Classification Using Neural Networks

ECE 283 Hw2

May 1, 2020

Student: Ivan Arevalo
Email: ifa@ucsb.com

Department of Electrical and Computer Engineering, UCSB

0 Generating 2D synthetic data for binary classification

We'll start by generating mixed Gaussian samples from 2 classes as described in Figure 1. In order to generate Gaussian random samples, we need to specify the mean and covariance matrix. We can construct the covariance matrix from its eigen-decomposition $C = V\Lambda V^T$, where V is a matrix with eigenvectors as columns and Λ is a diagonal matrix with corresponding eigenvalues. Figure 2 and 3 shows the probability density contour of our generated data and 200 samples from each class.

Class 0: Gaussian with mean vector $\mathbf{m} = (1, -1)^T$ and covariance matrix $\mathbf{C}$ with eigenvalue, eigenvector pairs:
$\lambda_1 = 1$, $\mathbf{u}_1 = (\cos\theta, \sin\theta)^T$, $\lambda_2 = 4$, $\mathbf{u}_2 = (-\sin\theta, \cos\theta)^T$, with $\theta = 0$.
Class 1: Gaussian mixture with two components:
Component A: $\pi_A = \frac{2}{3}$, $\mathbf{m}_A = (-1, 0)^T$, $\mathbf{C}_A$ with eigenvalue, eigenvector pairs: $\lambda_1 = 4$, $\mathbf{u}_1 = (\cos\theta, \sin\theta)^T$, $\lambda_2 = 1/2$, $\mathbf{u}_2 = (-\sin\theta, \cos\theta)^T$, with $\theta = -\frac{3\pi}{4}$.
Component B: $\pi_B = \frac{1}{3}$, $\mathbf{m}_B = (4, 1)^T$, $\mathbf{C}_B$ with eigenvalue, eigenvector pairs: $\lambda_1 = 1$, $\mathbf{u}_1 = (\cos\theta, \sin\theta)^T$, $\lambda_2 = 4$, $\mathbf{u}_2 = (-\sin\theta, \cos\theta)^T$, with $\theta = \frac{\pi}{4}$.

Figure 1: Characteristics of each class

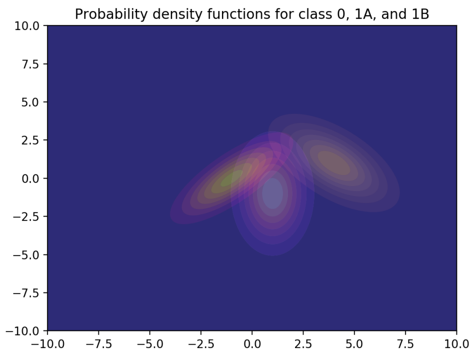

Figure 2: PDF for class 1A (left), 0 (middle), and 1B (right)

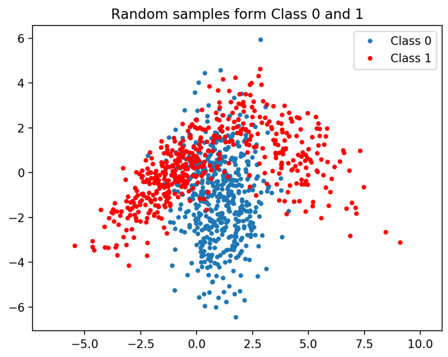

Figure 3: 200 random samples of each class

1 Fully connected neural network

To implement a fully connected neural network we will use Pytorch. We will compare the decision boundaries provided by a neural network side by side with the MAP decision boundaries. We will design neural networks with 1 and 2 hidden layers both with ReLU and Tanh non-linearities with a sigmoid non-linearity at the output. Furthermore we will estimate the overall and conditioned probability of error and compare with the error probability of MAP decision rule, logistic,regression with explicit feature engineering, and kernelized logistic regression.

1.1 Neural network with relu non-linearities

Our first network is made up of a single hidden layer with Relu nonlinearities and softmax output. Both sigmoid and softmax at the outputs had the same performance. Figure 4 shows the architecture of this first network, with two inputs, a hidden layer with 5 neurons and two outputs. From multiple iterations, I found 5 neurons to be the minimum number in order to achieve the highest performance.

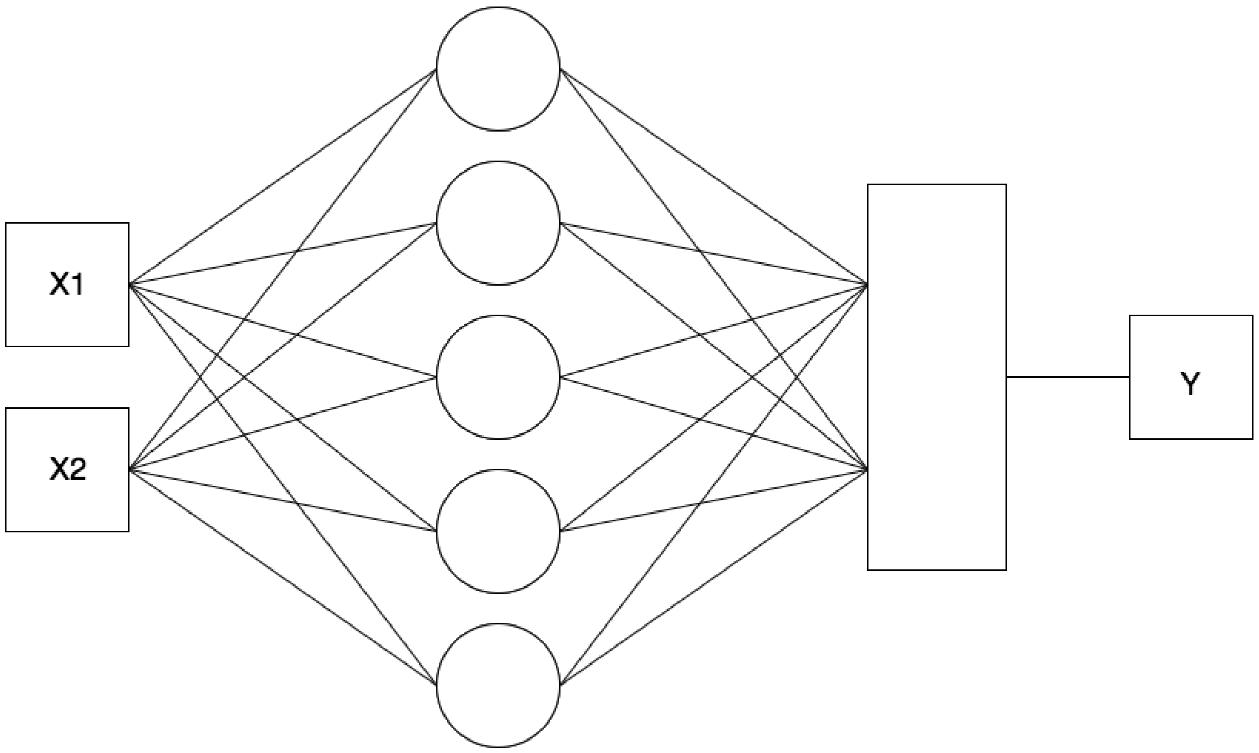

Figure 4: Fully connected neural network with 5 neurons in a single hidden layer

Figure 5 and 6 show the boundary decision for this neural network and the decision boundaries of map rule respectively. Both decision rules were tested with same test data for consistency.

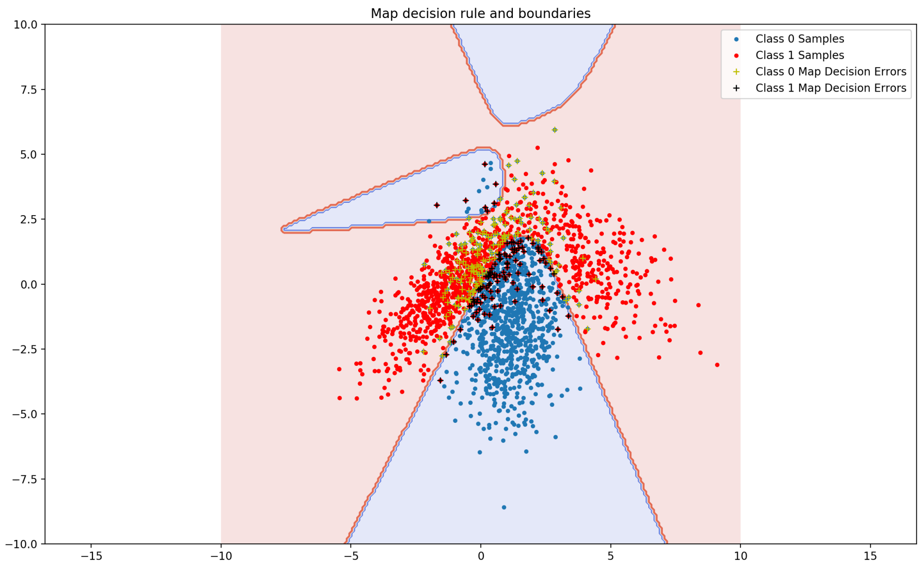

Figure 5: Map decision rule with class 0 and 1 decision boundaries in blue red respectively

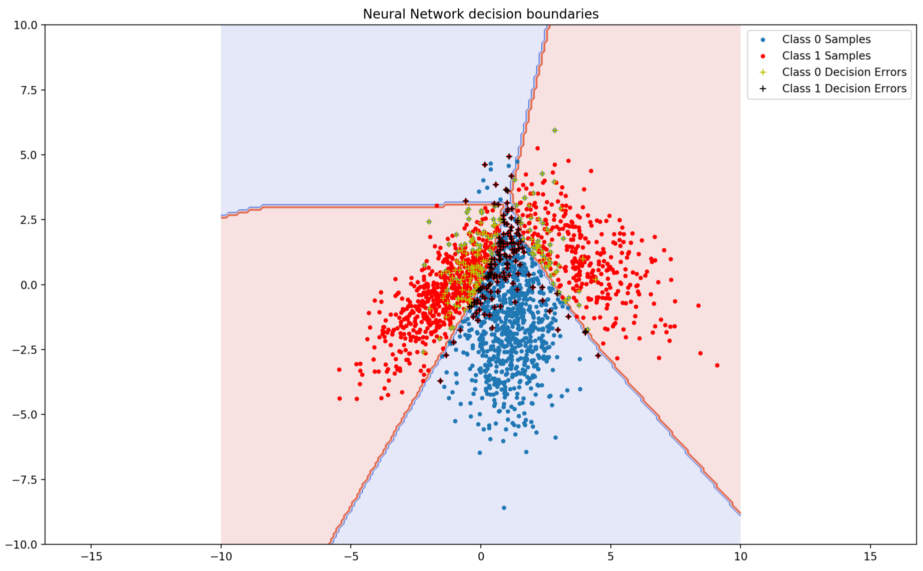

Figure 6: Fully connected neural network decision rule with class 0 and 1 decision boundaries in blue red respectively

We now plot in figure 7 and 10 an alternative sample distribution by modifying the mean of class 0 to (-2, 2) rather than (-1, 1) so the classes become easier to distinguish.

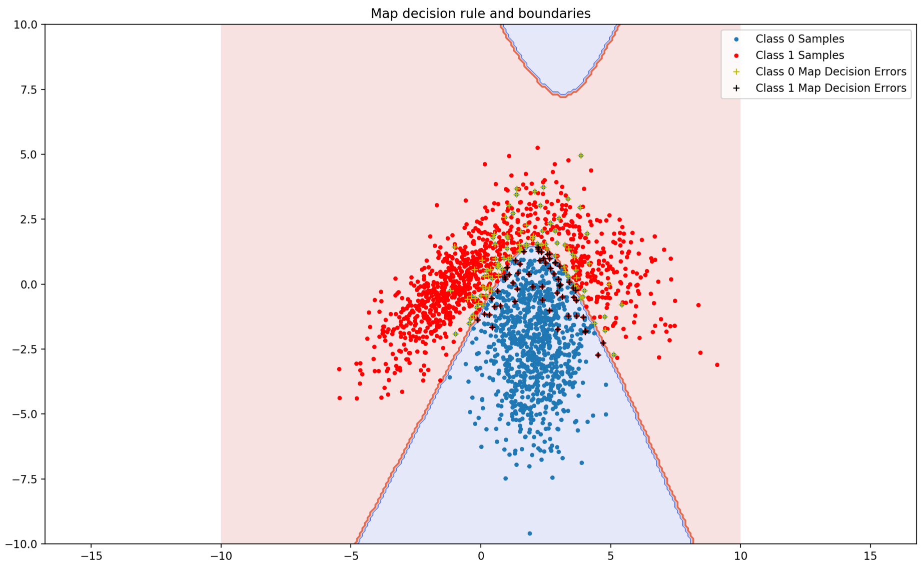

Figure 7: Map decision rule with class 0 and 1 decision boundaries in blue red respectively, for alternate data parameters

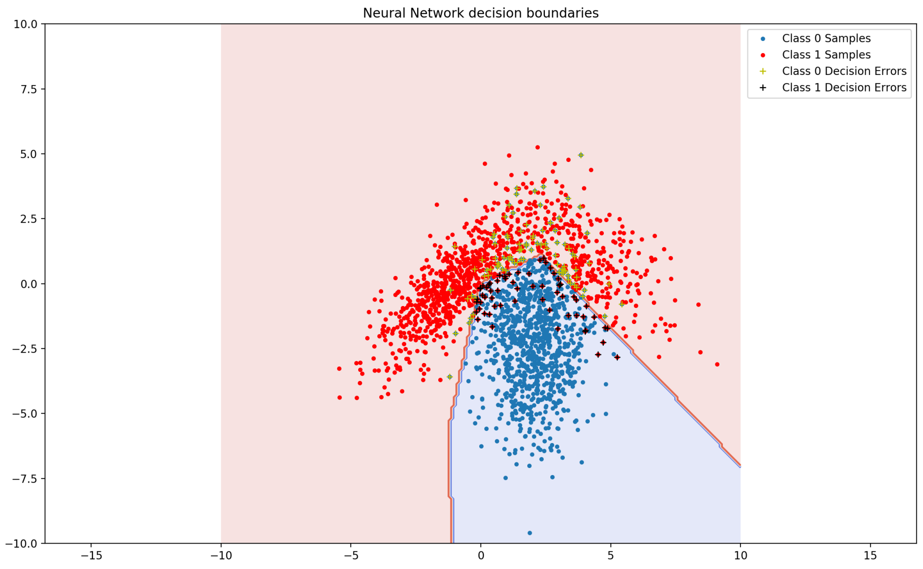

Figure 8: Fully connected neural network decision rule with class 0 and 1 decision boundaries in blue red respectively, for alternate data parameters

We can see that the neural network is able to approximate the boundaries pretty well with a total probability error of 15 % versus 13 % for map rule. Furthermore we notice that the decision boundaries for the neural network seem to be made up of straight lines while map rule decision boundaries have smoother curves. Through multiple iterations, I also noticed that Map rule decision boundaries stay constant while neural network boundaries change each time. This is because the network learns slightly different each time, while conditional probabilities contours are always the same.

Figure 9 shows the architecture of a neural network with 2 hidden layers, 5 neurons in the first layer and 2 neurons in the second layer. It has 2 outputs and calculates decision rule with a softmax layer. I was not able to get a better performance than with the single hidden layer network, even with a high number of neurons. I found five and two neurons in the first and second layers to be the minimum number of neurons necessary to have an accuracy of 85 %. For brevity, we will now only consider the original data set and the neural network decision boundaries and compare them to figure 5.

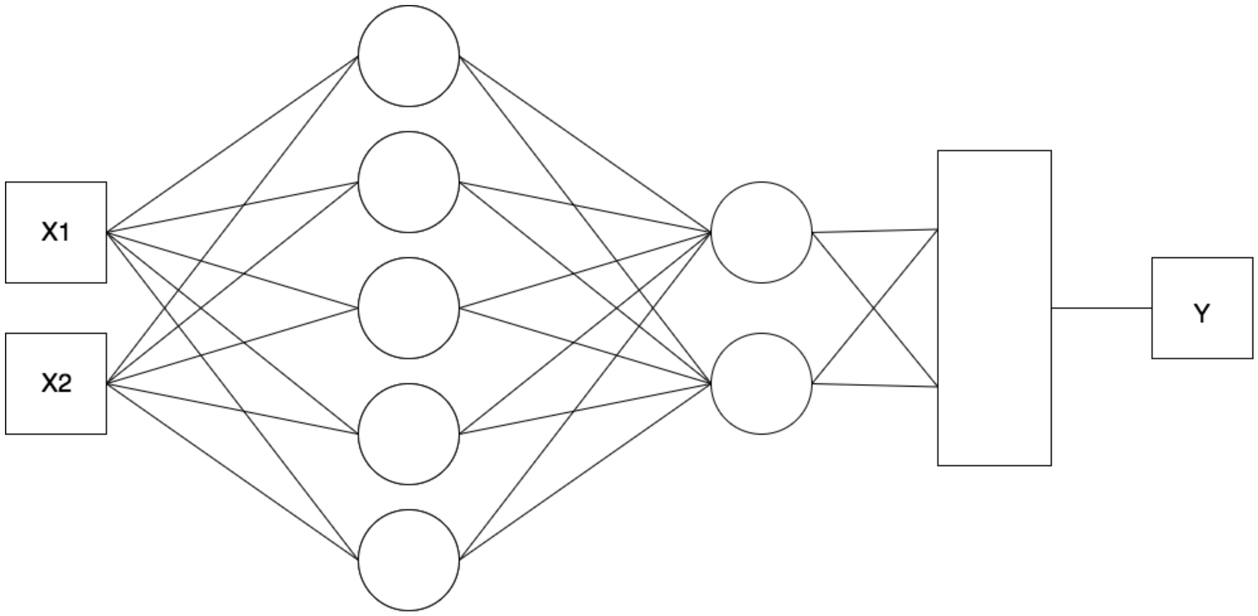

Figure 9: Fully connected neural network with 5 neurons in first hidden layer and 2 neurons in the second hidden layer

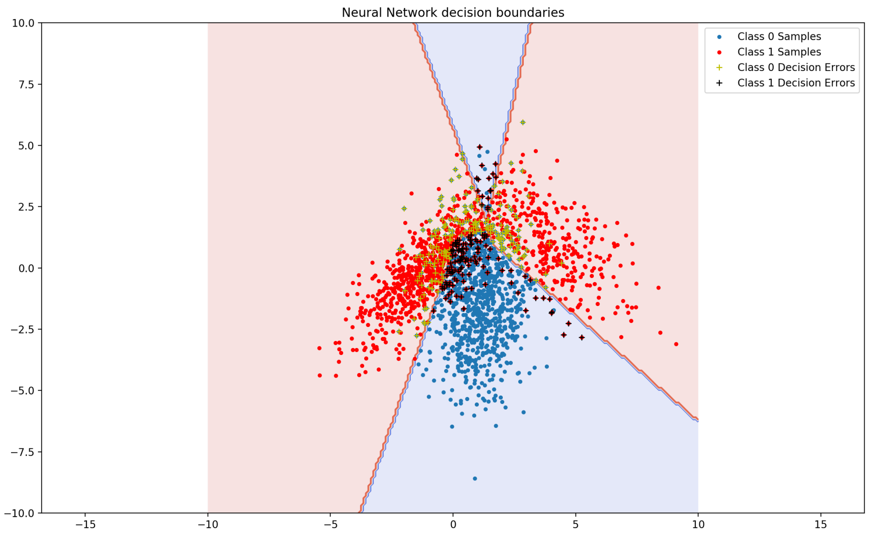

Figure 10: 2 layer neural network decision rule with class 0 and 1 decision boundaries in blue red respectively

This neural network performed exactly the same as the one-layer network in figure 4 with a total error of 15 %.

Comparing neural network accuracy We can estimate the probability of misclassification by keeping keeping track of the number of errors and dividing by the number of total test samples. I tested each network with a 1000 samples from each class and found the following probability errors. The single hidden layer network had a percent error of 14.9 and 16.1 for class 0 and 1 respectively and a total of 15 %. We can compare these error probabilities against Map rule, logistic regression, and kernelized logistic regression from last assignment to asses how our network is performing. Map rule had a percent error of 16.8 and 10.6 for class 0 and 1 respectively and a total of 13.7 %. Logistic regression had a percent error of 16 and 19 for class 0 and 1 respectively and a total of 17 %. Finally kernelized logistic regression had a percent error of 15.2 and 13.7 for class 0 and 1 respectively and a total of 14.45 %. We can conclude that our neural network compares to these other methods, coming in third behind Map rule and kernelized logistic regression.

Hyperparameters In order to train these networks I tested different number of training samples and number of batches. I created a dataset class in order to easily shuffle training data and organize batches. I started with 1000 samples from each class and built up from there. I found that 5000 samples from each class organized into batches of 100 performed well in the 5-node single-layer network, and any more than that yielded the same performance. For validation, I used 1000 samples from each class and found 50 epocs to be a good amount of iterations to train the network with a 0.01 learning rate. I tried many different combinations and these seemed to work best. Furthermore, I tested the network by letting a while loop determine the amount of epocs to train for by calculating the difference in loss function from two consequetive epocs and comparing against an epsilon value. Expectedly, the classification error dropped most dramaticaly in the first 25 epocs and leveled off after that.

1.2 Neural network with tanh non-linearities

We now create single and double layer networks as depicted in figure 4 and 9 but replace the relu non-linearities for tanh. Figures 11 and 12 show the performance and decision boundaries for each of these networks.

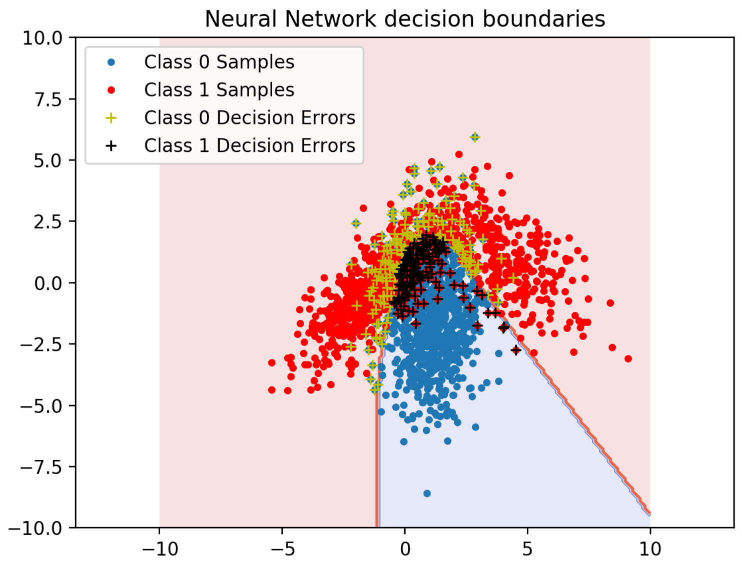

Figure 11: Fully connected neural network with 5 neurons in first hidden layer and tanh non-linearities

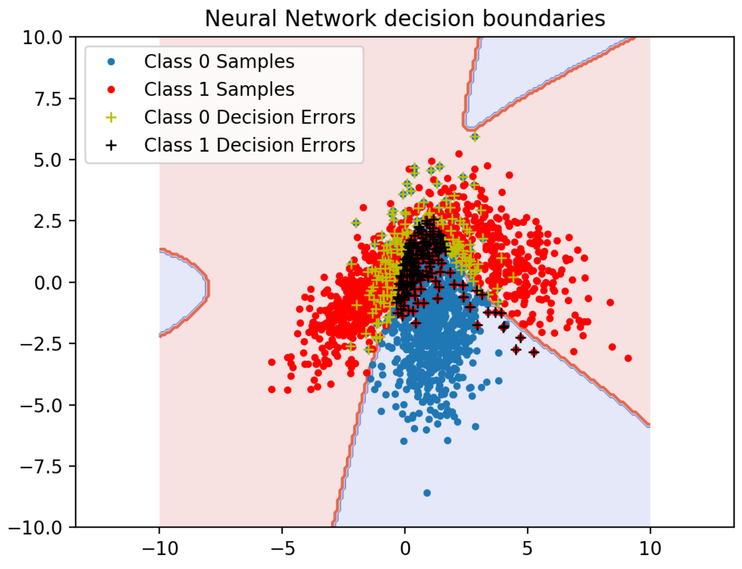

Figure 12: Fully connected neural network with 5 neurons in first hidden layer and 2 neurons in the second hidden layer and tanh non-linearities

Comparing neural network accuracy Interestingly, the decision boundaries with tanh non-linearities seem to be smoother than with relu non-linearities and more closely resemble the Map rule decision boundaries. Again, I tested each network with a 1000 samples from each class and found the following probability errors. The single hidden layer network had a percent error of 16.2 and 14.0 for class 0 and 1 respectively and a total of 15 %. Meanwhile, the network with two hidden layers had a percent error of 16.4 and 15.7 for classes 0 and 1 respectively for a total error of 16 %. Again, compared with Map rule, logistic regression, and kernelized logistic regression, it compares in performance. Interesting enough, I found the single layer network to consistently perform better than the two layer network.

Hyperparameters As before, I started with 1000 samples from each class and built up from there finding 5000 samples from each class organized into batches of 100 to perform best in. For validation, I used 1000 samples from each class and found 50 epocs to be a good amount of iterations to train the network with a 0.01 learning rate.

1.3 Analysis

In order to build a successful network, I had to tune different parameters in order to find the best outcome. I experimented with the learning rate quite a bit and found that for gradient descent, a learning rate from 0.01 to 0.1 worked best. A higher learning rate would result in over-stepping while a smaller learning rate would require too many epocs to train successfully. I also tested Adam optimizer which I found needed a smaller learning rate than GSD. Moreover, I found that L2 weight regularization didn't help the performance of the network when using GSD but it did help when using Adam optimizer. I also tested a few different batch sizes and found batches of size 100 worked best. Again any lower or over started to decrease the classification performance. Pre-processing didn't seem to make much a difference in any of the networks. I hypothesize that because it is a relatively small data set, the benefits can't be seen as opposed to regularizing images. I tried both uniform and normal weight initialization but again didn't see any change in performance. I believe it would make a difference if the number of training samples and network size was smaller, but given that the current number of parameters train the data very fast, I didn't see a need to decrease them. As mentioned earlier, 5 neurons in the first layer and 2 in the second resulted in the same performance than many other combinations with more neurons. There are many different knobs to turn when training neural networks and a more systematic approach to find the correct parameters would save a lot of time. Automating this process by iterating through random parameter combinations might be a better approach in the future. In conclusion, we observed the performance of neural networks in a very simple setting and proved it compares to the performance of MAP rule, logistic regression, and kernelized logistic regression.

2 Code

```
import torch
from torch.utils.data import Dataset
import sample_generation
import map_rule
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class GaussianDataSet(Dataset):
    def __init__(self, points, labels=None, transforms=None):
        self.X = points
        self.y = labels
        self.transforms = transforms

    def __len__(self):
        return len(self.X)

    def __getitem__(self, i):
        data = self.X[i, :]
        data = torch.from_numpy(data)

        if self.transforms:
            data = self.transforms(data)

        if self.y is not None:
            return data, self.y[i]
        else:
            return data

def run_relu_network(num_samples, layer1_size, layer2_size=2,
    plot_decision_boundaries=False):

    class GaussianNetwork(nn.Module):
        def __init__(self, layer1_size, layer2_size=2):
            super(GaussianNetwork, self).__init__()
            self.fc1 = nn.Linear(2, layer1_size)
            if layer2_size != 2:
                self.fc2 = nn.Linear(layer1_size, layer2_size)
                self.fc3 = nn.Linear(layer2_size, 2)
            else:
                self.fc2 = nn.Linear(layer1_size, 2)

        def forward(self, x):
            x = self.fc1(x)
            x = F.relu(x) # Replace with tanh for part 2
            x = self.fc2(x)
            if hasattr(self, 'fc3'):
```

```
            x = F.relu(x) # Replace with tanh for part 2
            x = self.fc3(x)
        return x

# generate train data
class_0, class_1, class_1A, class_1B = \
    sample_generation.generate_samples(num_samples, seed0=255, seed1A=400, seed1B=545)
samples, map_decision, data_labels, error_probability_class_0,
    error_probability_class_1 = \
        map_rule.evaluate_map_rule(num_samples, class_0, class_1, class_1A, class_1B)

trainset = GaussianDataSet(samples, data_labels)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=100, shuffle=True)

# Display sample distribution
map_decision_contour, XX, YY = map_rule.display_decision_boundary(class_0, class_1,
    class_1A, class_1B)

def weights_init(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            # initialize the weight tensor, here we use a normal distribution
            m.weight.data.normal_(0, 1)
            # m.weight.data.uniform_(0, 1)

net_1_layer = GaussianNetwork(layer1_size, layer2_size)
net_1_layer.apply(weights_init)
print(net_1_layer)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net_1_layer.parameters(), lr=0.01, weight_decay=0)

net_1_layer.train()

for epoch in range(50): # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data[0], data[1]

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net_1_layer(inputs.float())
        labels = labels.squeeze(1)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 100 == 99: # print every 5 mini-batches
            print('[%d, %5d] loss: %.5f' %
                (epoch + 1, i + 1, running_loss/100))
            running_loss = 0.0

print('Finished Training')

if plot_decision_boundaries:
    num_pts = 200
    limit = 10
    X = Y = np.linspace(-limit, limit, num_pts)
    XX, YY = np.meshgrid(X, Y)
    decisions = np.zeros([num_pts, num_pts])
    grid_samples = np.c_[np.ravel(XX), np.ravel(YY)]
    testset = GaussianDataSet(grid_samples)
    testloader = torch.utils.data.DataLoader(testset, batch_size=num_pts*num_pts,
                                             shuffle=False)
    decisions = np.ravel(decisions)
    with torch.no_grad():

        for data in testloader:
            test_points = data
            outputs = net_1_layer(test_points.float())
            _, decisions = torch.max(outputs.data, 1)

    decisions = decisions.numpy().reshape(XX.shape)

    plt.figure(2)
    plt.contour(XX, YY, decisions, alpha=1, cmap=plt.cm.coolwarm, antialiased=True)
    plt.contourf(XX, YY, decisions, alpha=.15, cmap=plt.cm.coolwarm, antialiased=True)
    plt.title('Neural Network decision boundaries')

    class_0, class_1, class_1A, class_1B = sample_generation.generate_samples(1000,
        seed0=255, seed1A=400,
                                                                         seed1B=545)
    samples, map_decision, data_labels, error_probability_class_0,
        error_probability_class_1 = \
        map_rule.evaluate_map_rule(1000, class_0, class_1, class_1A, class_1B)
    # Test model
    testset = GaussianDataSet(samples, data_labels)
    testloader = torch.utils.data.DataLoader(testset, batch_size=2000,
                                             shuffle=False)

    plt.plot(class_0.data[:, 0], class_0.data[:, 1], '.', alpha=1)
    plt.plot(class_1[:, 0], class_1[:, 1], 'r.', alpha=1)

    net_1_layer.eval()

    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            test_points, labels = data
            outputs = net_1_layer(test_points.float())

            _, predicted = torch.max(outputs.data, 1)
```

```
                total += labels.size(0)
                correct += (predicted == labels.squeeze()).sum().item()
                decision_errors0 = np.not_equal(0, predicted[:1000].numpy())
                decision_errors1 = np.not_equal(1, predicted[1000:].numpy())
                break

        class_0_errors = samples[np.flatnonzero(decision_errors0)]
        class_1_errors = samples[np.flatnonzero(decision_errors1) + 1000] # Foward index
            to second half of samples

        plt.plot(class_0_errors[:, 0], class_0_errors[:, 1], 'y+')
        plt.plot(class_1_errors[:, 0], class_1_errors[:, 1], 'k+')
        plt.axis('equal')
        plt.legend(['Class 0 Samples', 'Class 1 Samples', 'Class 0 Decision Errors',
            'Class 1 Decision Errors'])

        print('Percent error of Class 0:', 100*class_0_errors.shape[0]/1000)
        print('Percent error of Class 1:', 100*class_1_errors.shape[0]/1000)
        print('Total accuracy of the network on the 10000 test images: %d %%' % (100 * (1
            - correct / total)))

        plt.show()


if __name__ == '__main__':
    number_of_training_samples_per_class = 5000

    run_relu_network(number_of_training_samples_per_class, 5,
        plot_decision_boundaries=True)
    # run_relu_network(number_of_training_samples_per_class, 5, 2,
        plot_decision_boundaries=True)
    # run_tanh_network(number_of_training_samples_per_class, 5)
    # run_tanh_network(number_of_training_samples_per_class, 5, 2)
```