ECE194N HW 2

XOR Report

February 22, 2019

Student: Erik Rosten
Perm Number: 7143571
Email: erosten@ucsb.edu

Department of Electrical and Computer Engineering, UCSB

Original Code

```python
import numpy as np
import matplotlib.pyplot as plt

# reference for # of hidden layer nodes
    #https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-i
# reference for building your own neural net
    #https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08

def visualize_samples(x,y):
    plt.scatter(x[:,0], x[:,1])

    for i, label in enumerate(y.tolist()):
        plt.annotate(label, (x[i,0], x[i,1]))
    plt.xlabel('Input 1')
    plt.ylabel('Input 2')
    plt.title('XOR of Input 1 and Input 2 and it\'s Annotated Output')
    plt.xticks([0,1])
    plt.yticks([0,1])
    plt.show()

def visualize_classification_regions(x,y, nn):
    points = []
    classes = []
    for i in range(10000):
        point = np.random.rand(1,2)
        points.append(point)
        network_output = nn.forward_pass(point)
        classes.append(map_nn_output(network_output))

    points = np.array(points).reshape(-1,2)
    classes = np.array(classes).reshape(-1,)
    color = ['red' if label == -1 else 'blue' for label in classes]
    plt.scatter(points[:,0], points[:,1], color=color, s = 5)


    plt.scatter(x[:,0], x[:,1])

    for i, label in enumerate(y.tolist()):
        plt.annotate(label, (x[i,0], x[i,1]))
    plt.xlabel('Input 1')
    plt.ylabel('Input 2')
    plt.title('XOR of Input 1 and Input 2 and it\'s Annotated Label')
    plt.xticks([0,1])
    plt.yticks([0,1])
    plt.show()
    plt.show()



def sigmoid(x):
    return 1.0/(1+ np.exp(-x))
```

```

def sigmoid_derivative(x):
    return sigmoid(x) * (1.0 - sigmoid(x))

# use squared error loss
def loss(y_pred, y):
    return .5 * np.sum((y_pred - y)**2)

def plot_loss(loss):
    plt.plot(np.arange(loss.shape[0]), loss, linestyle = '--', marker = 'o', color = 'b')
    plt.xlabel('Iterations')
    plt.ylabel('Squared Error Loss')
    plt.show()


# 0 -> 1
# 1 -> -1
def map_nn_output(y):
    return -2 * (y - 0.5)

# -1 -> 1
#  1 -> 0
def map_nn_input(y):
    return (-0.5 * y + 0.5).astype('int')


class neural_net:
    def __init__(self, x, y):
        self.input = x
        self.num_hidden_layer_perceptrons = 20
        # random returns random values between 0 and 1 in a given shape
        self.w1 = np.random.rand(self.input.shape[1],self.num_hidden_layer_perceptrons)
        self.w2 = np.random.rand(self.num_hidden_layer_perceptrons,1)
        self.y = y
        self.output = np.zeros(self.y.shape)

    def forward_pass(self, input = None):
        input = input if input is not None else self.input
        self.in_h_layer = np.dot(input,self.w1)
        self.hidden_layer = sigmoid(self.in_h_layer)
        self.in_output = np.dot(self.hidden_layer, self.w2)
        self.output = sigmoid(self.in_output)
        return np.round(self.output)

    def back_prop(self, alpha):
        delta_0 = (self.output - self.y) * sigmoid_derivative(self.in_output)
        d_w2 = np.dot(self.hidden_layer.T, delta_0)
        d_w1 = np.dot(self.input.T, (np.dot(delta_0, self.w2.T) *
            sigmoid_derivative(self.in_h_layer)))

        self.w1 -= alpha * d_w1
        self.w2 -= alpha * d_w2

    def generate_noise(self, x, sigma):
```

```
        x_noisy = []
        for i in range(x.shape[0]):
            mu = x[i].reshape(-1,)
            cov = np.array([[sigma, 0],[0, sigma]])
            x_noisy.append(np.random.multivariate_normal(mu,cov))

        return np.array(x_noisy).reshape(-1,2)

    def train(self, num_iter, alpha, x, useNoise, sigma):
        self.loss = np.zeros(num_iter)
        for i in range(num_iter):
            if (useNoise == True):
                self.forward_pass(self.generate_noise(x, sigma))
            else:
                self.forward_pass(x)
            if (i > 40000):
                self.back_prop(alpha / 10000)
            elif (i > 30000):
                self.back_prop(alpha / 1000)
            elif (i > 20000):
                self.back_prop(alpha / 100)
            elif (i > 10000):
                self.back_prop(alpha / 10)
            else:
                self.back_prop(alpha)
            self.loss[i] = loss(self.output,self.y)

    def loss(self):
        return loss(self.output, self.y)



def run_neural_net():
    # define neural net inputs
    x = np.array([[1,1],
                  [0,0],
                  [1,0],
                  [0,1]])
    y = np.array([[1],[1],[-1],[-1]])
    visualize_samples(x,y)
    y = map_nn_input(y)
    # generate noise
    # x_noisy, y_new = generate_noise(x, y, 1)

    # define neural net
    nn = neural_net(x, y)
    # define neural net training inputs
    num_iter = 100000
    alpha = 1
    # train the net
    nn.train(num_iter, alpha, x, True, 2)
    # map outputs from 0 -> 1 and 1 -> -1
    y_pred = map_nn_output(nn.output)
    # print predictions
```

```python
159     # print(y_pred)
160     # plot loss
161     plot_loss(nn.loss)
162     # visualize classification regions
163     visualize_classification_regions(x,y, nn)
164 
165 
166 
167 if __name__ == "__main__":
168     run_neural_net()
```

This code is run through the *run_neural_net*() function.

Part a: Visualizing Samples and Their Classes

This section is accomplished in line 221 of the main function with *visualize_samples*(x, y). This function is defined on lines 86-96 and produces the output below.

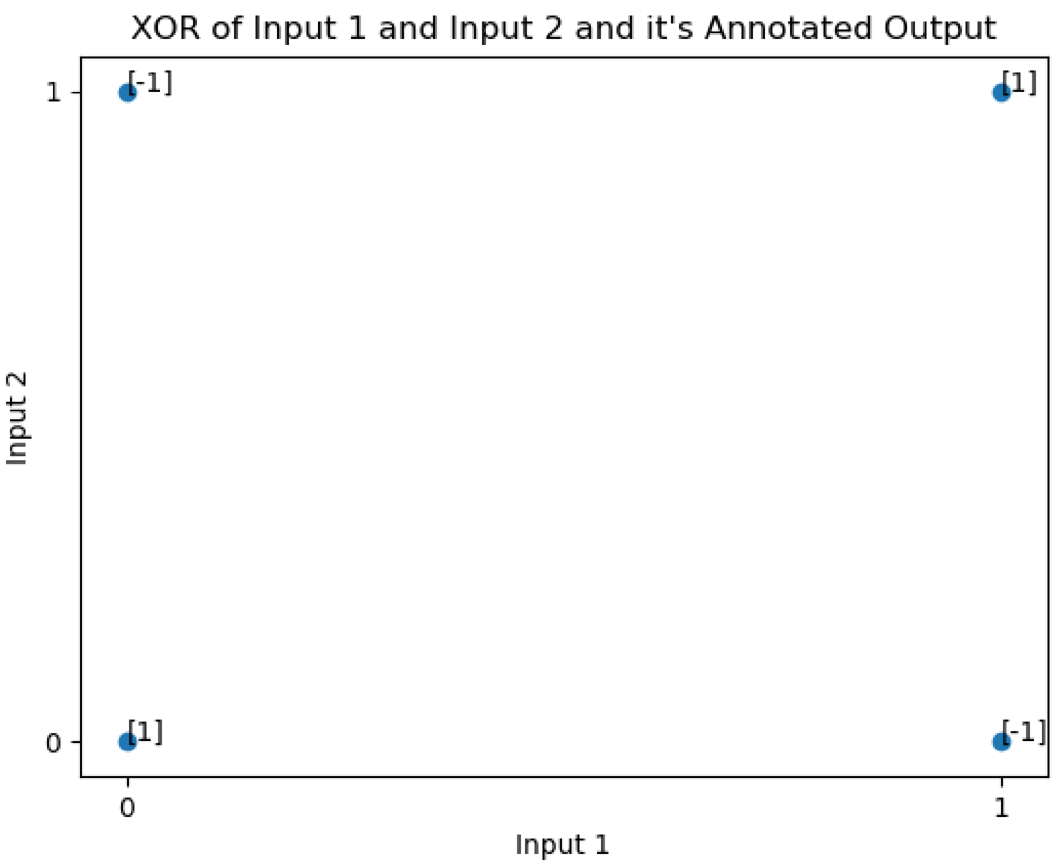

Figure 1: Visualization of XOR inputs and the corresponding outputs

Part b: Network Implementation

The network is defined on lines 155-210 as a class. This is a network with one hidden layer, and an adjustable amount of hidden layer perceptrons. After experimentation on part d with gaussian noise, I ended up with 20 perceptrons in the hidden layer. I chose a sum of squared loss function to train the net due to it's simple derivative, and sigmoids as they perform nicely for two-class classification. The sum of squared loss function is

$$Loss = \frac{1}{2} \sum (\hat{y} - y)$$

where $\hat{y}$ are the predicted labels.

The update rules for gradient descent are not derived here, but are in the code in vectorized form in the *back_prop* function on lines 173-179. Note that tensorflow is not used in this code, and all operations are implemented in numpy.

Part c: Visualizing the Classification Regions

For this section, I chose a slightly brute force approach, and plotted points ranging throughout 0 and 1, colored by their predicted class by the trained neural network. For 20 perceptrons in the hidden layer, the output regions and loss below was produced.

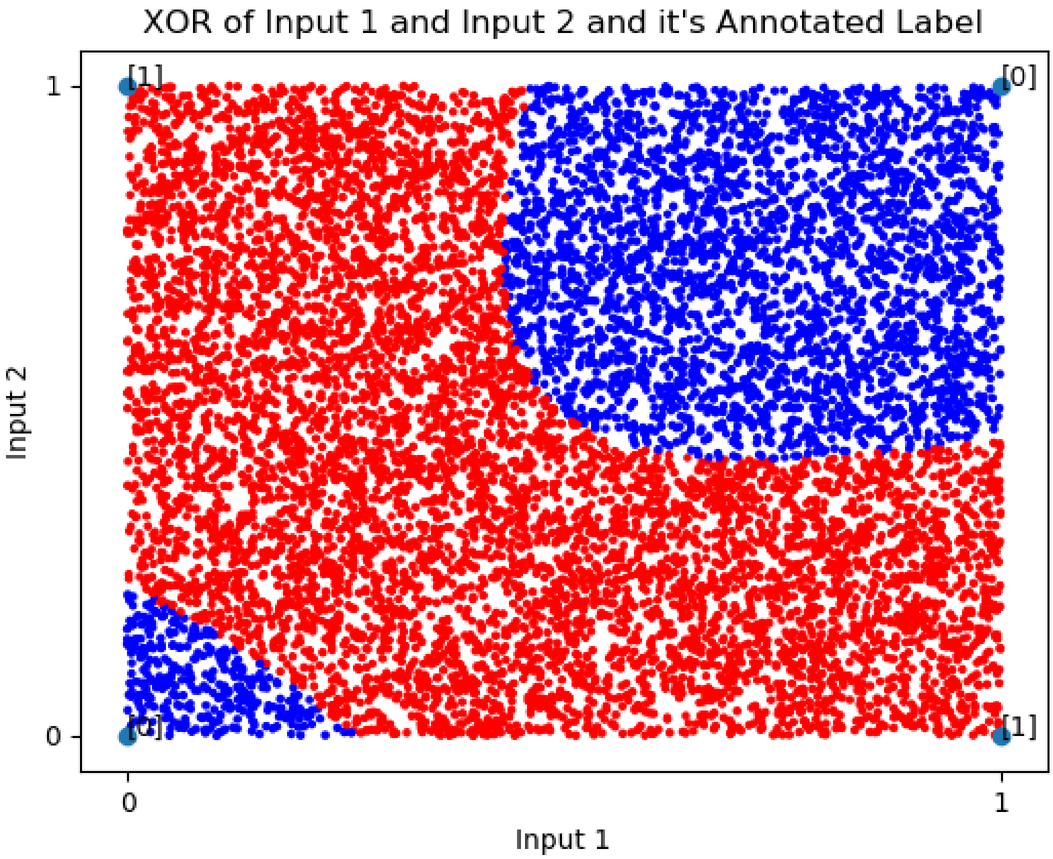

Figure 2: Visualization of the classification regions with 20 perceptrons

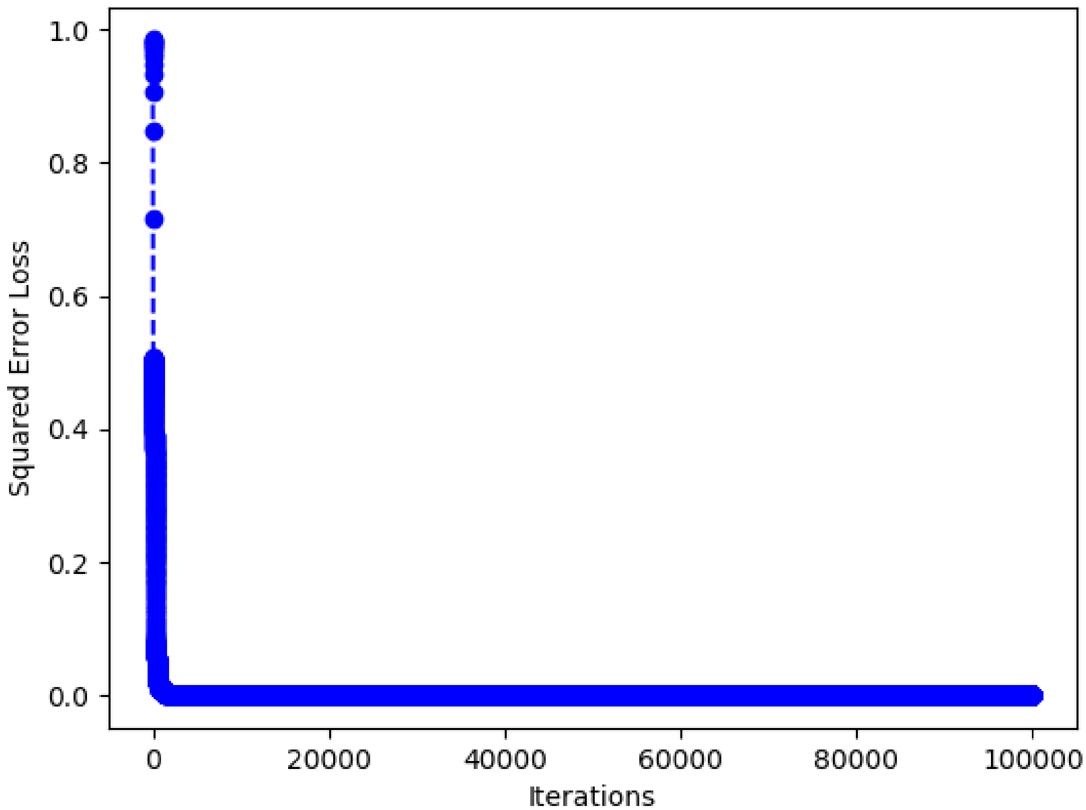

Figure 3: Loss with no noise

Part d: Adding Gaussian Noise

1. $\sigma = 0.5$

Adding noise on line 155 during training with $\sigma = 0.5$ gives the plots below

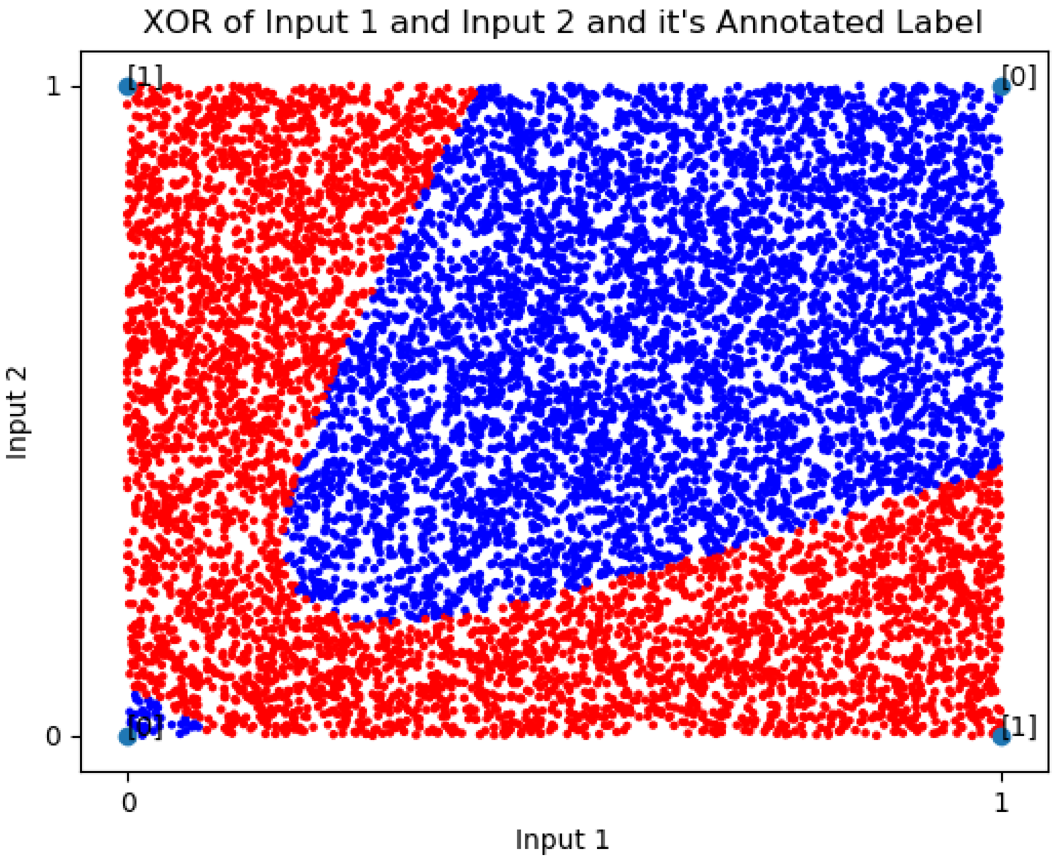

Figure 4: Visualization of the classification regions with 20 perceptrons, $\sigma = 0.5$

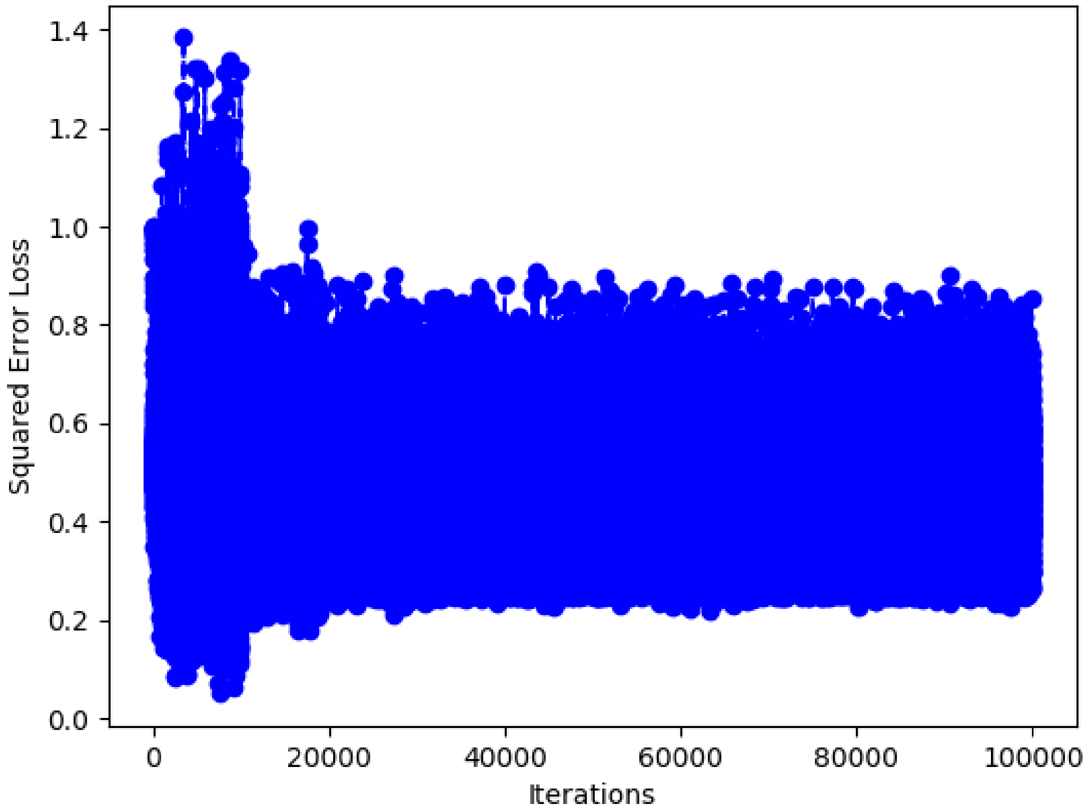

Figure 5: Loss with $\sigma = 0.5$

2. $\boldsymbol{\sigma = 1}$

Adding noise on line 155 during training with $\sigma = 1$ gives the plots below

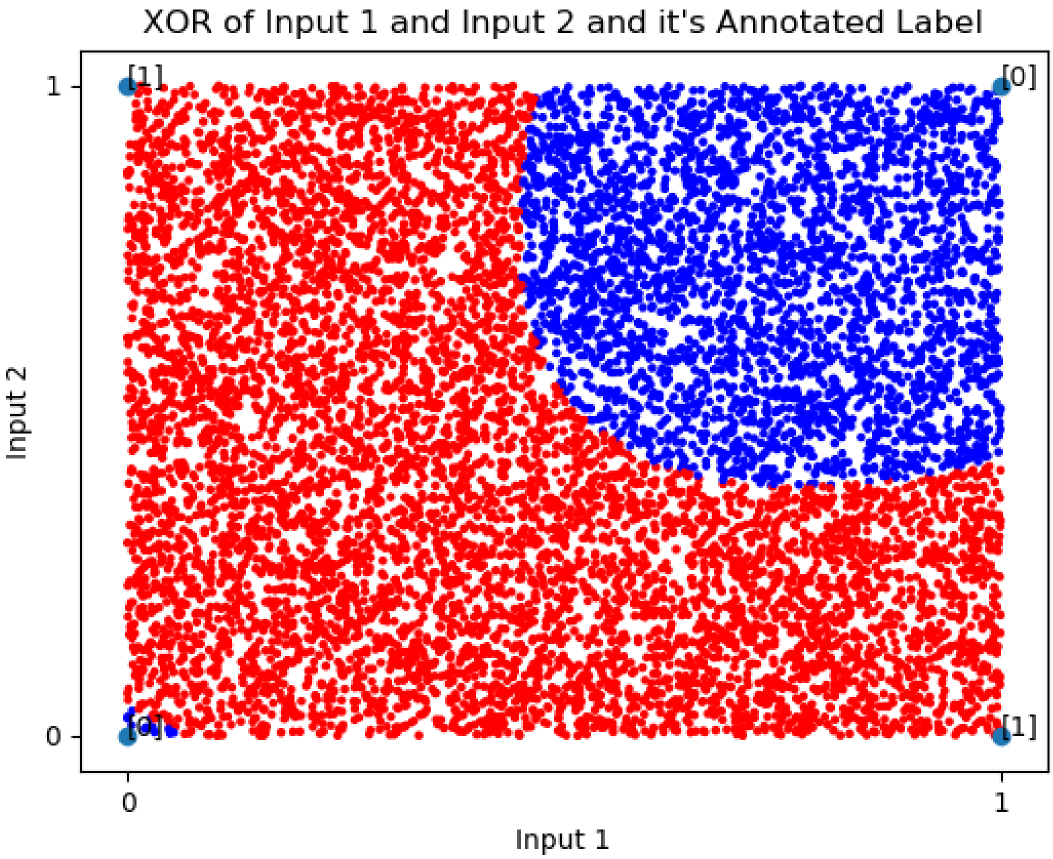

Figure 6: Visualization of the classification regions with 20 perceptrons, $\sigma = 1$

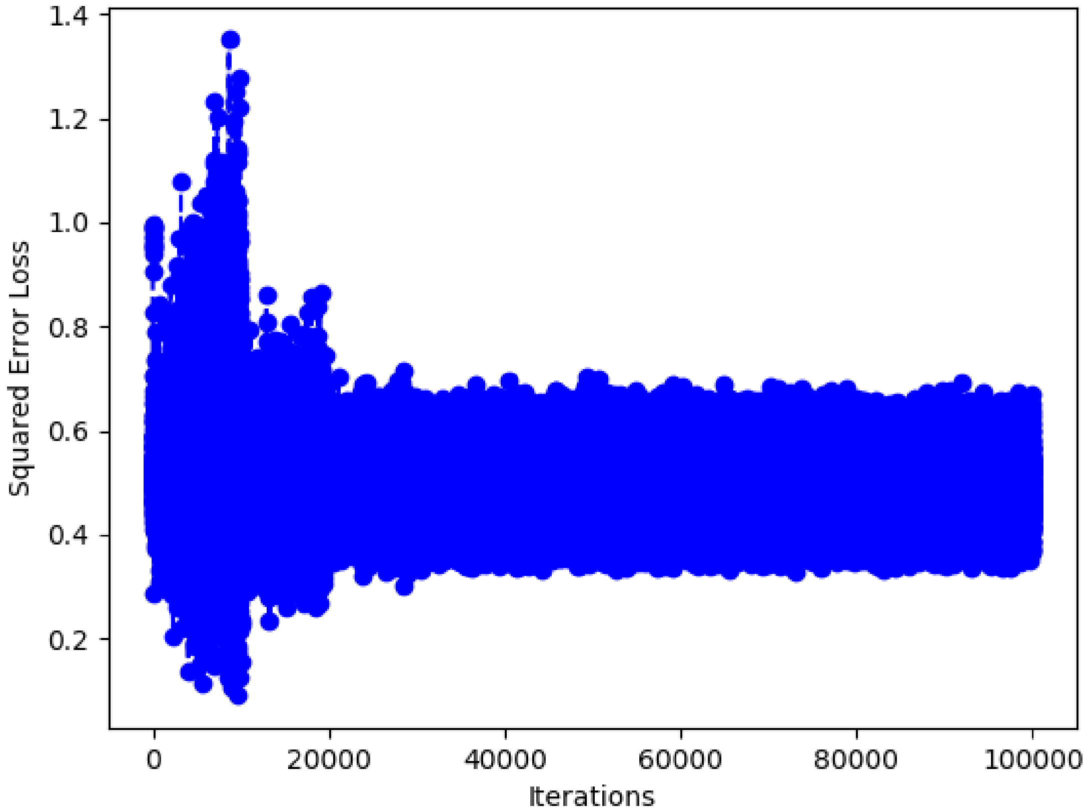

Figure 7: Loss with $\sigma = 1$

3. $\boldsymbol{\sigma = 2}$

Adding noise on line 155 during training with $\sigma = 2$ gives the plots below

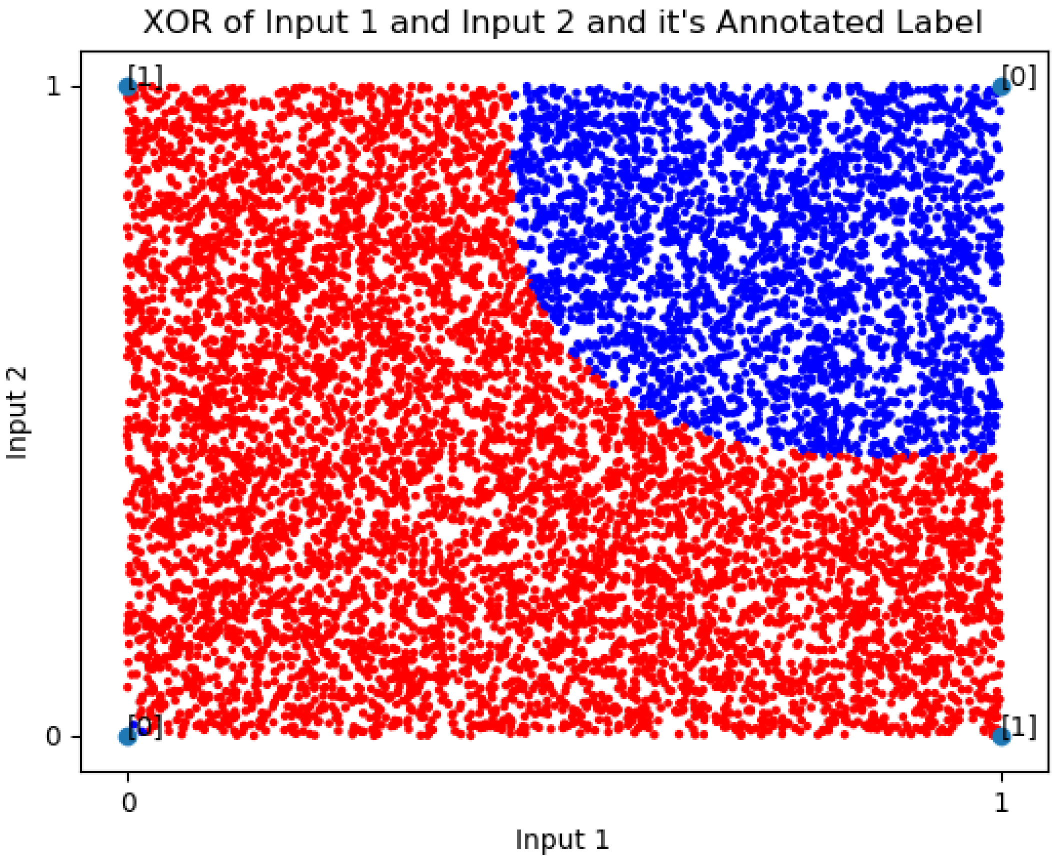

Figure 8: Visualization of the classification regions with 20 perceptrons, $\sigma = 2$

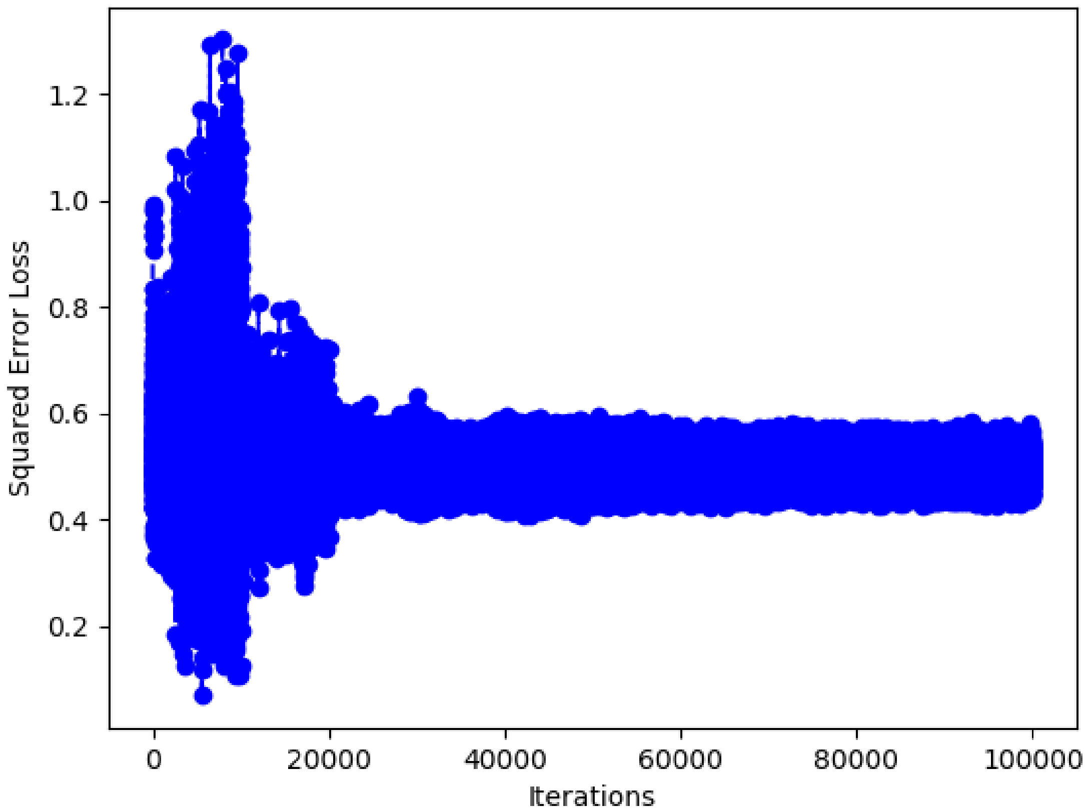

Figure 9: Loss with $\sigma = 2$

Note that the classification region in the bottom left is getting smaller and smaller. To prove that it is still correctly classifying, here is a zoomed in version

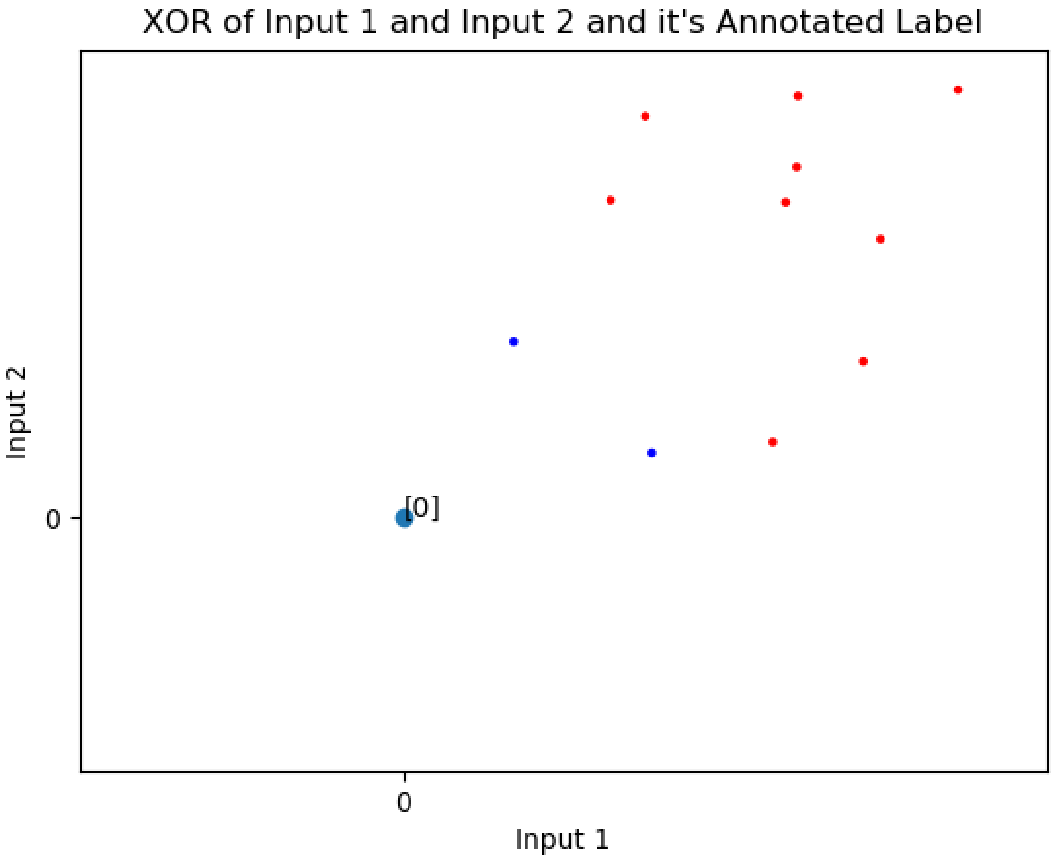

Figure 10: Zoomed in Classification region with $\sigma = 2$