ECE174-MiniProject2

Jay Paek

December 2022

1 Computing the Gradient

Consider the following function

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

where $w_1, w_2, ..., w_{16}, x_1, x_2, x_3 \in \mathbb{R}$ and

$$\phi(x) = \mathbf{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Let $\phi(a, b, c, d) = \phi(ax_1 + bx_2 + cx_3 + d)$

$$\begin{aligned} f_{\mathbf{w}}(\mathbf{x}) &= w_1\phi(w_2, w_3, w_4, w_5) \\ &+ w_6\phi(w_7, w_8x, w_9, w_{10}) \\ &+ w_{11}\phi(w_{12}, w_{13}, w_{14}, w_{15}) + w_{16} \end{aligned}$$

Also let the first derivative of ϕ be $\phi^\dagger$
Define the gradient vector:

$$\nabla_{\mathbf{w}} = \begin{bmatrix} \frac{\partial}{\partial w_1} & \frac{\partial}{\partial w_2} & ... & \frac{\partial}{\partial w_{16}} \end{bmatrix}$$

Find $\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x})$

$$\begin{aligned} \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}) &= \begin{bmatrix} \frac{\partial}{\partial w_1} & \frac{\partial}{\partial w_2} & ... & \frac{\partial}{\partial w_{16}} \end{bmatrix} f_{\mathbf{w}}(\mathbf{x}) \\ &= \begin{bmatrix} \frac{\partial f_{\mathbf{w}}(\mathbf{x})}{\partial w_1} & \frac{\partial f_{\mathbf{w}}(\mathbf{x})}{\partial w_2} & ... & \frac{\partial f_{\mathbf{w}}(\mathbf{x})}{\partial w_{16}} \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} \phi(w_2, w_3, w_4, w_5) \\ w_1 x_1 \phi^\dagger(w_2, w_3, w_4, w_5) \\ w_1 x_2 \phi^\dagger(w_2, w_3, w_4, w_5) \\ w_1 x_3 \phi^\dagger(w_2, w_3, w_4, w_5) \\ w_1 \phi^\dagger(w_2, w_3, w_4, w_5) \\ \phi(w_7, w_8, w_9, w_{10}) \\ w_6 x_1 \phi^\dagger(w_7, w_8, w_9, w_{10}) \\ w_6 x_2 \phi^\dagger(w_7, w_8, w_9, w_{10}) \\ w_6 x_3 \phi^\dagger(w_7, w_8, w_9, w_{10}) \\ w_6 \phi^\dagger(w_7, w_8, w_9, w_{10}) \\ \phi(w_{12}, w_{13}, w_{14}, w_{15}) \\ w_{11} x_1 \phi^\dagger(w_{12}, w_{13}, w_{14}, w_{15}) \\ w_{11} x_2 \phi^\dagger(w_{12}, w_{13}, w_{14}, w_{15}) \\ w_{11} x_3 \phi^\dagger(w_{12}, w_{13}, w_{14}, w_{15}) \\ w_{11} \phi^\dagger(w_{12}, w_{13}, w_{14}, w_{15}) \\ 1 \end{bmatrix}^T$$

2 Computing the Derivative Matrix

Define a non-linear function: $y^{(n)} = \mathbf{g}(\mathbf{x}^{(n)}), n = 1, 2, ..., N$
such that $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(N)} \in \mathbb{R}^3, y^{(1)}, y^{(2)}, ..., y^{(N)} \in \mathbb{R}$
Define $r_n(\mathbf{w}) = f_{\mathbf{w}}(\mathbf{x}^{(n)}) - y^{(n)}$
Define $\mathbf{r(w)} = \begin{bmatrix} f_{\mathbf{w}}(\mathbf{x}^{(1)}) - y^{(1)} & f_{\mathbf{w}}(\mathbf{x}^{(2)}) - y^{(2)} & ... & f_{\mathbf{w}}(\mathbf{x}^{(n)}) - y^{(n)} \end{bmatrix}^T = \begin{bmatrix} r_1(\mathbf{w}) & r_2(\mathbf{w}) & ... & r_n(\mathbf{w}) \end{bmatrix}^T$
Define the sum of squared errors:

$$\sum_{n=1}^{N} (f_{\mathbf{w}}(\mathbf{x}^{(n)}) - y^{(n)})^2 = \sum_{n=1}^{N} r_n^2(\mathbf{w})$$

Find the gradient of the sum of square errors:

$$\nabla_{\mathbf{w}} \sum_{n=1}^{N} r_n^2(\mathbf{w})$$

$$= 2 \sum_{n=1}^{N} r_n(\mathbf{w}) \nabla_{\mathbf{w}} r_n(\mathbf{w})$$

$$= 2 \sum_{n=1}^{N} r_n(\mathbf{w}) \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(n)})$$

$$= 2(r_1(\mathbf{w}) \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(1)}) + r_2(\mathbf{w}) \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(2)}) + ... + r_n(\mathbf{w}) \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(n)}))$$

$$= 2 \begin{bmatrix} (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(1)}))^T & (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(2)}))^T & ... & (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(n)}))^T \end{bmatrix} \begin{bmatrix} r_1(\mathbf{w}) \\ r_2(\mathbf{w}) \\ ... \\ r_n(\mathbf{w}) \end{bmatrix}$$

$$= 2 \begin{bmatrix} (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(1)})) \\ (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(2)})) \\ ... \\ (\nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(n)})) \end{bmatrix}^T \begin{bmatrix} r_1(\mathbf{w}) \\ r_2(\mathbf{w}) \\ ... \\ r_n(\mathbf{w}) \end{bmatrix}$$

$$= 2\mathbf{D}\mathbf{r}(\mathbf{w})^T \mathbf{r}(\mathbf{w})$$

where $\mathbf{D}\mathbf{r}(\mathbf{w}) = \begin{bmatrix} \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(1)}) \\ \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(2)}) \\ ... \\ \nabla_{\mathbf{w}} f_{\mathbf{w}}(\mathbf{x}^{(n)}) \end{bmatrix}$

3 Levenburg-Marquadt Algorithm

To minimize the loss function, we will apply the Levenburg Marquadt Algorithm by minimizing the training loss function $l(\mathbf{w})$

$$l(\mathbf{w}) = \sum_{n=1}^{N} [(f_{\mathbf{w}}(\mathbf{x}^{(n)}) - y^{(n)})^2] - \lambda ||\mathbf{w}||_2^2 = ||\mathbf{r}(\mathbf{w})||_2^2 + ||\sqrt{\lambda}\mathbf{w}||_2^2$$

given $N = 500$ data points with the constraints on every $\mathbf{x}^{(n)}$ such that $\mathbf{max}([\mathbf{x}^{(n)}]_1, [\mathbf{x}^{(n)}]_2, [\mathbf{x}^{(n)}]_3) \leq 1$ for $n = 1, 2, ..., N$.
The stopping criterion used in the code is looking for the training loss to be minimized. for the heuristic algorithm depends on what kinds of patterns I am looking for.

- For generic "minimize error" objectives, the stopping criterion is for $l(\mathbf{w}) < 1$ and maximum steps to be 1000. I want to maximize iterations to decrease the loss function as much as possible.
- To test different starting values for the weights, the stopping criterion is set to $l(\mathbf{w}) < 25$ and maximum iterations at 25 or 50. Reasons are stated in the respective section.
- For testing training loss and seeing number of iterations for varied values of λ,
- For testing training loss under different values Γ and λ, I used the stopping criterion l($\mathbf{w}$) ¡ 25 with maximum iterations of 100. This is because

The main error metric to be used is the mean squared error defined as the following:

$$\frac{1}{N} \sum_{i=1}^{N} (y^{(i)})^2 - f_{\mathbf{w}}(\mathbf{x}^{(i)})$$

which is essentially the difference between the expected output and the predicted output, squared and summed over all data points, then divided by the number of data points.
Throughout this document, I will post code that generate the results in the graph. The code will ALWAYS precede the results. Some results will not have code, mainly due to the fact that the same code were used but with different inputs. All algorithms and loops created in this project are in the code. Set-up code

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
```

```
import math

# make e
e = np.exp(1)
e

# generate data points and yielded results for non-linear function
def generate_data_1(n, gamma):
    data = np.empty((0,3))
    out = np.empty((0,1))
    for i in range(n):
        x = np.random.uniform(-gamma, gamma, (3,))
        data = np.vstack((data,x))
        out = np.append(out,[x[0]*x[1]+x[2]])
    return data, out

# derivative of tanh
def d_tanh(x):
    return (4*e**(2*x))/((e**(2*x)+1)**2)
# characteristic approximation function, expected results function
def f(x, weights):
    return weights[0]*np.tanh(weights[1]*x[0] + weights[2]*x[1] + weights[3]*x[2] + weights[4]) + \
            weights[5]*np.tanh(weights[6]*x[0] + weights[7]*x[1] + weights[8]*x[2] + weights[9]) + \
            weights[10]*np.tanh(weights[11]*x[0] + weights[12]*x[1] + weights[13]*x[2] + weights[14]) +
# get vector of expected results given matrix of data points
def f_vector(data, weights):
    result = np.empty((0,))
    for x in data:
        result = np.concatenate((result, np.asarray([f(x, weights)])))
    return result
# calculate gradient
def g_tanh(x, weights):
    return np.asarray([
        np.tanh(weights[1]*x[0] + weights[2]*x[1] + weights[3]*x[2] + weights[4]),
        weights[0]*x[0] * d_tanh(weights[1]*x[0] + weights[2]*x[1] + weights[3]*x[2] + weights[4]),
        weights[0]*x[1] * d_tanh(weights[1]*x[0] + weights[2]*x[1] + weights[3]*x[2] + weights[4]),
        weights[0]*x[2] * d_tanh(weights[1]*x[0] + weights[2]*x[1] + weights[3]*x[2] + weights[4]),
        weights[0] * d_tanh(weights[1]*x[0] + weights[2]*x[1] + weights[3]*x[2] + weights[4]),
        np.tanh(weights[6]*x[0] + weights[7]*x[1] + weights[8]*x[2] + weights[9]),
        weights[5]*x[0] * d_tanh(weights[6]*x[0] + weights[7]*x[1] + weights[8]*x[2] + weights[9]),
        weights[5]*x[1] * d_tanh(weights[6]*x[0] + weights[7]*x[1] + weights[8]*x[2] + weights[9]),
        weights[5]*x[2] * d_tanh(weights[6]*x[0] + weights[7]*x[1] + weights[8]*x[2] + weights[9]),
        weights[5] * d_tanh(weights[6]*x[0] + weights[7]*x[1] + weights[8]*x[2] + weights[9]),
```

```
        np.tanh(weights[11]*x[0] + weights[12]*x[1] + weights[13]*x[2] + weights[14]),
        weights[10]*x[0] * d_tanh(weights[11]*x[0] + weights[12]*x[1] + weights[13]*x[2] + weights[14])
        weights[10]*x[1] * d_tanh(weights[11]*x[0] + weights[12]*x[1] + weights[13]*x[2] + weights[14])
        weights[10]*x[2] * d_tanh(weights[11]*x[0] + weights[12]*x[1] + weights[13]*x[2] + weights[14])
        weights[10] * d_tanh(weights[11]*x[0] + weights[12]*x[1] + weights[13]*x[2] + weights[14]),
        1
    ])
# calculate derivative matrix, or stacked gradients
def gradient_matrix(data, weights):
    jacobian = np.empty((0, weights.shape[0]))
    for x in data:
        jacobian = np.vstack((jacobian, g_tanh(x, weights)))
    return jacobian
#find weights
def find_weights(data, expected, lamb=0.00001, first_step=1, variation=1, weights=None, stop=1, max_ite
    # initializations
    if weights is None:
        weights=np.random.normal(0, variation, (16,))
    trust=0.8
    distrust = 2
    diff = np.sum((f_vector(data, weights) - expected)**2)
    err = diff+lamb*np.linalg.norm(weights)
    step = first_step
    steps = [step]
    errs = [err]
    # while np.linalg.norm(2*gradient.T@expected) < threshold:
    while (len(errs) == 0 or err > stop) and len(errs) < max_iter:
        try:
            # this part of the code will convert the program to a least squares problem
            # First find the gradient matrix
            jacobian = gradient_matrix(data, weights)

            #Initialize a column vector with the expected values with the given weights
            predicted = f_vector(data, weights)
            # Find the value we are trying to project
            y = predicted - expected - jacobian @ weights
            y = np.concatenate((y, np.zeros((weights.shape))), axis=0)
            y = np.concatenate((y, -1*math.sqrt(step)*weights), axis=0)

            # Create the range space by appending the trust value to the gradient matrix
            A = np.concatenate((jacobian, math.sqrt(lamb)*np.identity(jacobian.shape[1])), axis=0)
            A = np.concatenate((A, math.sqrt(step)*np.identity(jacobian.shape[1])), axis=0)
```

```
            # Solve least squares problem
            new_weights = np.linalg.pinv(A)@y
            # Calculate the error for newly create weights from solving least squares problem
            diff = np.sum((f_vector(data, new_weights) - expected)**2)
            new_err = diff+lamb*np.linalg.norm(new_weights)

            # print(err, end=' ')
            # print(new_err)
            # Check if the new error is less than the previous error, otherwise, increase lamba
            if new_err < err:
                weights = new_weights
                err = new_err
                step = trust*step
            else:
                step = distrust*step

            # add step and error values to the list for plotting
            steps.append(step)
            errs.append(err)
        except KeyboardInterrupt:
            return None, None, None
        except LinAlgError:
            return
    return errs, steps, weights
```

4 Training Model on Non-Linear Function $[x]_1[x]_2 + [x]_3$

4.1 Training Results

```
# generate data
data, expected = generate_data_1(500, 1)
# generate weights
errs, steps, weights = find_weights(data, expected, lamb=0.00001, max_iter=1000, stop=0.1)
```

The following tables show the change in training loss and trust value over the iterations when training the model. The stopping criterion for this objective is $l(\mathbf{w}) < 0.1$ with maximum iteration of 1000. This is because I am aiming for the most accurate model possible without spending too much time. Approximately 0.1 squared error across 500 data points means that there is an average of 0.002 average squared error per data point, which is pretty accurate. Since I am only training one model, I can spend as much time as I want.

I specifically selected this non-linear equation due to the fact that it was similar to the original function, yet different. Both equation had two variables multiplied to each other and added with another variable.

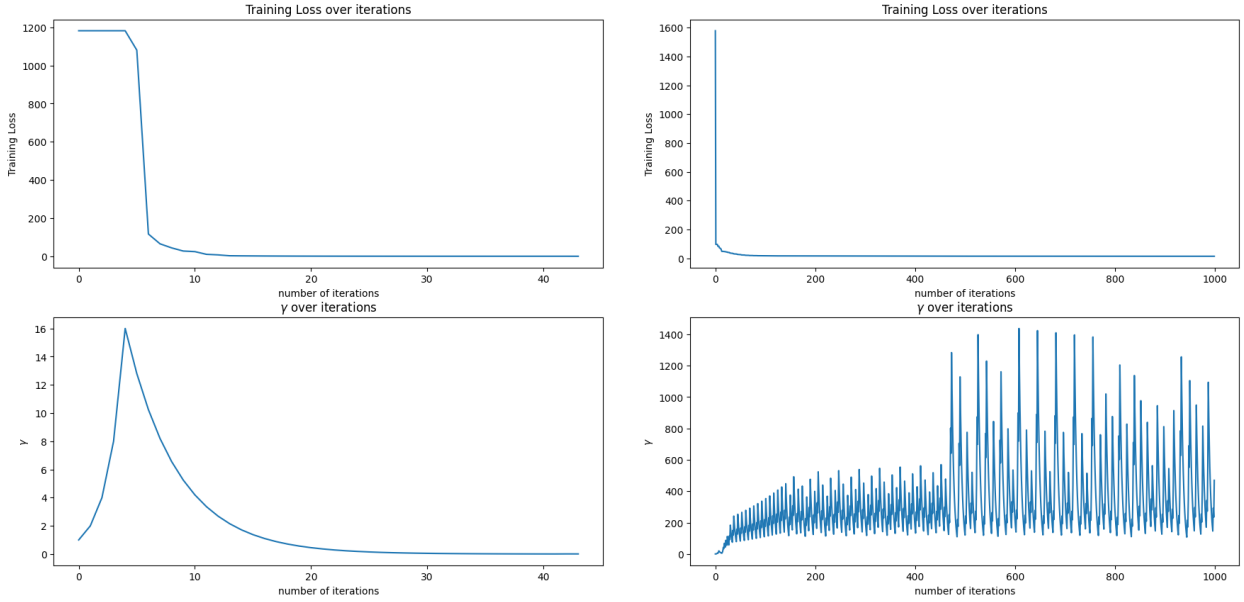

Figure 1: The graphs depict the change in training loss and gamma values over iterations for two instances of the algorithm

The graph above is an instance where the the algorithm unfortunately converged to a local minima. The final error for that specific instance was 13.9. The graph below shows a better instance where the final training loss hit below 0.1 within 50 iterations.

4.2 Training and Testing on Varied Initializations

```
# values to plot
all_losses = []
all_iters = []
```

```
all_errs = []
# create set of variations in different magnitudes
all_vals = [[a*(10**b) for a in range(20)] for b in range(-5,1)]
# iterate through different magnitudes
for i in all_vals:
    loss = []
    iters = []
    error = []
    # get variance value
    for j in i:
        # accumulate weights
        total_loss = []
        total_iters = []
        total_errs = []
        # make weights 25 times
        for k in range(25):
            # make weights
            errs, steps, weights = find_weights(data, expected, lamb=0.00001, stop=25, max_iter=50, var
            # add final error, loss, and iteration values
            total_loss.append(errs[-1])
            total_iters.append(len(errs))
            total_errs.append(np.sum((f_vector(data, weights) - expected)**2))
        # average all data values over 500 data points and 25 instances
        loss.append(sum(total_loss)/500/len(total_loss))
        iters.append(sum(total_iters)/len(total_iters))
        error.append(sum(total_errs)/500/len(total_errs))
    # append the averages to the cumulative lists
    all_losses.append(loss)
    all_iters.append(iters)
    all_errs.append(error)
```

In the graphs below, I make the values of the initial weights vary from a larger range, specifically from between -0.00001 and 0.00001 all the way to -20 to 20. I would've raised this to -100 to 100, but that would yield an SVD error due to very large numbers. Each row has the training loss, expected-predicted difference, and number of iterations plotted over each magnitude. For every variance value, I create run the algorithm with 25 separate random initializations and then average the training loss, number of iterations, and expected-predicted differences.
It is important to notice that for these tests, the stopping criterion was set to 25 with 50 maximum iterations because of multiple reasons.

- More than 120 variance values $\times$ 50 algorithms trials for each variance value, I needed to save time, hence which significantly lowered the maximum number of iterations
- I observed that when the number of iterations is very large, the algorithm can significantly decrease error more consistently (especially when the initializations aren't absurdly large numbers in the thousands

and millions). My goal is to see if some initializations decreases the training loss significiantly more quickly, which prompted the high stopping criterion and low number of iterations.

- These justifications will be mentioned in further problems.

The first four rows in Figure 2 depict a decrease in training loss, which implies that the model is more comfortable with low-value initializations. As the variance of the weights got closer to 10, the training loss, even with the stopping criterion at 25, fail to decrease significantly within 25 iterations.
These are some observations between the variance of testing values and training loss/number of iterations. Note that the lowest possible average training loss value is around $0.02 = 25/500$, the stopping criterion divided by the number of data points.

- Very low values of variance hinders the rate of the algorithm to decrease training loss. We can see from the maximized iterations which converges at 0.4 at most.
- After a variance value beyond 0.00016, the number of iterations to hit below 25 decreases steadily.
- The training loss and number of iterations continues to descend over the course of variance values until 2. Lowest number of iterations occur when variance is between 0.25 and 0.50. After 2, the number of iterations maxes out again at 50.
- The orange points is the mean squared difference between the expected and predicted values. We can see that the orange points do not deviate from the blue line as much, which implies very low weight values.
- The most optimal settings for variance minimizes the number of iterations to achieve the smallest training loss, which is when $variance = 0.275$
- Larger variance values increase the training loss and number of iterations.

It is also important to note that I run multiple instances of the algorithm (25) because sometimes, the random weights can make the weights close to an optimal solution, which would throw off the error and number of iterations due to random chance. Doing multiple trials and averaging the results can leverage the results away from random chance and display more accurate patterns in the experiments.

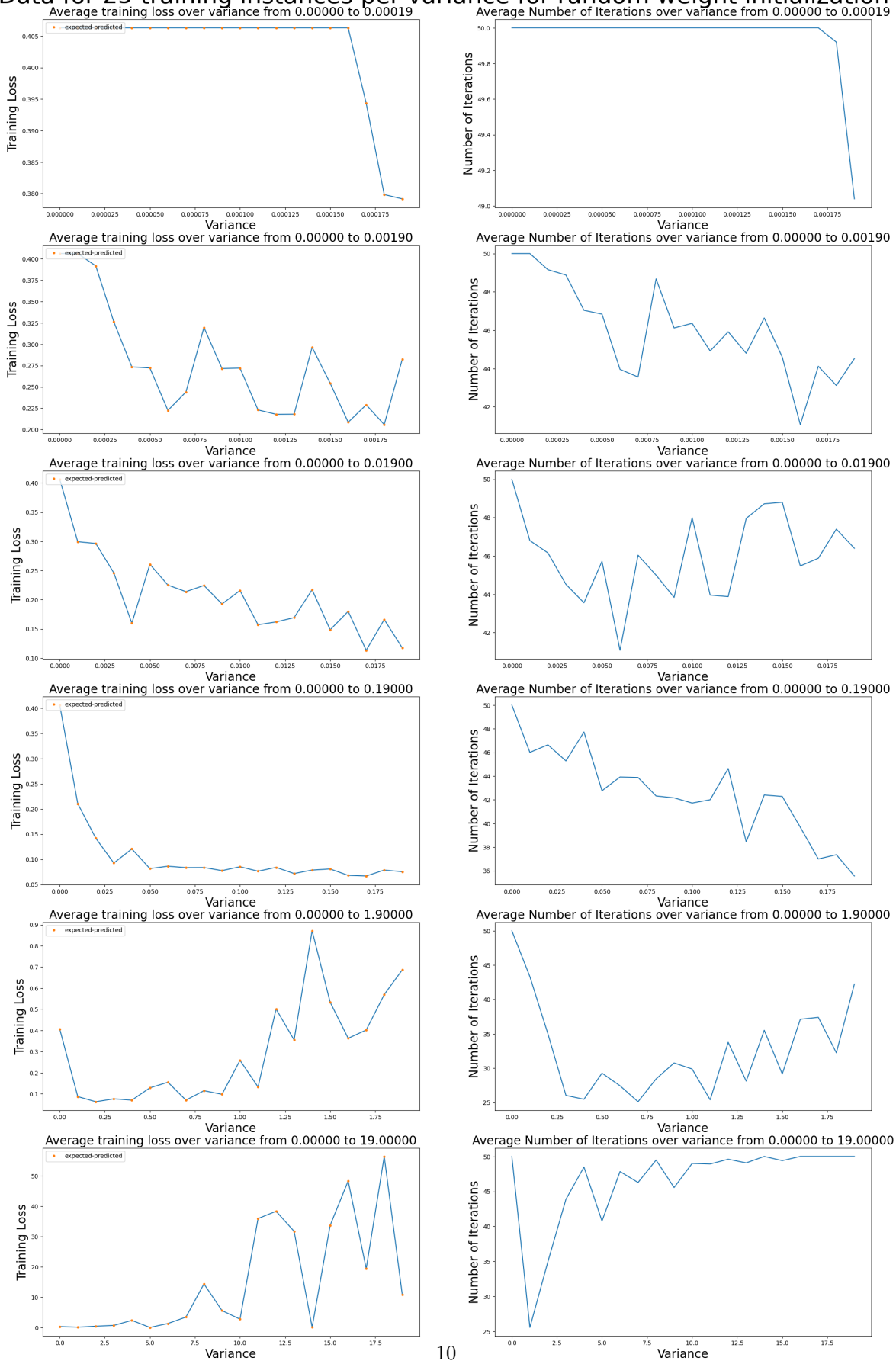

Figure 2: Changes in average training loss, error, and iterations as the variance of initial weights increase

```
# generate lambda values to test
lambs = [0.00001] + [0.00005*i for i in range(1,1000)]
# values to plot
all_losses = []
all_iters = []
all_errs = []
# make new weights for each lambda value and plot loss, errs, and number of iterations
for i in range(len(lambs)):
    # generate weights for each lambda value
    errs, steps, weights = find_weights(data, expected, lamb=lambs[i], stop=25, max_iter=100)
    all_losses.append(errs[-1])
    all_iters.append(len(errs))
    all_errs.append(np.sum((f_vector(data, weights) - expected)**2))
```

The graph below depicts the change in training loss over various λ values from 0.00001 to 0.05. New random weights were generated for each and every unique lambda value with the stopping criterion or $l(\mathbf{w}) < 25$ with at most 100 iterations. 100 iterations is comparatively a lot of repetitions of the algorithm, which probably explains the sporadic average training loss and iterations values across the lambda selections. Hence, I concluded that for more iterations, lambda has significantly less effect on the values for training loss and number of iterations.

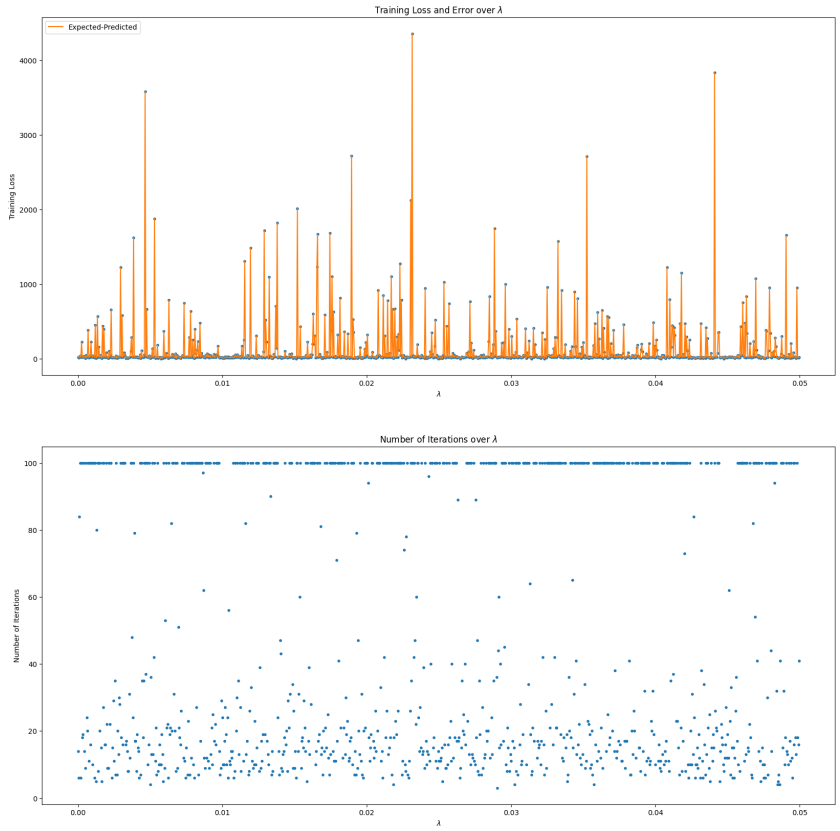

Figure 3: Changes in average training loss, error, and iterations as the values of the λ increase

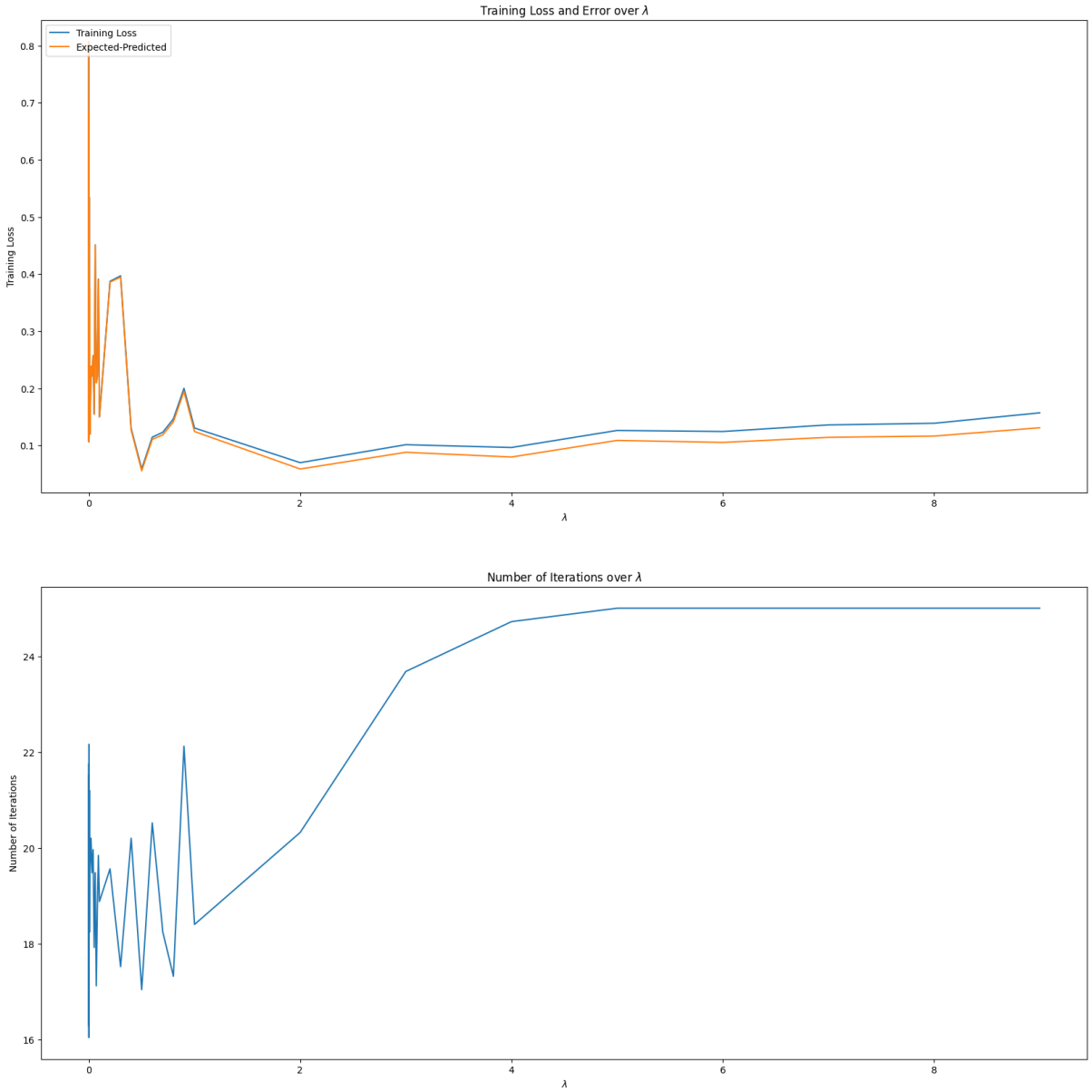

Figure 4: Changes in average training loss, error, and iterations as the values of the λ increase

And for this next graph, λ is from 10^{-7} to 10. The stopping criterion is still $l(\mathbf{w}) < 25$ with 25 maximum iterations. At each and every lambda value, 25 weights were made.
It seemed that extremely small values of λ hindered the progression of decreasing training loss quickly, and optimal settings were between 0.5 and two. The gap between the orange and blue lines in the first graph is because of the added "weight control" value $\lambda||\mathbf{w}||$. $l(\mathbf{w})$ would increase despite decreased training loss. For larger values of λ, the training loss slowly looked like increasing as it capped on the maximum iterations. So I conducted another experiment, which took much time to compute.

```
# generate lambda values
lambs = [[i*(10**j) for i in range(1,10)] for j in range(-7, 1) ]
# data to plot
all_losses = []
all_iters = []
all_errs = []
# for every lambda value find average error, loss, and number of iterations across 25 individual weights
for i in lambs:
    loss = []
    iters = []
    error = []
    for j in i:
        total_loss = []
        total_iters = []
        total_errs = []
        for k in range(25):
            errs, steps, weights = find_weights(data_2, expected_2, lamb=j, stop=25, max_iter=50)
            total_loss.append(errs[-1])
            total_iters.append(len(errs))
            total_errs.append(np.sum((f_vector(data_2, weights) - expected_2)**2))
        loss.append(sum(total_loss)/500/len(total_loss))
        iters.append(sum(total_iters)/len(total_iters))
        error.append(sum(total_errs)/500/len(total_errs))
    all_losses.append(loss)
    all_iters.append(iters)
    all_errs.append(error)
```

I began at low values of λ with a magnitude starting at 10^{-7} all the way to 10. These first four rows are progression with λ magnitudes $10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}$ respectively. These are done with stopping criterion $l(\mathbf{w}) < 25$ with 50 maximum iterations with varied λ.
Some initial observations for the graphs:

- Very low λ values doesn't seem to improve the training progression as much
- Number of iterations doesn't significantly decrease. Mainly, it fluctuates between 25 and 35 iterations.
- Similarly, the average training loss over the 25 instances of training for that one lambda value, There doesn't seem to be much of a pattern present.

From these data points, I concluded that for $\lambda < 10^{-3}$, λ does not have significantly effect on training the model.

Figure 5: Changes in average training loss, error, and iterations as λ increased

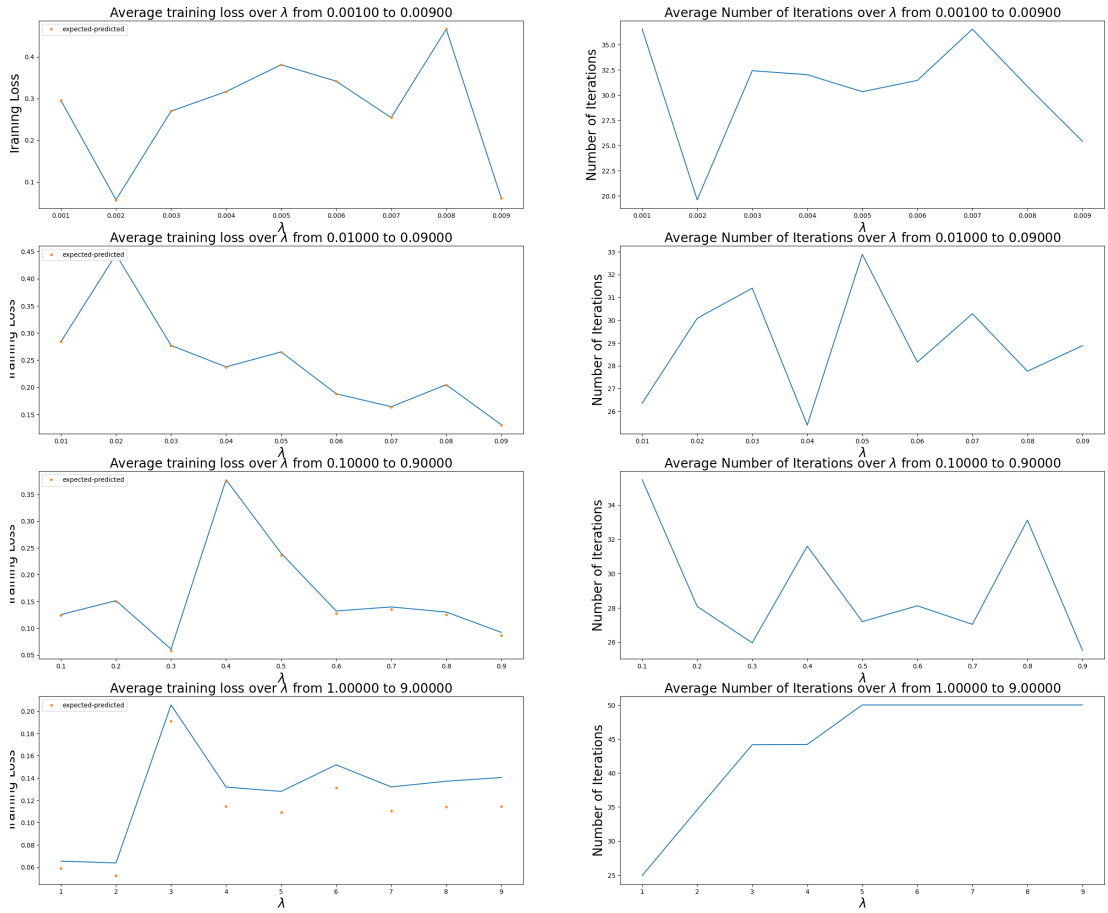

Figure 6: Changes in average training loss, error, and iterations as λ increased

From this graph, a clear pattern emerges for the largest magnitude λs. When λ is varied between 1 through 9, the number of iterations maxes out, which implies that it required all 50 iterations of the algorithm to achieve, 0.14 average training loss.

Additionally, we can see a great difference between λ values between 1 and 2, and 3-9. Less iterations, and generally less average training loss. From these graphs, I concluded that λ does not have significantly effect on training as long as $\lambda < 1$

4.3 Training and Testing on Varied Γ and λ

```
# generate gamma values
gammas = [1] + [i*(10**-2) for i in range(1, 1000)]
# generate lambda values
lambs = [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100]
# points to plot
all_errs = []
# generate data
data, expected = generate_data_1(500, 1)
# make initial weights for consistency
w = np.random.normal(0,1,(16,))
# for every lambda value, create weights and test created weights on test points from different ranges
for h in lambs:
    _, _, weights = find_weights(data, expected, lamb=h, stop=25, max_iter=25, weights=w )
    avg_errs = []
    print(h)
    for i in range(len(gammas)):
        errs = []
        for j in range(25):
            data_new, data_expected = generate_data_1(100, gammas[i])
            err = np.sum((f_vector(data_new, weights) - data_expected)**2)
            errs.append(err)
        avg_errs.append(sum(errs)/len(errs)/len(gammas))
    all_errs.append(avg_errs)
```

For this graph below I made one set of weights for each different lambda value at stopping criterion $l(\mathbf{w}) < 25$ and maximum number of iterations of 25. Each set of weights were tested on 25 different sets of test data with $N = 100$ with each value of $\mathbf{x}$ from different bounds Γ, where Γ is the value limiting:

$$\mathbf{max}([\mathbf{x}^{(n)}]_1, [\mathbf{x}^{(n)}]_2, [\mathbf{x}^{(n)}]_3) \leq \Gamma$$

This graph compares the change in gamma on the x-axis and the mean squared expected-predicted error on the y-axis.

This graph on its own did not reveal anything specific. The first noticeable pattern is that the error increases exponentially as gamma increases. Also we can only see that for very large values of Γ, λ has no effect on the average training loss across the data. However, we can zoom into specific parts of the graph to see patterns.

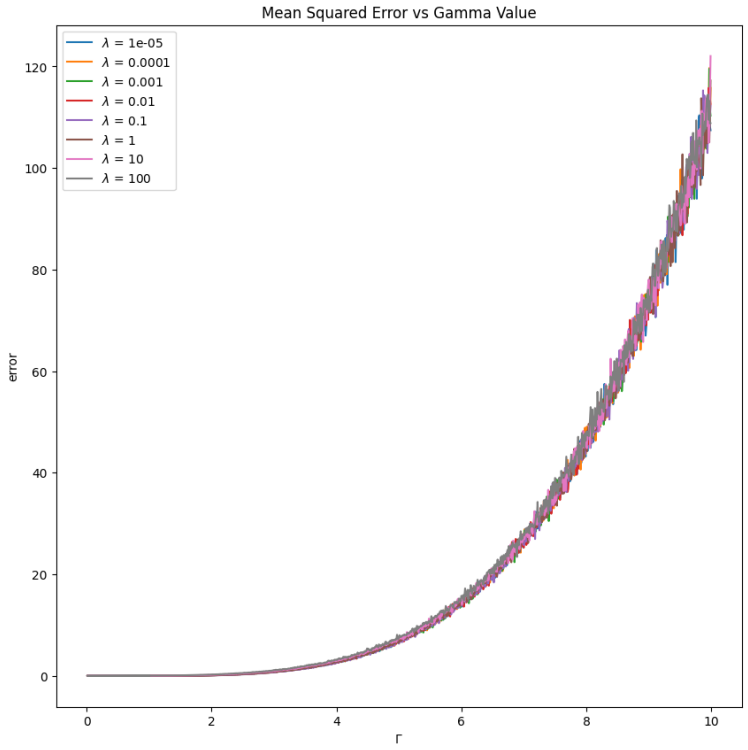

Figure 7: Changes in mean squared error as the values of the Γ and λ increase

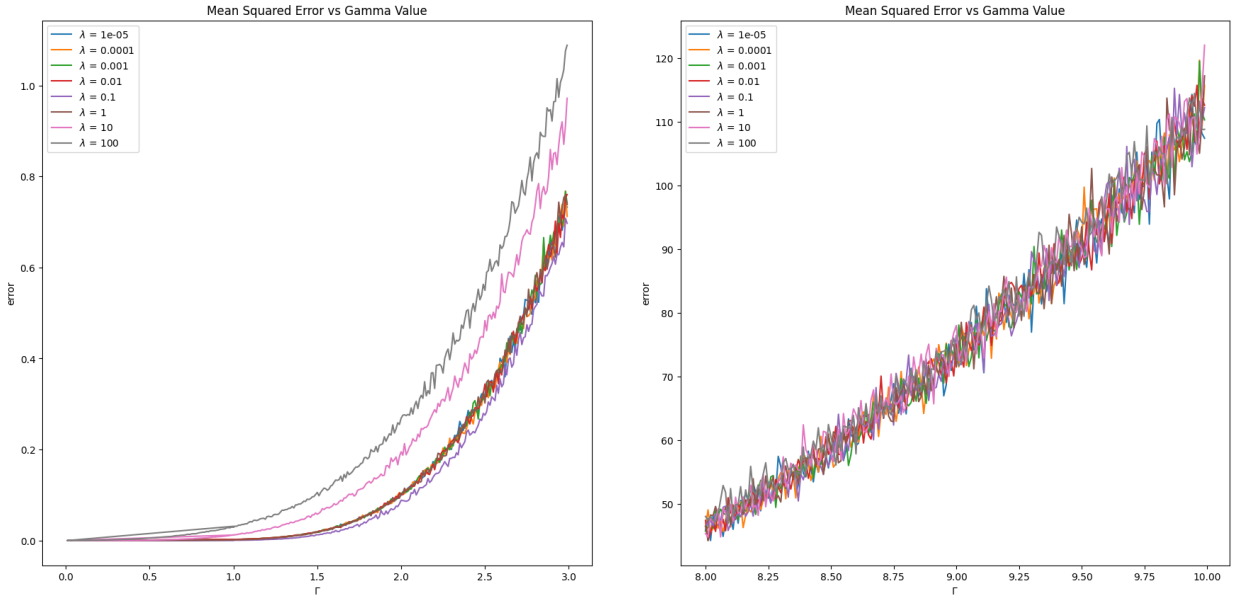

Figure 8: The graphs depict the change in mean squared error over Γ values and λ values

These two graphs depict different intervals of the larger graph. The graph on the left has an interval between 0 and 2, while the one on the right has an interval between 8 and 10.

From the graph on the left, we can see that there is a clear distinction in training loss when $\lambda > 1$. These statistics concur with the previous section, where very large values of λ can mess with training loss.

On the right however, the mean squared error of the test values from the true values doesn't seem to change with large values of lambda.

From this point on, all code is either reused or modified. Modifications will be specified, but general code will not be commented.

5 Training Model on New Non-Linear Function $[x]_1 - [x]_2[x]_3$

For a new non-linear function approximation using the model, I just created a new function that would generate a new dataset and set of expected output. I would just pass the dataset and expected outputs into the training function to create the optimal weights. No new codes were made other than this one to make the new dataset.

```
def generate_data_2(n, gamma):
    data = np.empty((0,3))
    out = np.empty((0,1))
    for i in range(n):
        x = np.random.uniform(-gamma, gamma, (3,))
        data = np.vstack((data,x))
        out = np.append(out,[x[0]-x[1]*x[2]])
    return data, out
```

5.1 Training Results

The following tables show the change in training loss and trust value over the iterations when training the model. The stopping criterion for this objective is $l(\mathbf{w}) < 0.1$ with maximum iteration of 1000. This is because I am aiming for the most accurate model possible without spending too much time. Approximately 0.1 squared error across 500 data points means that there is an average of 0.002 average squared error per data point, which is pretty accurate. Since I am only training one model, I can spend as much time as I want.

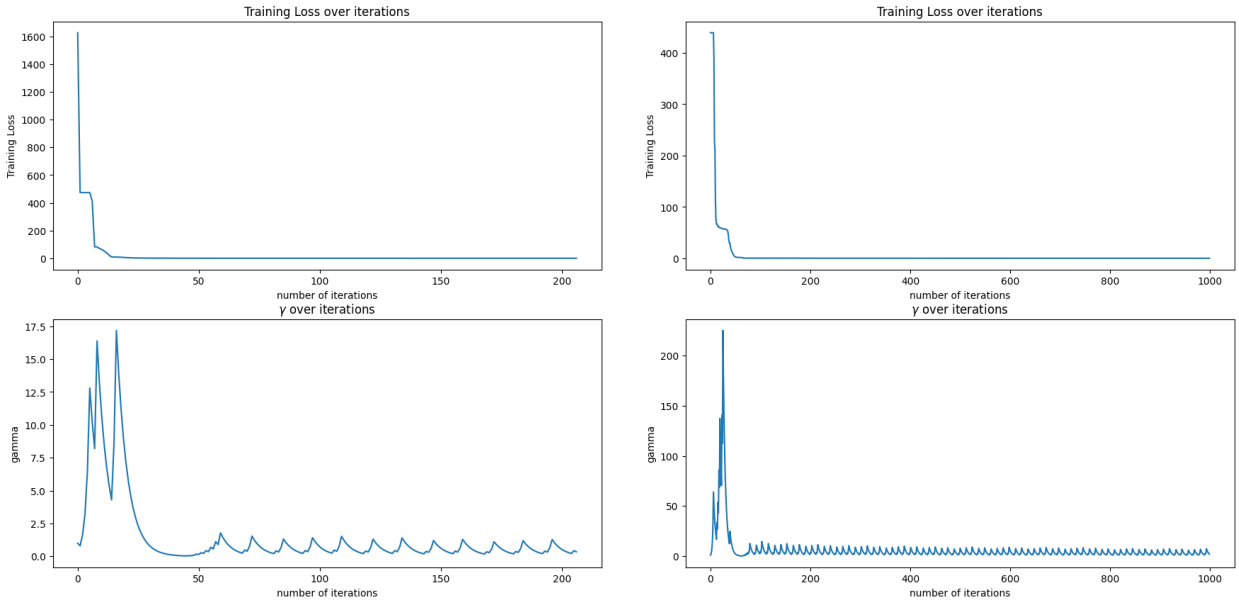

Figure 9: The graphs depict the change in training loss and gamma values over iterations for two instances of the algorithm

The graph above is an instance where the the algorithm unfortunately converged to a local minima. The final error for that specific instance was 0.12. The graph below shows a better instance where the final

training loss hit below 0.1 within 250 iterations.

5.2 Training and Testing on Varied Initializations

In the graphs below, I make the values of the initial weights vary from a larger or smaller range. Information regarding values for the different initializations is specified in page 5. Same methodologies and code have been used for both scenarios, just the data and expected results were swapped out for a new data set.

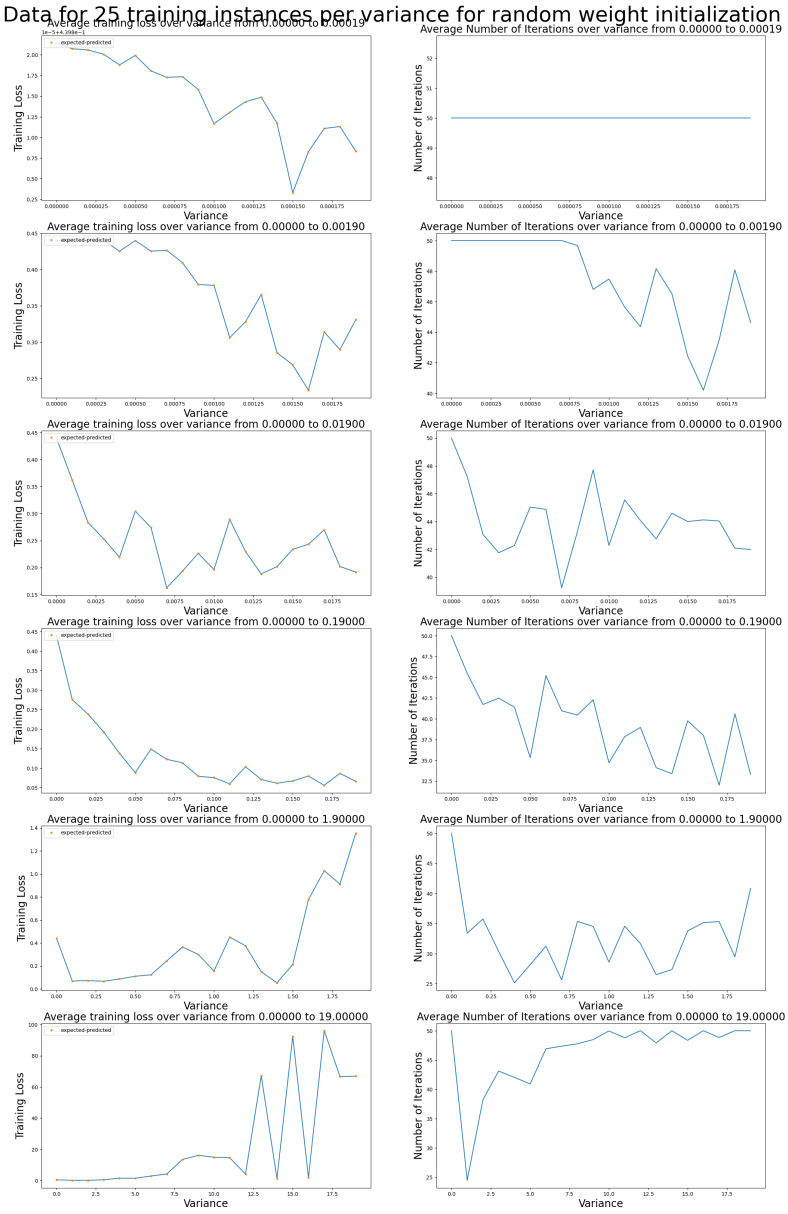

Figure 10: Changes in average training loss, error, and iterations as the variation of initial weights increase

Just like the previous non-linear function, the first four rows depict a decrease in training loss. However, this new function gives different patterns than the previous function.

- Very low variance values will immediately max out the iterations at 50. However, the average training loss for these extremely small variance values are pretty high are similar to the previous function, lingering around 0.4.
- At around 10^{-3} average training loss decreases significantly. Average training loss continues to be low for values between 0.1 and 1.5. during this interval, the average number of iterations are rather low as well, terminating at around 25 to 35 iterations on average.
- At around 1.75 variance, the average training loss increases significantly.
- After a variance of 5, the number of iterations begins to max out at 50, which implies that it could not achieve the stopping criterion by 50 iterations.

This data further supports that optimal selections of the variance is from 0.1 to 1.5.
In the graphs below, I vary λ from 0.0000001 to 10. I can see very similar results to the previous function. There seems to be no real correlation between rather small λ values and training loss and number of iterations. However, larger λ values beyond 2, significantly bottlenecks the error and maxes out on the number of iterations. The graphs for this new function seem to concur with the previous function as well. I wonder if this is a universal pattern for approximately any non-linear function with this model structure.

Figure 11

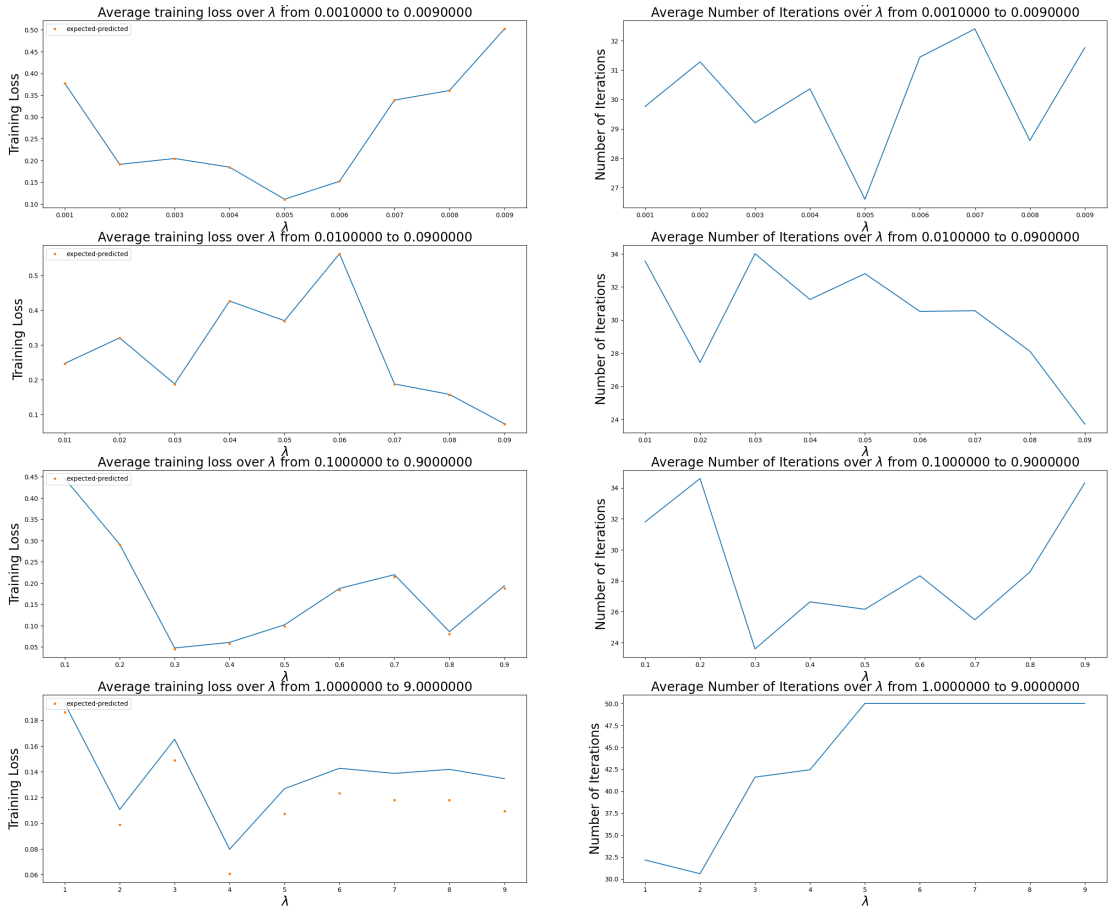

Figure 12: Changes in average training loss, error, and iterations as the values of the λ increase

5.3 Training and Testing on Varied Γ and λ

The graph for this function yields almost identical results as the previous function. As Γ increased, the mean squared error also increased as well. From the entire interval between 0 and 10, the functions are almost indistinguishable. Just like the previous function, we can zoom into the first fifth of the interval.

Figure 14 shows the exact same patterns as the previous function. The mean squared error almost increased faster for larger values of λ. This can not be said for larger values of Γ since we observed that the functions basically overlap for larger values of Γ.

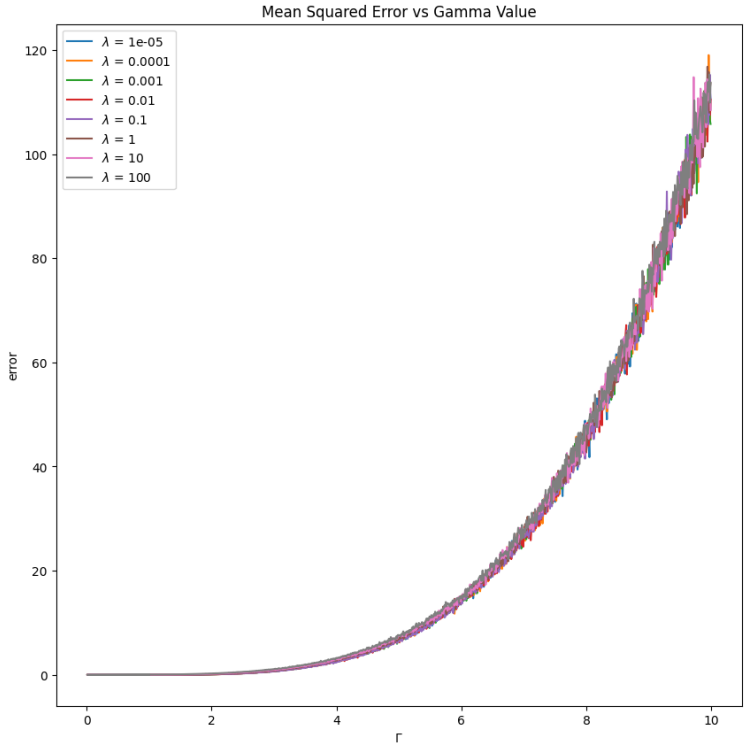

Figure 13: Changes in mean squared error as the values of the Γ and λ increase

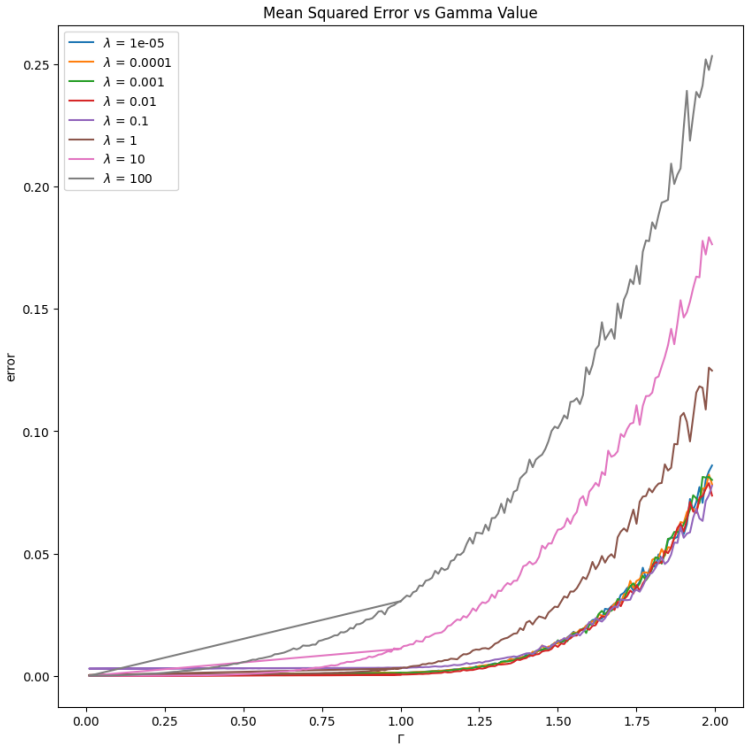

Figure 14: Changes in mean squared error as the values of the Γ and λ increase

6 Training Model on Noisy Data on Original Non-Linear Function

Some new code was made to apply the noise.

```
data, expected = generate_data_1(500, 1)
# generate random noise
n = np.random.normal(0,1,(500,))
# unit noise
n = n/np.linalg.norm(n)
# generate norms
norms = [i for i in range(1, 100, 2)]
reps = 25
```

6.1 Training Results

The model was created by tweaking the expected results as follows:

```
errs, steps, weights = find_weights(data, expected + n * 1, lamb=0.00001, max_iter=1000, stop=0.1)
```

In the graphs below, I approximate the original function after applying independent noise with norm n to the expected results. The graph below is trained under stopping criterion $l(\mathbf{w}) < 0.1$ with maximum iterations at 1000.

These models are trained with stopping criterion $l(\mathbf{w} < 25)$ at 25 maximum iterations. For the following instance, the norm of the noise was set to 5.

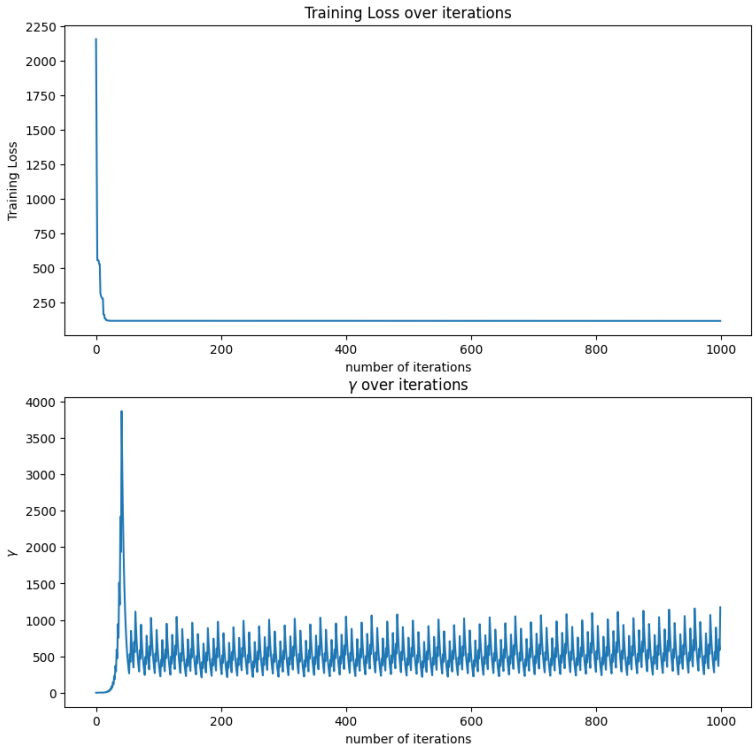

Figure 15: Changes in average training loss, error

The final training loss at the end of the 1000 iterations was approximately 118. Since the independent random noise was added, the model seems to have a harder time analyzing, identifying, and characterize the data points with the ϕ function.
This graph does not speak on its own. I varied the noise between 0 and 100. I trained the model with the noisy expected results then compare them with the original results. As the norm of the noise n increases, the mean squared error also tends to increase.

```
# values to plot
avg_errs = []
avg_iter = []
all_loss = []
# for each and every norm, test 25 separate weights with 25 maximum iterations.
for norm in norms:
    total_errs = []
    total_iters = []
    losses = []
    for i in range(reps):
        errs, steps, weights = find_weights(data, expected + n*norm, lamb=0.0001,
        stop=25, max_iter=25)
        losses.append(errs)
        total_errs.append(np.sum((f_vector(data, weights)- expected)**2)/500)
        total_iter.append(len(errs))
    avg_errs.append(sum(total_errs)/len(data)/reps)
    avg_iter.append(sum(total_iter)/len(data)/reps)
    all_loss.append(losses)
```

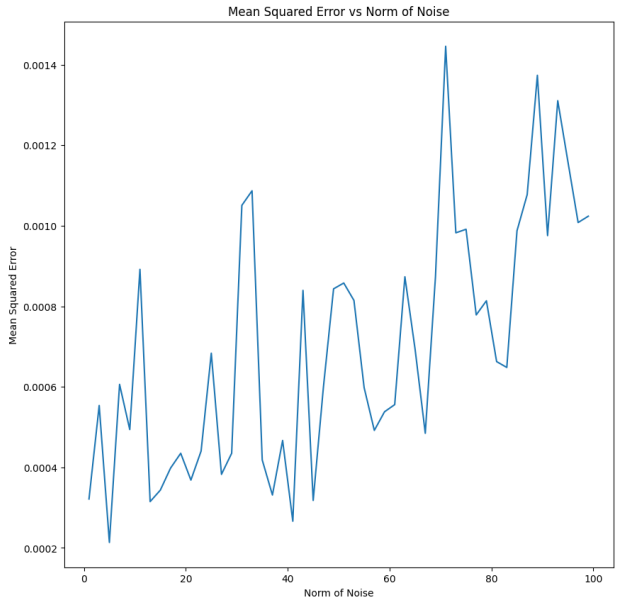

Figure 16: Changes in mean squared error as norm of noise increase

6.2 Training and Testing on Varied Initializations

```
# this code is essentially the same as the previous code that tested for varying variation.
# However, the same tests were done through different noise norms.
all_losses = []
all_iters = []
all_errs = []
all_vals = [0.1] + [0.5*i for i in range(1,11)]
norms = [1, 5, 10, 20, 50]
for norm in norms:
    nloss = []
    niters = []
    nerror = []
    for i in all_vals:
        total_loss = []
        total_iters = []
        total_errs = []
        for k in range(10):
            errs, steps, weights = find_weights(data, expected+n*norm, lamb=0.00001, stop=25, max_iter=5
            total_loss.append(errs[-1])
            total_iters.append(len(errs))
            total_errs.append(np.sum((f_vector(data, weights) - expected)**2))
        nloss.append(sum(total_loss)/500/len(total_loss))
        niters.append(sum(total_iters)/len(total_iters))
        nerror.append(sum(total_errs)/500/len(total_errs))
    all_losses.append(nloss)
    all_iters.append(niters)
    all_errs.append(nerror)
```

The graph below depicts the variation of variance for the initial training weights and its effects on the training loss. From first row to the fifth the norm of noise is 1, 5, 10, 20, 50 respectively. We can see that the number of iterations is constantly maxed out, and the average training loss is often over 50. The random independent noise coupled with made initializations of weights through higher variance truly destroys the model.

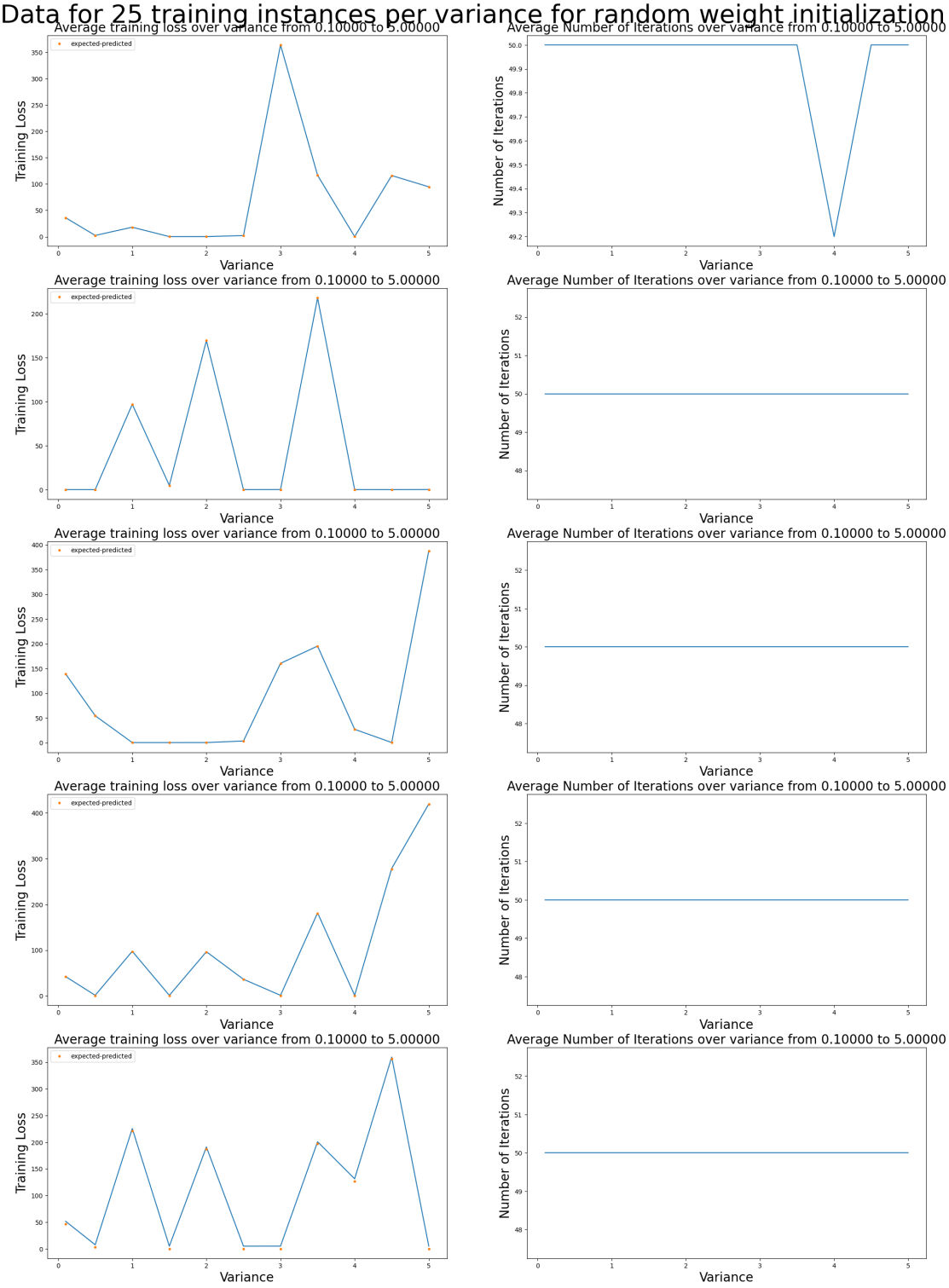

Figure 17: Changes in average training loss, error, and iterations as the variation of the initial weights and norm of noise increase

For the graphs below, I varied λ from 0.1 to 5, and it seems the low values of λ significantly decreases. However, the mean squared error,the orange points, remained low even when the noise was present, which makes me believe that the model is pretty robust against independent noise. I can still see that the iterations are maxed out most of the time, which implies that it takes more iterations to approach the desired stopping criteria.

```
# this code is essentially the same as the previous code that tested for varying variation.
# However, the same tests were done through different noise norms.
lambs = [0.1] + [0.5*i for i in range(1,11)]
all_losses = []
all_iters = []
all_errs = []
norms = [1, 5, 10, 20, 50]
for norm in norms:
    print(norm)
    nloss = []
    niters = []
    nerror = []
    for l in lambs:
        total_loss = []
        total_iters = []
        total_errs = []
        for k in range(10):
            errs, steps, weights = find_weights(data, expected+n*norm, lamb=0,
            stop=25, max_iter=50)
            total_loss.append(errs[-1])
            total_iters.append(len(errs))
            total_errs.append(np.sum((f_vector(data, weights) - expected)**2))
        nloss.append(sum(total_loss)/500/len(total_loss))
        niters.append(sum(total_iters)/len(total_iters))
        nerror.append(sum(total_errs)/500/len(total_errs))
    all_losses.append(nloss)
    all_iters.append(niters)
    all_errs.append(nerror)
```

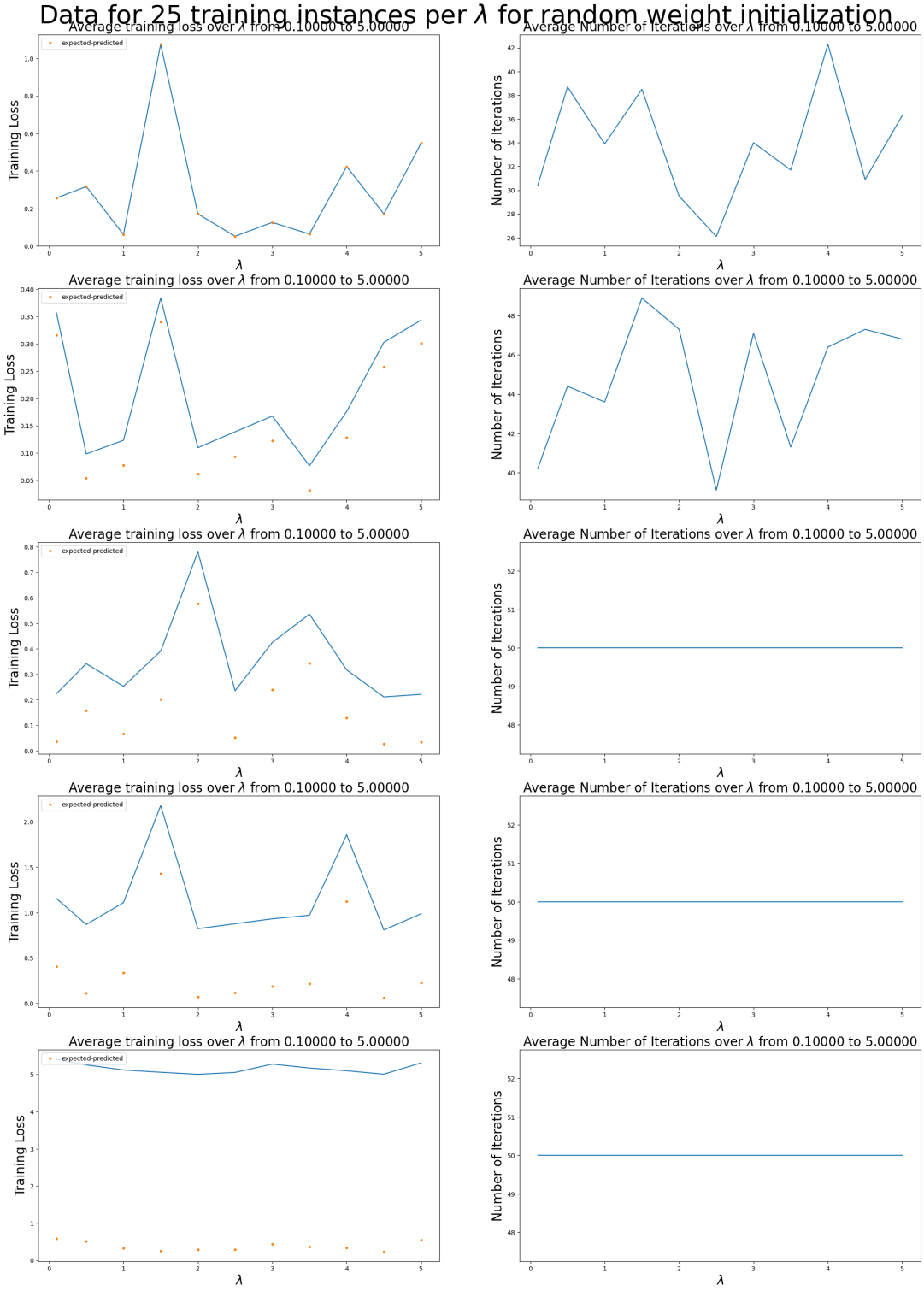

Figure 18: Changes in average training loss, error, and iterations as the values of the λ and norm noise increase

6.3 Training and Testing on Varied Γ and λ

The following graphs depict the change in mean squared error as Γ is increased for each respective λ value. The graphs at the grand scale in intervals 0 to 10 do not signify anything in particular. The graph show a similar pattern to previous functions with tests on the Γ value. The norm of noise also doesn't seem to change the error rate across different Γ values. However, let us zoom into the first section.

```
# this code is essentially the same as the previous code that tested for varying variation.
# However, the same tests were done through different noise norms.
gammas = [1] + [i*(10**-2) for i in range(1, 1000)]
vals = [0.001, 0.01, 0.1, 1, 10, 100]
all_errs = []
data, expected = generate_data_1(500, 1)
w = np.random.normal(0,1,(16,))
for h in vals:
    print(h)
    errors = []
    for g in vals:
        _, _, weights = find_weights(data, expected+n*g, lamb=h, stop=25, max_iter=25, weights=w )
        avg_errs = []
        for i in range(len(gammas)):
            errs = []
            for j in range(25):
                data_new, data_expected = generate_data_1(100, gammas[i])
                err = np.sum((f_vector(data_new, weights) - data_expected)**2)
                errs.append(err)
            avg_errs.append(sum(errs)/len(errs)/len(vals))
        errors.append(avg_errs)
    all_errs.append(errors)
```

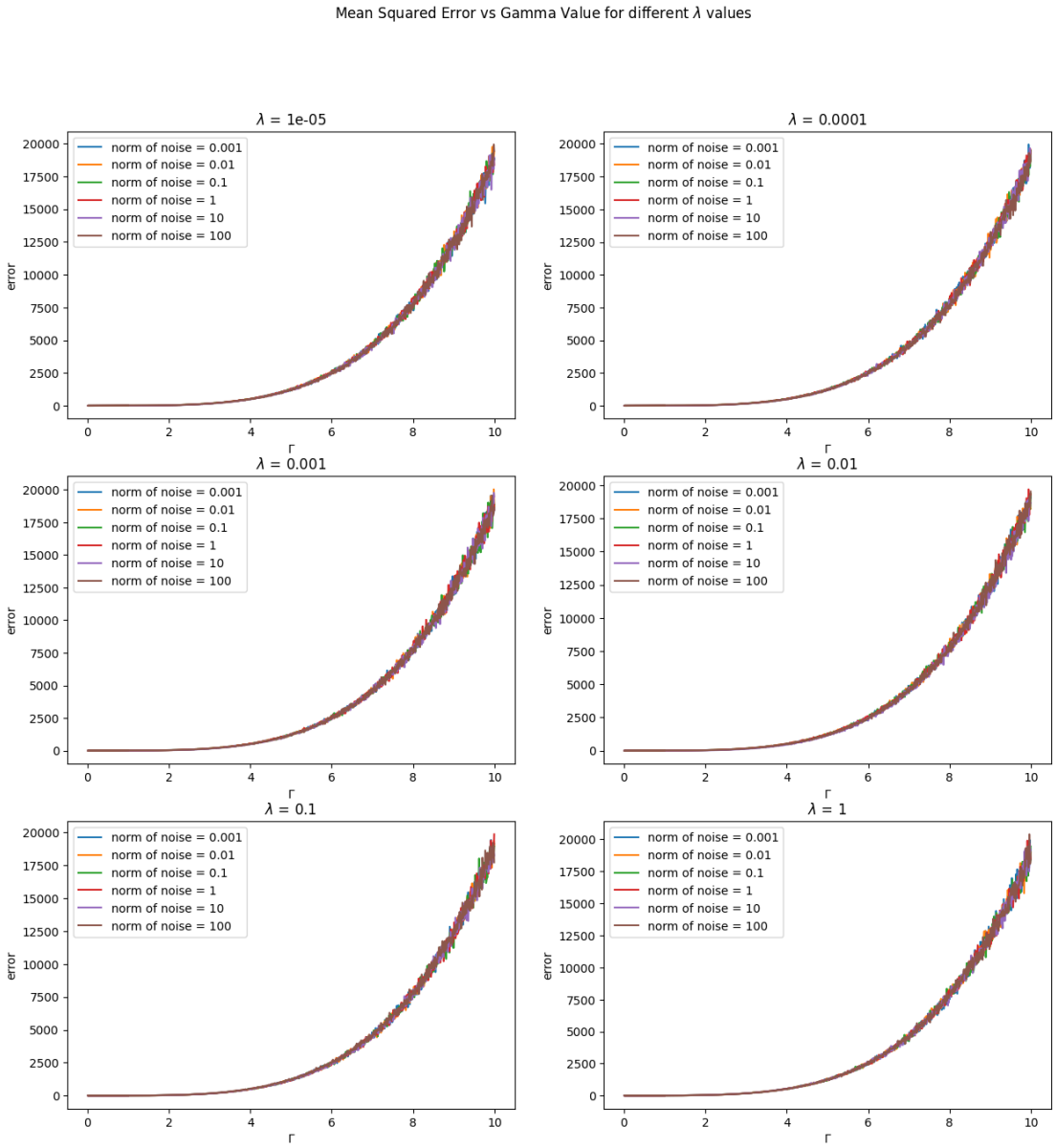

Figure 19: Changes in average squared error as the values of the Γ increase

Here, we can observe that the higher norm of noise actually results in greater mean squared error across the Γ values. The difference of mean squared error between the norms is only prevalent for certain values of λ . As λ increased, the average training loss with higher noise norm values assimilated with the other norms.

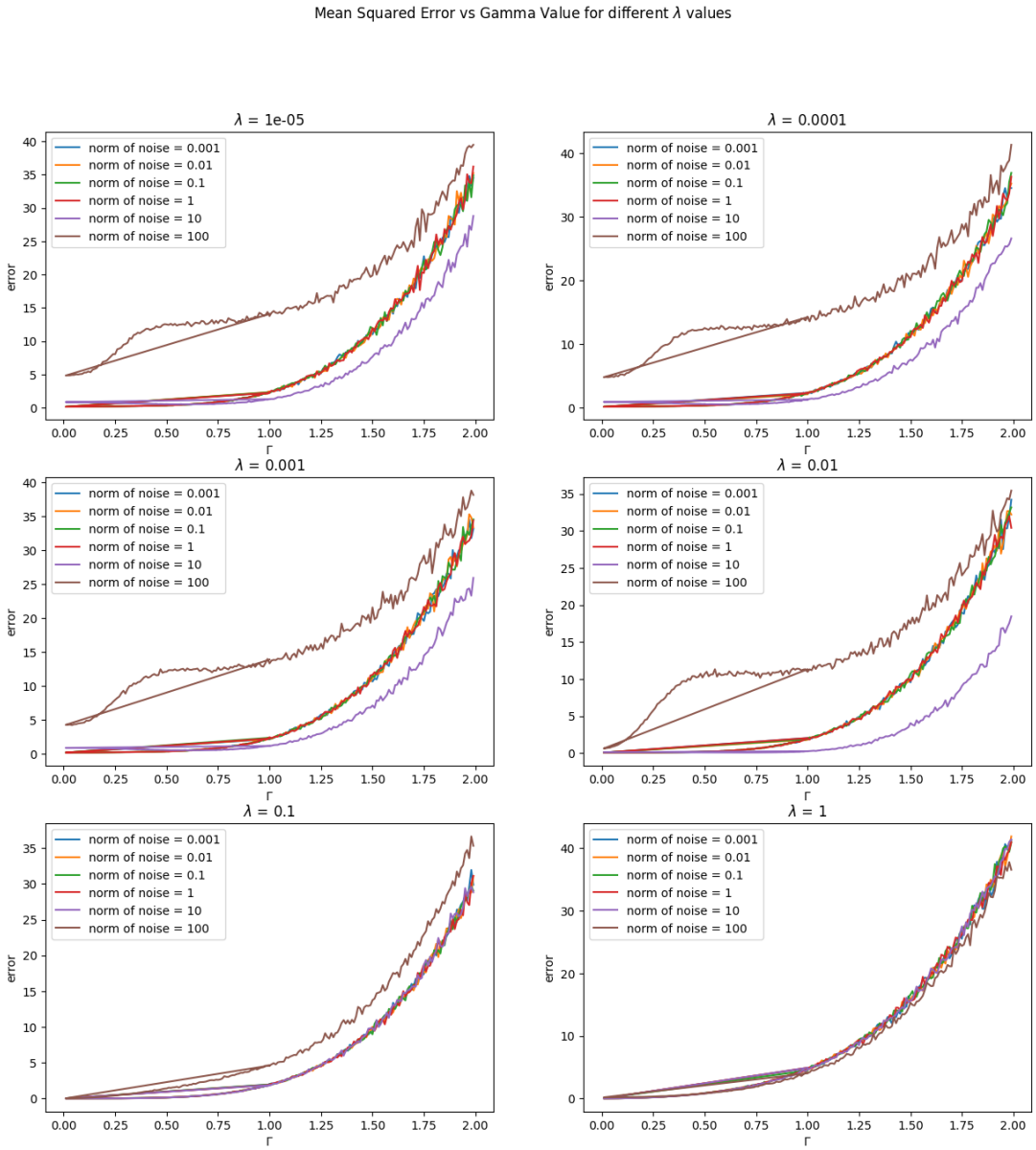

Figure 20: Changes in average squared error as the values of the Γ increase

And as predicted, higher Γ values indiscriminantly yielded higher error, which was a pattern from the previous functions as well.

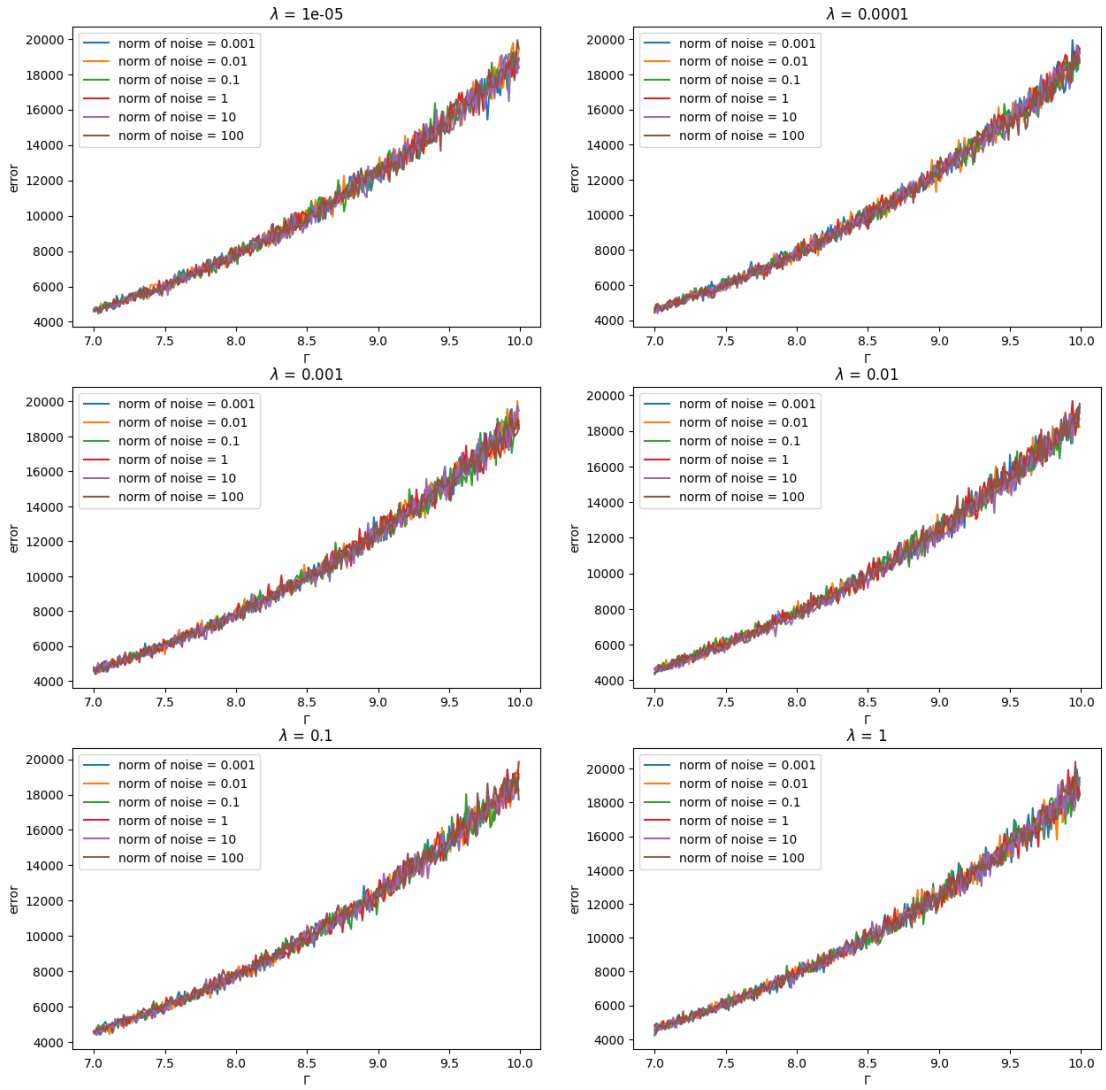

Figure 21: Changes in average squared error as the values Γ increase

From the patterns within the experiments, it seems that initial training weights between -0.75 and 0.75 with $\lambda < 1$ is the most optimal initialization for training.

7 Visualizing the Model

```
# this was just for fun
def contour_plot(weights, rho, alpha, N=100):
    data = np.random.uniform(-alpha, alpha, (N,2))
    data_expected = np.concatenate((data, np.reshape(rho-np.multiply(data[:,0],data[:,1]), (N,1))), axis

    x1 = data_expected[:,0]
    y1 = data_expected[:,1]
    z1 = data_expected[:,2]
    # compute fw
    data_predicted = np.empty((0,1))
    for row in data_expected:
        data_predicted = np.vstack((data_predicted, f(row, min_weights)))
    x2 = data_expected[:,0]
    y2 = data_expected[:,1]
    z2 = data_expected[:,2] + data_predicted[:,0] - rho

    avg_err = np.sum(np.square(z2 - z1))/N

    return x1, y1, z1, x2, y2, z2, avg_err
fig = plt.figure()
fig.set_size_inches(10, 15)
all_vals = [val for val in range(1001)]
vals = [0,1,10,50, 100, 500]
errs_rho = []
for i in range(len(all_vals)):
    x1, y1, z1, x2, y2, z2, avg = contour_plot(min_weights, all_vals[i], 1)
    errs_rho.append(avg)
    if i in vals:
        index = vals.index(i)
        ax = fig.add_subplot(int(len(vals)/2),2,index+1, projection='3d')
        ax.scatter(x1, y1, z1, marker="^", c='blue')
        ax.scatter(x2, y2, z2, marker="^", c='red')
        ax.set_xlabel(r'$[x]_1$')
        ax.set_ylabel(r'$[x]_2$')
        ax.set_zlabel(r'$[x]_3$')
        ax.set_title(r'Values of X where $\rho$ = {} and $\alpha$ = 1' '\n' 'Average Sq err= {}'.format
plt.show()

fig = plt.figure()
fig.set_size_inches(10, 15)
errs_alpha = []
for i in range(len(all_vals)):
```

```
    x1, y1, z1, x2, y2, z2, avg = contour_plot(min_weights, 0, all_vals[i])
    errs_alpha.append(avg)
    if i in vals:
        index = vals.index(i)
        ax = fig.add_subplot(int(len(vals)/2),2,index+1, projection='3d')
        ax.scatter(x1, y1, z1, marker="^", c='blue')
        ax.scatter(x2, y2, z2, marker="^", c='red')
        ax.set_xlabel(r'$[x]_1$')
        ax.set_ylabel(r'$[x]_2$')
        ax.set_zlabel(r'$[x]_3$')
        ax.set_title(r'Values of X where $\rho$ = 0 and $\alpha$ = {}' '\n' 'Average Sq err= {}'.format
plt.show()
```

Create a set of $\mathbf{x}$ such that $[x]_1[x]_2 + [x]_3 = \rho$ The model will provide the following about $\mathbf{x}$

$$[x]_1[x]_2 + [x]_3 = f_{\mathbf{w}}(\mathbf{x})$$

Since in most cases $f_{\mathbf{w}}(\mathbf{x}) \neq 0$, we can calculate how much residuals exist between $f_{\mathbf{w}}(\mathbf{x})$ and ρ by solving for $[x]_3$ in the equation above. Initially, we know that $[x]_!$ and $[x]_2$ are consistently throughout every single equation. Redefine the equation from provided by the model and solve for $[x]'_3$ and subtract the two equations.

$$\begin{aligned} [x]_1[x]_2 + [x]'_3 &= f_{\mathbf{w}}(\mathbf{x}) \\ - \quad [x]_1[x]_2 + [x]_3 &= \rho \\ \hline [x]'_3 - [x]_3 &= f_{\mathbf{w}}(\mathbf{x}) - \rho \end{aligned}$$

Ultimately, we get:

$$[x]'_3 = [x]_3 + f_{\mathbf{w}}(\mathbf{x}) - \rho$$

In simpler terms, the model's $[x]'_3$ is the original $[x]_3 + predicted - expected$.
We will predetermine $([\mathbf{x}^{(n)}]_1, [\mathbf{x}^{(n)}]_2$ from a uniform random distribution between $-\alpha$ and α. Notice that the value of ρ depends on whatever value that I choose. Also notice that the "normal" weights were trained with values such that $\mathbf{max}([\mathbf{x}^{(n)}]_1, [\mathbf{x}^{(n)}]_2, [\mathbf{x}^{(n)}]_3) \leq 1$, hence the maximum value for y for any such $([\mathbf{x}^{(n)}]_1, [\mathbf{x}^{(n)}]_2, [\mathbf{x}^{(n)}]_3)$ is $(1)(1) + 1 = 2$. The weights for the following model will only approximate/predict accurately around such points where the domain of the training data lay.
The three dimensional plot will display the 3-D contour map of 4-D non-linear equation

$$[x]_1[x]_2 + [x]_3 = y$$

. It will also plot how much $[x]_3$ is off by to match the contour map when $y = \rho$.
In the following plots, I will be varying the values of ρ and α

From these graphs, I can deduce that the algorithm approximately best around the range of y values, regardless of x values as long as the function of the x values yield between -1 and 1.

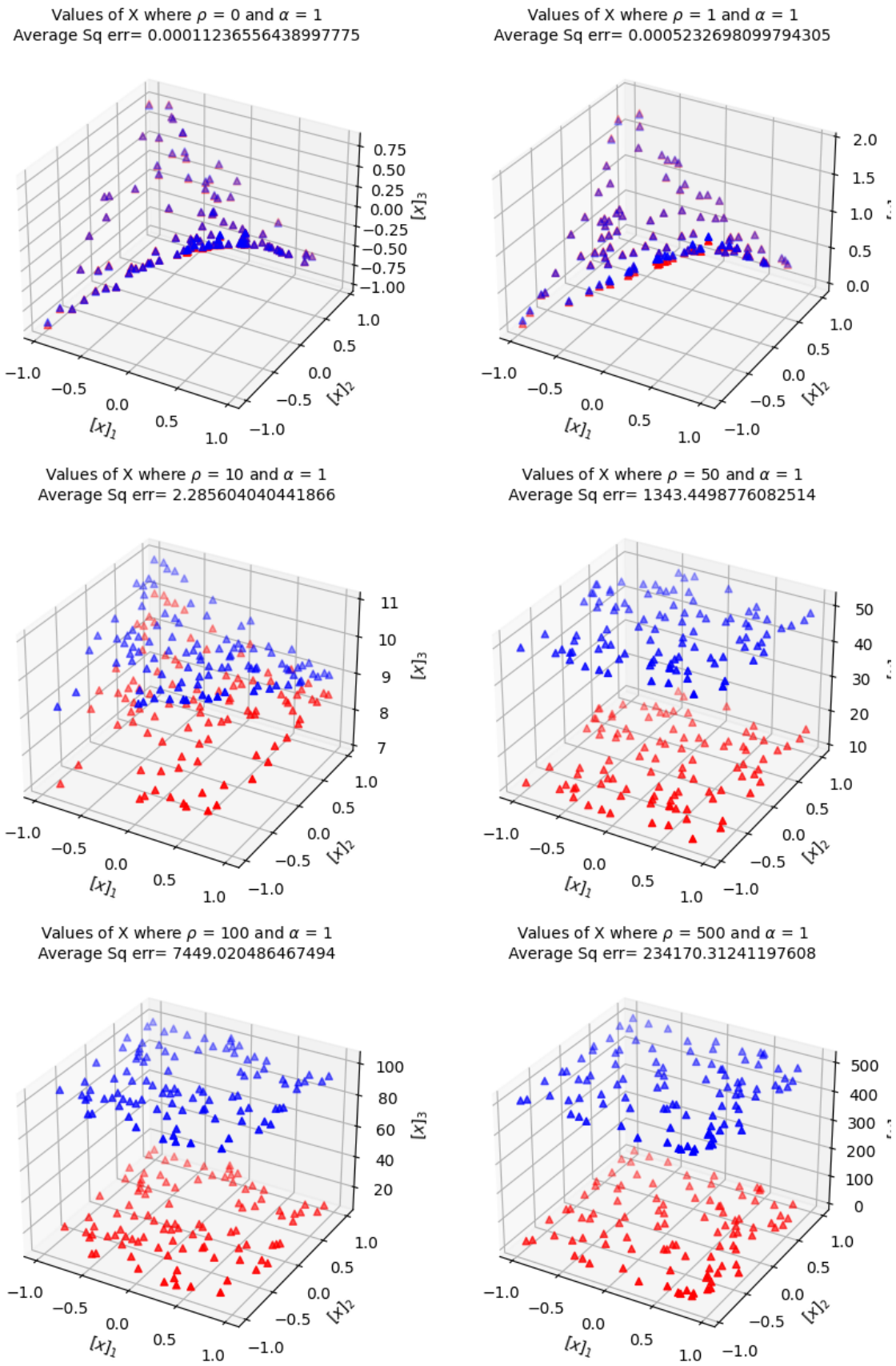

Figure 22: The graphs depict the error in $[x]_3$ when ρ is changed

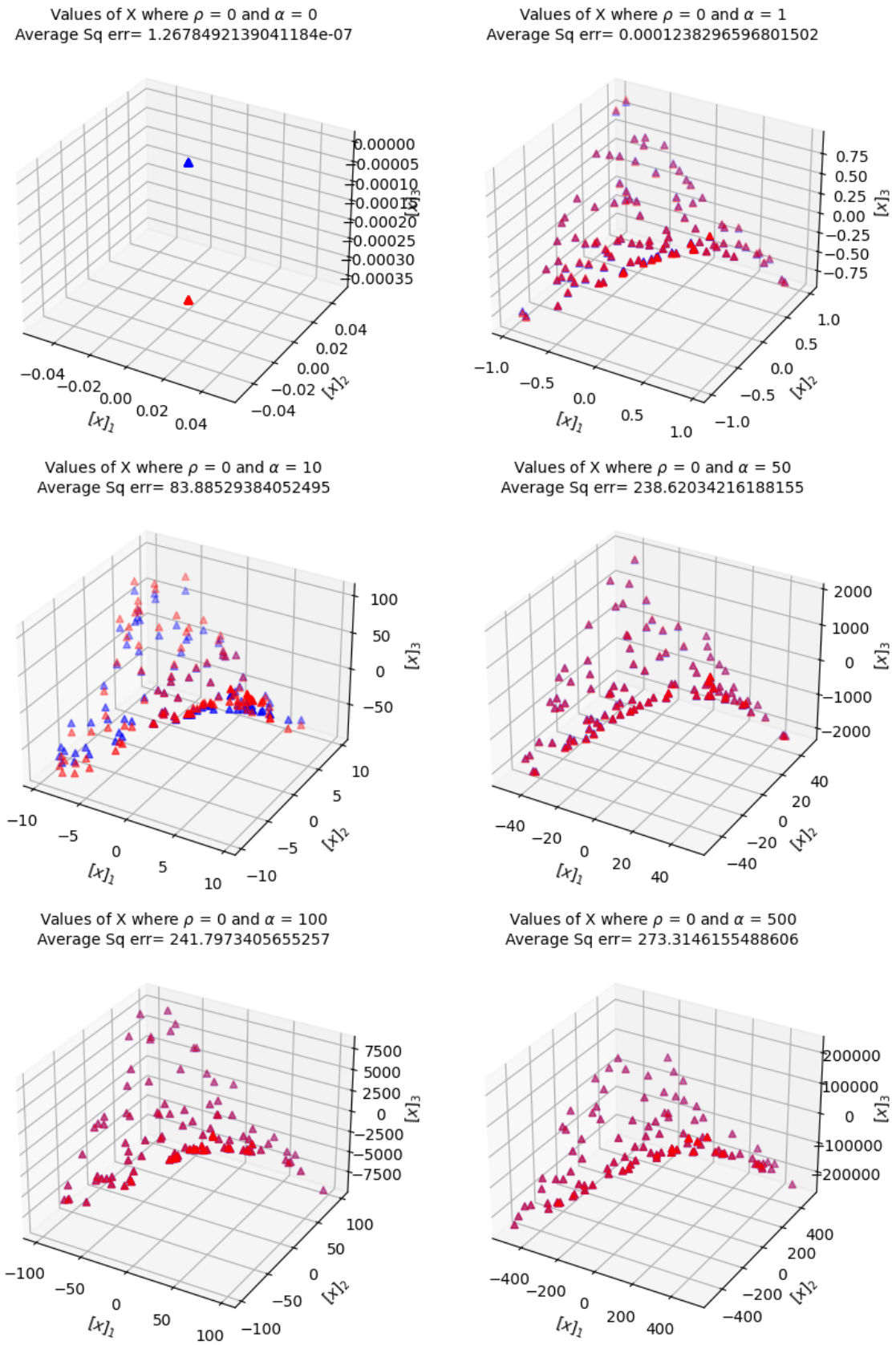

Figure 23: The graphs depict the error in $[x]_3$ when α is changed

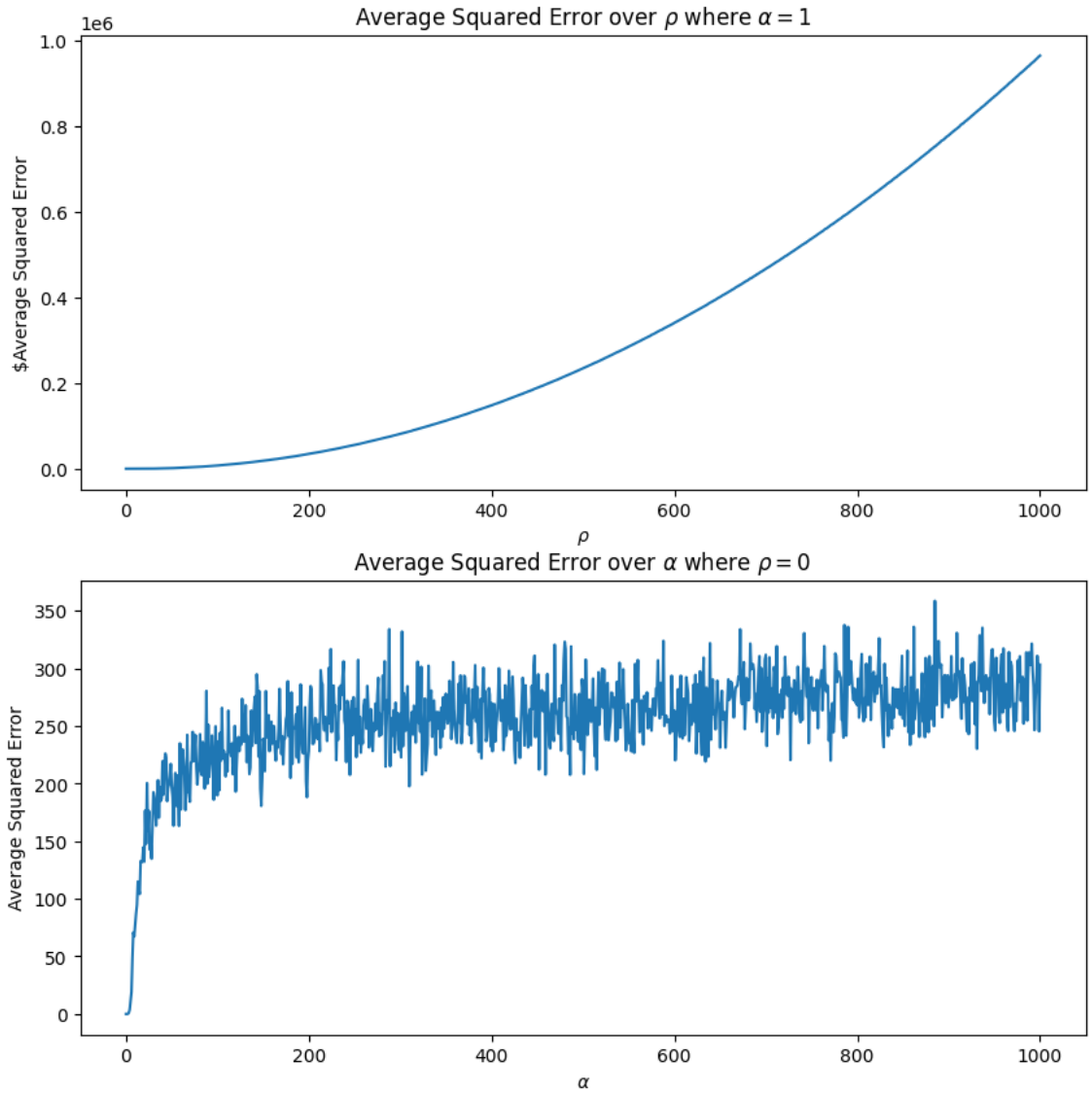

Figure 24: The graphs depict the error in $[x]_3$ when α is changed