt.rcParams['text.usetex'] = True
    fig, ax = plt.subplots(1)

    for noise in noise_levels:
        trainX, trainY = get_data(g, N, 'train', noise=noise)
        trainer = Trainer(w1, trainX, trainY)
        wk, losses = trainer.train(g, display=True)

        line, = ax.plot(losses)
        if noise_levels != [0]:
            line.set_label('$\epsilon=$ ' + str(noise))

    ax.set_title(gname + ', $\lambda=10^{-5}$, $\gamma=10^{-5}$, $\mathbf{
        w}_1\in[-0.1,0.1)$')
    if noise_levels != [0]:
        ax.legend()
    ax.set_xlabel('Number of iterations, $k$', fontsize=12)
    ax.set_ylabel('Loss, $l(\mathbf{w}_k)$', fontsize=12)
    plt.grid()
    plt.show()


# 3a, make it plot multiple lines for multiple noises
def plot_loss_wrt_gamma1(g, N, w1, noise_levels=[0]):
    plt.rcParams['text.usetex'] = True
    fig, ax = plt.subplots(1)

    delta = 1e-7
    gamma1s = np.linspace(1e-4, 1, 10)

```

```
    for noise in noise_levels:
        trainX, trainY = get_data(g, N, 'train', noise=noise)
        trainer = Trainer(w1, trainX, trainY)
        losses = []
        for gamma1 in gamma1s:
            trainer.gamma1 = gamma1
            unused, losses_i = trainer.train(g)
            losses.append(losses_i[-1])

        line, = ax.plot(gamma1s, losses)
        if noise_levels != [0]:
            line.set_label('$\epsilon=$ ' + str(noise))

    ax.set_title(gname + ', $\lambda=10^{-5}$, $\mathbf{w}_1\in[-0.1,0.1)$
        ')
    if noise_levels != [0]:
        ax.legend()
    ax.set_xlabel('$\gamma_1$', fontsize=12)
    ax.set_ylabel('Loss, $l(\mathbf{w}_{k_{max}})$', fontsize=12)
    plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
    plt.grid()
    plt.show()
def plot_loss_wrt_w1(g, N, w1, noise_levels=[0]):
    plt.rcParams['text.usetex'] = True
    fig, ax = plt.subplots(1)

    delta = 0.01
    scales = np.linspace(0.5, 2, 10)

    for noise in noise_levels:
        trainX, trainY = get_data(g, N, 'train', noise=noise)
        trainer = Trainer(w1, trainX, trainY)
        losses = []
        for scale in scales:
            trainer.w1 = w1*scale
            unused, losses_i = trainer.train(g)
            losses.append(losses_i[-1])


        line, = ax.plot(scales, losses)
        if noise_levels != [0]:
            line.set_label('$\epsilon=$ ' + str(noise))

    ax.set_title(gname + ', $\lambda=10^{-5}$, $\gamma_1=10^{-5}$')
    if noise_levels != [0]:
        ax.legend()
    plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
    ax.set_xlabel('Spread of $\mathbf{w}_1$', fontsize=12)
    ax.set_ylabel('Loss, $l(\mathbf{w}_{k_{max}})$', fontsize=12)
    plt.grid()
    plt.show()


# 3b, different initializations of LM
```

```
266 def plot_error_wrt_gamma1(g, N, w1, NT, type, noise_levels=[0]):
267     plt.rcParams['text.usetex'] = True
268     fig, ax = plt.subplots(1)
269 
270     delta = 1e-7
271     gamma1s = np.linspace(1e-5-delta, 1e-5+delta, 10)
272     dataX, dataY = get_data(g, NT, type)
273 
274     for noise in noise_levels:
275         trainX, trainY = get_data(g, N, 'train', noise=noise)
276         trainer = Trainer(w1, trainX, trainY)
277         errors = []
278         for gamma1 in gamma1s:
279             trainer.gamma1 = gamma1
280             wk_final, unused = trainer.train(g)
281             errors.append(Trainer.get_rms_error(wk_final, dataX, dataY))
282 
283         line, = ax.plot(gamma1s, errors)
284         if noise_levels != [0]:
285             line.set_label('$\epsilon=$ ' + str(noise))
286 
287     ax.set_title(gname + ', $\lambda=10^{-5}$, $\mathbf{w}_1\in[-0.1,0.1)$
        ')
288     if noise_levels != [0]:
289         ax.legend()
290     ax.set_xlabel('$\gamma_1$', fontsize=12)
291     ax.set_ylabel('RMS Error', fontsize=12)
292     plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
293     plt.grid()
294     plt.show()
295 def plot_error_wrt_w1(g, N, w1, NT, type, noise_levels=[0]):
296     plt.rcParams['text.usetex'] = True
297     fig, ax = plt.subplots(1)
298 
299     delta = 0.01
300     scales = np.linspace(1-delta, 1+delta, 10)
301     dataX, dataY = get_data(g, NT, type)
302 
303     for noise in noise_levels:
304         trainX, trainY = get_data(g, N, 'train', noise=noise)
305         trainer = Trainer(w1, trainX, trainY)
306         errors = []
307         for scale in scales:
308             trainer.w1 = w1*scale
309             wk_final, unused = trainer.train(g)
310             errors.append(Trainer.get_rms_error(wk_final, dataX, dataY))
311 
312         line, = ax.plot(scales, errors)
313         if noise_levels != [0]:
314             line.set_label('$\epsilon=$ ' + str(noise))
315 
316     ax.set_title(gname + ', $\lambda=10^{-5}$, $\gamma_1=10^{-5}$')
317     if noise_levels != [0]:
318         ax.legend()
```

```
    plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
    ax.set_xlabel('Spread of $\mathbf{w}_1$', fontsize=12)
    ax.set_ylabel('RMS Error', fontsize=12)
    plt.grid()
    plt.show()

# 3b, different lambda and Gamma
def plot_error_wrt_lambda(g, N, w1, NT, type, noise_levels=[0]):
    plt.rcParams['text.usetex'] = True
    fig, ax = plt.subplots(1)

    delta = 1e-7
    lambdas = np.logspace(-6, -4, 50)
    #lambdas = np.linspace(1e-5-delta, 1e-5+delta, 100)
    dataX, dataY = get_data(g, NT, type)

    for noise in noise_levels:
        trainX, trainY = get_data(g, N, 'train', noise=noise)
        trainer = Trainer(w1, trainX, trainY)
        errors = []
        for lambda0 in lambdas:
            trainer.lam = lambda0
            wk_final, unused = trainer.train(g)
            errors.append(Trainer.get_rms_error(wk_final, dataX, dataY))

        line, = ax.plot(lambdas, errors)
        if noise_levels != [0]:
            line.set_label('$\epsilon=$ ' + str(noise))

    ax.set_title(gname + ', $\gamma_1=10^{-5}$, $\mathbf{w}_1\in[-0.1,0.1)
       $')
    if noise_levels != [0]:
        ax.legend()
    ax.set_xlabel('$\lambda$', fontsize=12)
    ax.set_ylabel('RMS Error', fontsize=12)
    plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
    #plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))
    ax.set_xscale('log')

    plt.grid()
    plt.show()
def plot_error_wrt_Gamma(g, N, w1, NT, type, noise_levels=[0]):
    plt.rcParams['text.usetex'] = True
    fig, ax = plt.subplots(1)

    delta = 0.5
    scales = np.linspace(1-delta, 1+delta, 200)
    dataX, dataY = get_data(g, NT, type)

    for noise in noise_levels:
        trainX, trainY = get_data(g, N, 'train', noise=noise)
        trainer = Trainer(w1, trainX, trainY)
        wk_final, unused = trainer.train(g)

```

```
        errors = []
        for scale in scales:
            new_dataX = dataX*scale
            errors.append(Trainer.get_rms_error(wk_final, new_dataX, np.
                apply_along_axis(g, 1, new_dataX).T))

        line, = ax.plot(scales, errors)
        if noise_levels != [0]:
            line.set_label('$\epsilon=$ ' + str(noise))

    ax.set_title(gname + ', $\lambda=10^{-5}$, $\gamma_1=10^{-5}$, $\
        mathbf{w}_1\in[-0.1,0.1)$')
    if noise_levels != [0]:
        ax.legend()
    plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
    ax.set_xlabel('Testing $\Gamma$', fontsize=12)
    ax.set_ylabel('RMS Error', fontsize=12)
    plt.grid()
    plt.show()

def ThreeA(g, N, w1, noise_levels=[0]):
    #plot_loss_wrt_k(g, N, w1, noise_levels=noise_levels)
    plot_loss_wrt_gamma1(g, N, w1, noise_levels=noise_levels)
    plot_loss_wrt_w1(g, N, w1, noise_levels=noise_levels)

def ThreeB(g, N, w1, NT, noise_levels=[0]):
    plot_error_wrt_gamma1(g, N, w1, N, 'train', noise_levels=noise_levels)
    plot_error_wrt_w1(g, N, w1, N, 'train', noise_levels=noise_levels)

    plot_error_wrt_gamma1(g, N, w1, NT, 'test', noise_levels=noise_levels)
    plot_error_wrt_w1(g, N, w1, NT, 'test', noise_levels=noise_levels)

    #plot_error_wrt_lambda(g, N, w1, N, 'train', noise_levels=noise_levels
        )
    #plot_error_wrt_lambda(g, N, w1, NT, 'test', noise_levels=noise_levels
        )

    #plot_error_wrt_Gamma(g, N, w1, NT, 'test', noise_levels=noise_levels)

def train_alot(g, N, noise=0):
    trainX, trainY = get_data(g, N, 'train', noise=noise)
    w1 = np.random.uniform(-0.1, 0.1, size=16)
    trainer = Trainer(w1, trainX, trainY)
    trainer.print_init()
    for i in range(20):
        unused, errors = trainer.train(g, display=True)
        trainer.w1 = np.random.uniform(-0.1, 0.1, size=16)
def stopping_criterion(loss, rms_error, stagnate, time, k):
    return rms_error > 0.006 and stagnate < 1200 # used for 3a
    #return rms_error > 0.17 and stagnate < 1000 # used for 3b
    #return k < 1000 # used for 3e

g1name = '$g(\mathbf{x})=x_1x_2+x_3$'
g2name = '$g(\mathbf{x})=69\sin(x_1)+x_2x_3$'
```

```
gname = g2name
def main():
    N=500
    NT=100
    g1 = lambda x: x[0]*x[1]+x[2];
    g2 = lambda x: 69*math.sin(x[0])+x[1]*x[2];

    #train_alot(g1, N)
    #w1=np.random.uniform(-0.1,0.1,size=16)

    # initial weights for g1
    w1g1 = np.array([-0.03907819,  0.06158748, -0.08008263, -0.06117632,
       -0.00110027,  0.06879582,
                     -0.00660737, -0.08597136, -0.06461364, -0.03900993,
                        0.07414459,  0.07428109,
                     0.06751018,  0.02483472,  0.07676263, -0.07073906])

    w1g2 = np.array([-0.03236776, -0.02194876, -0.00409352,  0.07917061,
       -0.07397286, -0.05966028,
                     0.03283233, -0.01403717,  0.09412561,  0.0943238,
                        -0.04830366,  0.05694761,
                     0.02113755, -0.07613354, -0.09833506, -0.02382732])
    # 3a
    ThreeA(g1, N, w1g1)

    # 3b
    #ThreeB(g1, N, w1g1, NT)

    # 3c
    #ThreeA(g2, N, w1g2)
    #ThreeB(g2, N, w1g2, NT)

    # 3e
    #noise_levels = [0, 0.25, 0.5, 1]
    #ThreeA(g1, N, w1g1, noise_levels=noise_levels)
    #ThreeB(g1, N, w1g1, NT, noise_levels=noise_levels)

if __name__ == '__main__':
    main()
```