ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Problem Set II

Karan Chawla
karangc2@illinois.edu

Course Instructor: **Prof. G. Chowdhary**

March 29, 2017

1 Solution 1

1.1 Part 2

The [1] learned weights after linear regression are shown below:

x	Real Weight	Learned Weight	Percent Error
1	0.8	0.8020	0.250
ϕ	.2314	0.2300	0.605
p	0.6918	0.7045	1.835
$\|\phi\|$p	-0.6245	-0.6384	2.225
|p|p	0.0095	0.0140	47.36
ϕ^3	0.0214	0.0211	1.401

1.2 Part 4

Data	Initial Conditions	MSE
Training Data	(x,y) = (1.2,1), (3,6), ((3-1.2)/2,(6-1)/2)	2.53
Test Data	(x,y) = ((3-1.2)/2,(6-1)/2)	50.6730

Table 1.1: Results

```
%% simple GP regression example
% author: Girish Chowdhary,
% using publically available code on
% http://web.mit.edu/girishc/www/resources/resources%20files
% /Module_4_Nonparameteric_Adaptive_Control.zip
% Thanks to Hassan Kingravi for writing a lot of the onlineGP function

clear all
close all

%% load input data
load('data.mat', 'data') % gp_regression_example_data
% the only data we want for this example is 2 inputs X1_IN, X2_IN, and an
% output Y1_OUT. Example data is loaded by the above program

THETA=data(:, 1);
P=data(:, 2);
U=data(:, 3);
PDOT=data(:, 4);
```

[1] This problem set was solved with Wyatt McAllister

```
%% Linear Regression
A=zeros(length(data), 6);
A(:,1)=1;
A(:,2)=data(:,1);
A(:,3)=data(:,2);
A(:,4)=abs(data(:,1)).*data(:,2);
A(:,5)=abs(data(:,2)).*data(:,2);
A(:,6)=(data(:,1)).^3;
b=data(:,4)- data(:,3);
w=A\b;
err=b-A*w;

%% invoke GP regression model object
%% Gaussian Process parameter settings and initialization
%parameters
bandwidth = 100;% this is the bandwidth sigma of the square exponential
                % kernel we are using, the kernel is given by
% k(x1,x2)=exp(-norm(x1-x2)^2/bandwidth^2
noise = 1;      % this is the noise we assume that our data has, right now
                % its set to 1, this needs to be inferred from data,
                % there are data driven techniques available to do this
                % (see Rassmussen and Williams 2006 )
tol = 0.00001; % this is a parameter of the sparsification process,
                % smaller results in the algorithm picking more kernels
                % (less sparse)
                % This has been explained in Chowdhary et al. ACC 2012
                % (submitted)
max_points=100; % this is our budget of the kernels, we allow 100 here
                % with real data this number needs to be tuned,
                % althoug the regression is not too sensitive to it
                % with appropriate bandwidth its mostly for limiting
                % computationl effort

% the following line invokes the gp regression class
gpr = onlineGP(bandwidth,noise,max_points,tol);
% the goal is to learn a generative model from data
% let the mean function be f(x_in), the measuremente are y, and
% they are y =f(x_in)+noise_function(noise), currently our
% model of noise is white noise
% The KL diverence based sparsification method developed by Csato et
% al. is used, and is referenced in the class
% the GPR function has several subfunctions, which are called as
% initialization function: gpr.process(x_in,y)
% regression/learning function gpr.update(x_in,y)
% prediction function gpr.predict(x_in)
% model saving function gpr.save('model_name');
% model loading function gpr.load('model_name')
% get internal variables: gpr.get('var_name'), var_names are documented
% in the function object itself, the main ones are:
```

```
% The GP current basis: 'basis' or 'BV'
% Current set of active observations: 'obs'
% Current set of kernels: 'K','kernel'
% Current size of active basis set: 'current_size' or 'size' or
% 'current size'
% see more definitions in the class itself

%% loop through data to learn
for ii=1:max(size(PDOT))
    x_in=[THETA(ii);P(ii);U(ii)];
    if ii == 1
        % if first step, initialize GP
        gpr.process(x_in,PDOT(ii));
    else
        gpr.update(x_in,PDOT(ii));
    end
end

%% now we can predict
load('datanew.mat', 'data')
THETA_new=data(:, 1);
P_new=data(:, 2);
U_new=data(:, 3);
PDOT_new=data(:, 4);

% define the grid over which we are going to predict
range_size=max(size(PDOT_new));

%%
% loop through the grid to get the predicted values NEW DATA
for ii=1:range_size
       x_in=[THETA_new(ii);P_new(ii);U_new(ii)];
       [mean_post, var_post] = gpr.predict(x_in);
       EST_MEAN_POST_GP(ii)=mean_post;
       EST_VAR_POST_GP(ii)=var_post;
end

e=(PDOT_new-EST_MEAN_POST_GP');
MSE_tot=mean(e.^2);
RMSE_tot=sqrt(mean(e.^2));
figure(1)
Samp=1:1:length(PDOT_new);
plot(Samp,EST_MEAN_POST_GP,Samp,PDOT_new);
legend('y','yd');
ylabel('y^d,y');
xlabel('Sample');
title('y Vs. y^d');
grid on;
```

```
figure(2)
Samp=1:1:length(PDOT_new);
plot(Samp,e);
title('Error');
xlabel('Sample');
ylabel('MSE');
grid on;
```

2 Solution 2

The average run time of the following code was 0.187 seconds on an i7 5th generation computer.
The following figures show the clusters with their respective means around which the data is assumed to be centered. The x and y coordinates of the means are plotted as histograms and can be viewed in the Results section.

2.1 Compute Centroids

```
%% This function calculates the centroids for the complete data set once the
%% indices have been assigned to each point. (Auxillary function)
%% Input: Data and sum
%% Return: New Centroids
function post_mean = computeCentroids(sum,X,Y)
        % Go over every example, find its closest centroid, and store
        %the index inside idx at the appropriate location.
        [~, idx] = sort(sum);
        sum_sorted = idx(1,:);
        posterior_sum = zeros(2,5);
        ctr = ones(1,5);
        %go over each data point
        for j=1:500
            for t = 1:5
                %check which idx it has been assigned to
                if sum_sorted(j)==t
%                   Go over every centroid and compute mean of all points that
%                   belong to it.
                    posterior_sum(1,t) = posterior_sum(1,t) + X(j);
                    posterior_sum(2,t) = posterior_sum(2,t) + Y(j);
                    ctr(1,t) = ctr(1,t)+1;
                end
            end
        end
        post_mean(1,:) = posterior_sum(1,:)./ctr;
        post_mean(2,:) = posterior_sum(2,:)./ctr;
end
```

2.2 Euclidean Distance

```
%% Auxillary function to calculate the Euclidean distance between each centroid
%% and data point.
%% Function input: Data and prior centroid estimates
%% Return: Distance
function sum = summation(X,Y,prior_mean,j)
sum = (X - prior_mean(1,j)).^2 + (Y - prior_mean(2,j)).^2;
end
```

2.3 EM Gaussian Mixture Model Setup

```
%% Main function for Problem 2
%gaussian mixture model clusturing with EM algo
% see Tomasi's notes

clear all
close all
%%
load gauss_mix_data

k=max(size(means));% parameteric method, so number of parameters known
N=max(size(X));
prior_mean=10*randn(2,5);
prior_var=vars*0+1;
rho=[0.8 0.1 0.05 0.025 0.025];
rho_post=rho;

posterior_mean=prior_mean;
posterior_var=prior_var;

epochs=100;

%% your code here
for k=1:20
    for i=1:epochs
        for j=1:5
            sum(j,:)= summation(X,Y,prior_mean,j);
        end
        %assign data points to clusters
        posterior_mean = computeCentroids(sum,X,Y);
        prior_mean = posterior_mean;

    end
    posterior_mean_TOT_1(k,:) = posterior_mean(1,:);
    posterior_mean_TOT_2(k,:) = posterior_mean(2,:);
end
```

```
figure(1)
histogram(posterior_mean_TOT_1,'BinMethod','integers');
figure(2)
histogram(posterior_mean_TOT_2,'BinMethod','integers');

%%
figure(3)
plot(x(:,1),y(:,1),'or')
hold on
plot(x(:,2),y(:,2),'*g')
hold on
plot(x(:,3),y(:,3),'+m')
hold on
plot(x(:,4),y(:,4),'.k')
hold on
plot(x(:,5),y(:,5),'+y')
%% this plots the posterior mean and variance
% if your inference works, you should be able to see the right allocation
for kk=1:5
    plot_gaussian_ellipsoid(posterior_mean(:,kk),diag([posterior_var(kk),posterior_var(kk)]));
end
%plot(posterior_mean(1,:),posterior_mean(2,:),'*')
```

2.4 Results

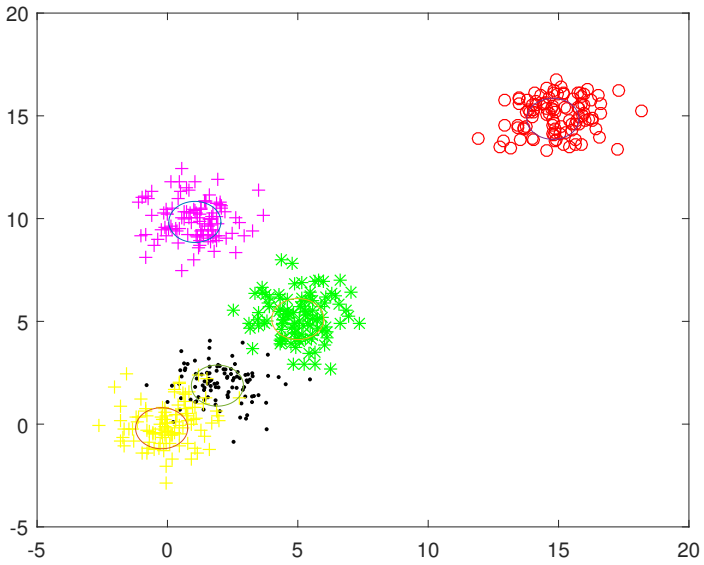

Figure 2.1: Clustered output data

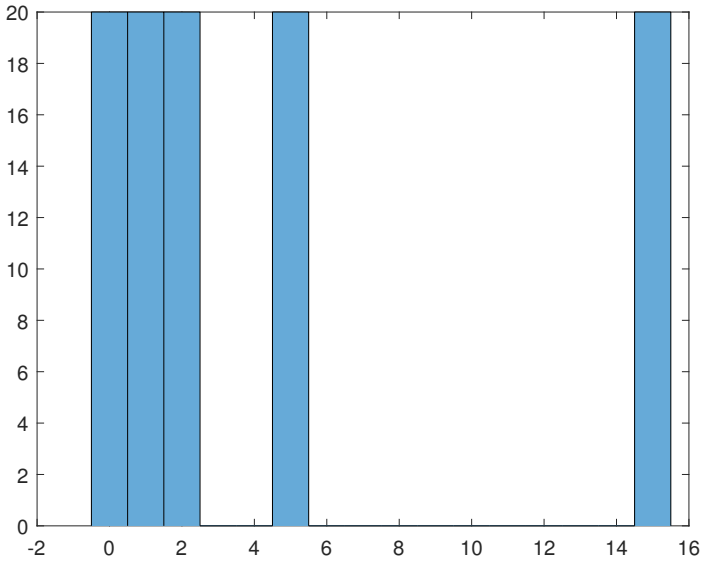

Figure 2.2: Means-X

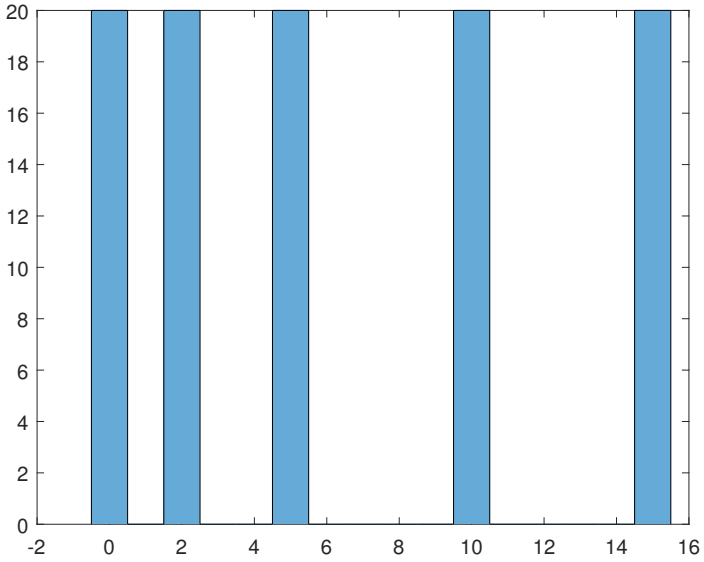

Figure 2.3: Means-Y

3 Solution 3

Single hidden layer neural network performs better than GP as the MMSE for the former case is about 0.04 whereas the GP gives an MMSE of about 0.25. So, the difference in the error is by a magnitude of 10.
Overall the performance of the neural network improves with the increase in the number of neurons in the hidden layer. This can be observed in the results section below. As can be seen in some of the graphs, the error increases slightly after reaching a minimum, this could be due to the over-fitting. This problem may be solved by adding dropout, or adding the weights to be learned in the cost function and penalizing higher weights.

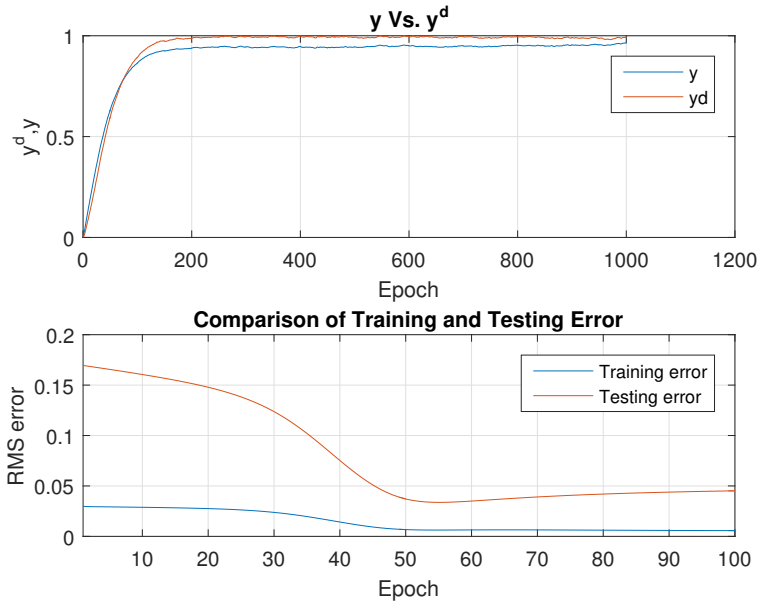

Figure 3.1: Neural Network Performance for ten neurons

3.1 Compute Cost

```
function J = computeCost(y,yd)
% J = COMPUTECOST(Y,YD) computes the cost of the output.

J = (y - yd)^2;
end
```

3.2 Sigmoid Function

```
function g = sigmoid(z)
```

```
% J = SIGMOID(z) computes the sigmoid of z.

g = 1.0 ./ (1.0 + exp(-z));
end
```

3.3 Compute Gradient

```
function grad = sigmoidGradient(x)
% J = SIGMOID(z) computes the gradient of sigmoid(x).
grad = x*(1-x);
end
```

3.4 Update Weights

```
function [Theta2, Theta1] = updateWeights(X, y, out, Theta1, Theta2, k, j,
    eta, v1)

%updating the weight for the outermost layer
delta = y(1,j) - out;
Theta2(1,k) = Theta2(1,k) + eta*(delta*v1(k));

%Weight update for hidden-input layers
Theta1(1,k) = Theta1(1,k) + eta*(sigmoidGradient(v1(k))*(y(j) -
    out)*Theta2(1,k)*X(1,j));
Theta1(2,k) = Theta1(2,k) + eta*(sigmoidGradient(v1(k))*(y(j) -
    out)*Theta2(1,k)*X(2,j));
Theta1(3,k) = Theta1(3,k) + eta*(sigmoidGradient(v1(k))*(y(j) -
    out)*Theta2(1,k)*X(3,j));

end
```

3.5 Normalize Features

```
function [X_norm, mu, sigma] = featureNormalize(X)
%FEATURENORMALIZE(X) normalizes features around the mean
mu = mean(X);
X_norm = bsxfun(@minus, X, mu);

sigma = std(X_norm);
X_norm = bsxfun(@rdivide, X_norm, sigma);

end
```

```
%11th March '17

%initialization
clc;
clearvars;
close all;

%setup the parameters to be used
load ('data.mat' , 'data');
theta = data(:,1);
p = data(:,2);
u = data(:,3);
p_dot = data(:,4)';

X(1,:) = theta;
X(2,:) = p;
X(3,:) = u;
y = p_dot;

input_layers = 3;
neurons = 16;

%normalize the data set to have zero mean
[X,mu,sigma] = featureNormalize(X);
[y,mu2,sigma2] = featureNormalize(y);

%Load test data
load('datanew.mat','data');
thetav = data(:,1);
pv = data(:,2);
uv = data(:,3);
p_dotv = data(:,4);

XV(1,:) = thetav;
XV(2,:) = pv;
XV(3,:) = uv;
yv = p_dotv;

%normalize the data set to have zero mean
[XV,muv,sigmav] = featureNormalize(XV);
[yv,muv2,sigmav2] = featureNormalize(yv);

epochs = 100;
a1 = zeros(neurons,1);
v1 = zeros(neurons,1);

Theta1(1,:) = 0.01*rand(1,neurons);
```

```
Theta1(2,:) = 0.02*rand(1,neurons);
Theta1(3,:) = 0.01*rand(1,neurons);
Theta2 = 0.1*rand(1,neurons);

eta = 0.1;%Learning rate

erms = zeros(epochs,1);
ermsTest = zeros(epochs,1);

for i = 1:epochs
    e = 0;
    for j = 1:size(X,2)
        a2 = 0;
        % Calculation of output (Training data)
        for k=1:neurons
            a1(k) = X(1,j)*Theta1(1,k) + X(2,j)*Theta1(2,k) +
                X(3,j)*Theta1(3,k);
            v1(k) = sigmoid(a1(k));
            a2 = a2 + v1(k)*Theta2(1,k);
        end
        out = a2;
        e = e + computeCost(out,y(j));

        for k = 1:neurons
          [Theta2, Theta1] = updateWeights(X, y, out, Theta1, Theta2, k, j,
              eta, v1);
        end
    end
    erms(i) = sqrt(e/length(p_dot));

    e_test = 0;

    for j = 1:length(p_dotv)
        a2 = 0;
        for k = 1:neurons
            a1(k) = X(1,j)*Theta1(1,k) + X(2,j)*Theta1(2,k) +
                X(3,j)*Theta1(3,k);
            v1(k) = sigmoid(a1(k));
            a2 = a2 + v1(k)*Theta2(1,k);
        end
            out_test(j) = a2;
            e_test = e_test + computeCost(out_test(j),yv(j));
    end
    ermsTest(i) = sqrt(e_test/length(p_dotv));
end
```

3.7 Results

The following plots show the variation in performance as the number of neurons in the hidden layer is changed.

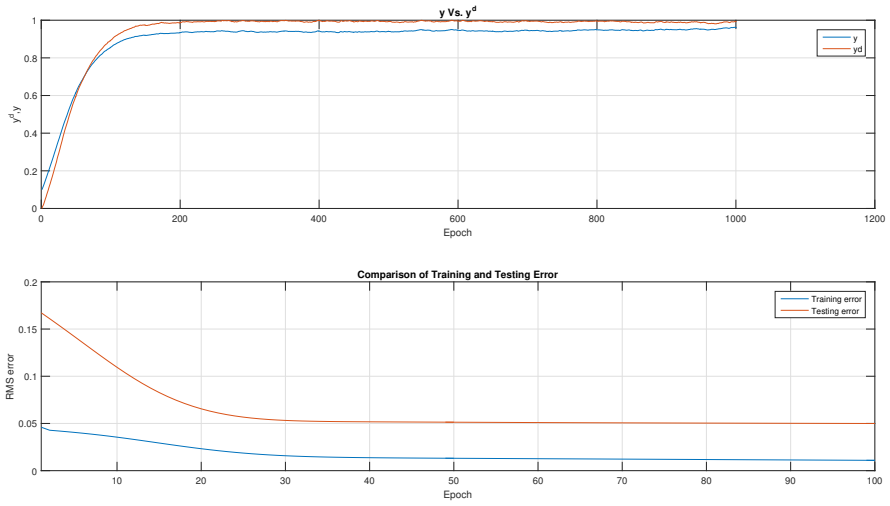

Figure 3.2: Neural Network Performance for 4 neurons

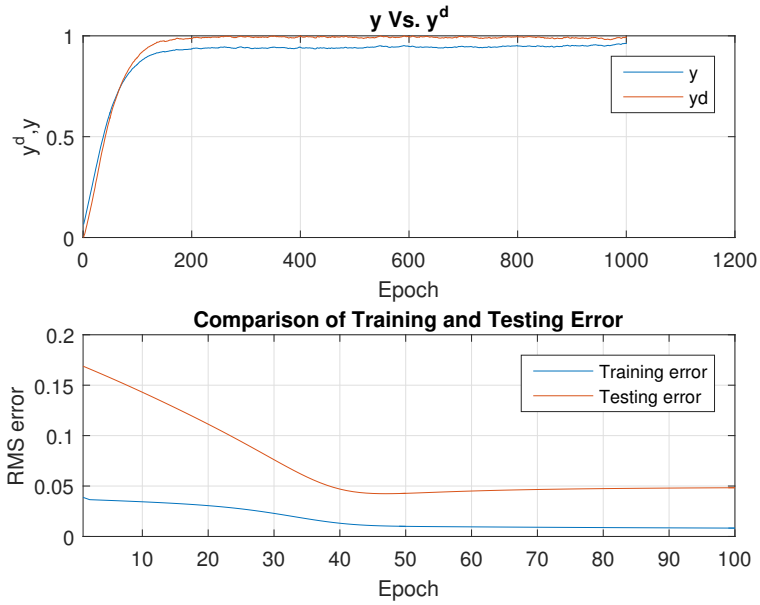

Figure 3.3: Neural Network Performance for 6 neurons

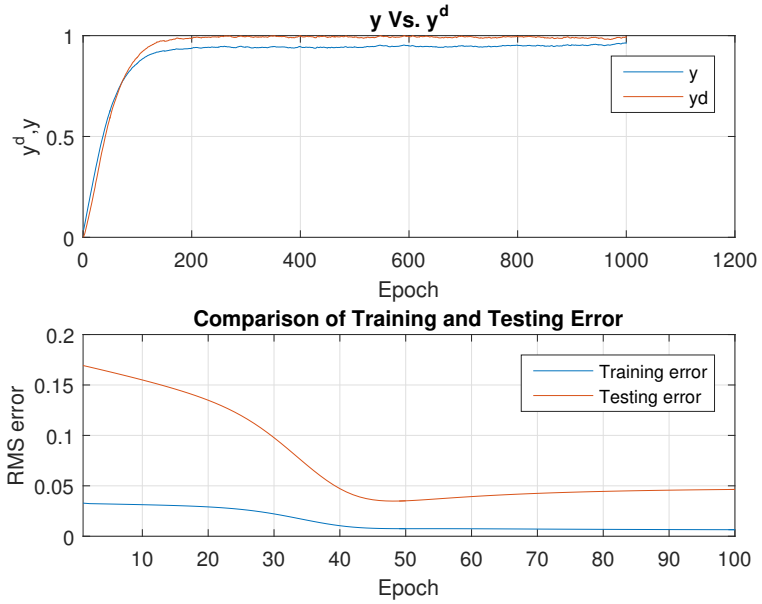

Figure 3.4: Neural Network Performance for 8 neurons

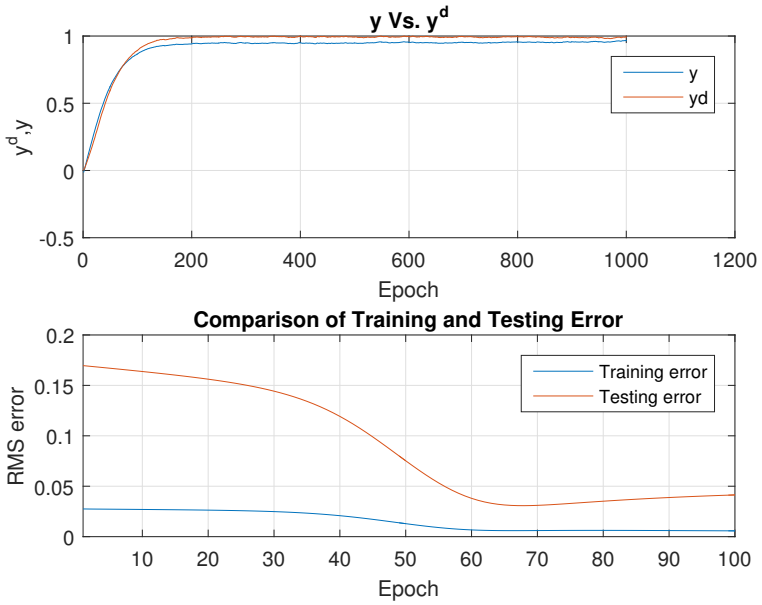

Figure 3.5: Neural Network Performance for 12 neurons

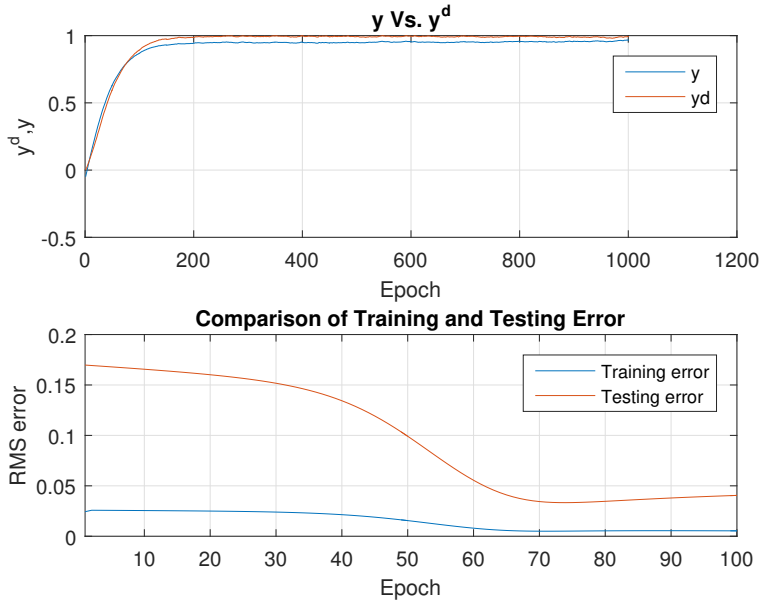

Figure 3.6: Neural Network Performance for 14 neurons

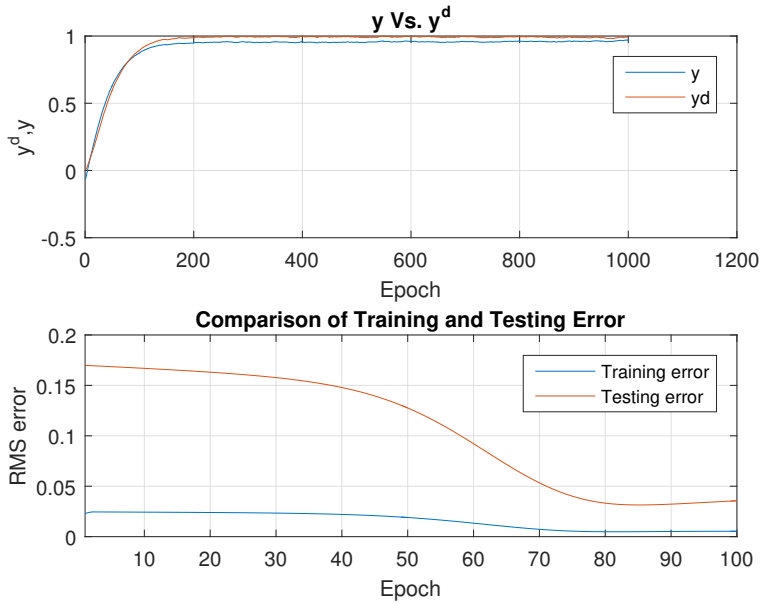

Figure 3.7: Neural Network Performance for 16 neurons

4 Bonus Question

Citation: https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/
The following code is the first iteration of the implementation of training on the mnist dataset. It uses only a single layer of ten softmax neurons. The model is $y = softwmax(xW + b)$ where x is the matrix for 100 grayscale images of 28x28 pixels, flattened (there are 100 images in a mini-batch), W is the weight matrix with 784 rows and 10 columns and b is the basis vector with 10 dimensions.

```
import tensorflow as tf
import tensorflowvisu
from tensorflow.contrib.learn.python.learn.datasets.mnist import read_data_sets
tf.set_random_seed(0)

# Download images and labels into mnist.test (10K images+labels) and
    mnist.train (60K images+labels)
mnist = read_data_sets("data", one_hot=True, reshape=False, validation_size=0)

# input X: 28x28 grayscale images, the first dimension (None) will index the
    images in the mini-batch
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
# correct answers will go here
Y_ = tf.placeholder(tf.float32, [None, 10])
# weights W[784, 10] 784=28*28
W = tf.Variable(tf.zeros([784, 10]))
# biases b[10]
b = tf.Variable(tf.zeros([10]))

# flatten the images into a single line of pixels
XX = tf.reshape(X, [-1, 784])

# The model
Y = tf.nn.softmax(tf.matmul(XX, W) + b)

# loss function: cross-entropy = - sum( Y_i * log(Yi) )
#                           Y: the computed output vector
#                           Y_: the desired output vector

# cross-entropy
cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0 # normalized for
    batches of 100 images

# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# training, learning rate = 0.005
train_step = tf.train.GradientDescentOptimizer(0.005).minimize(cross_entropy)
```

```
# matplotlib visualisation
allweights = tf.reshape(W, [-1])
allbiases = tf.reshape(b, [-1])
I = tensorflowvisu.tf_format_mnist_images(X, Y, Y_) # assembles 10x10 images
    by default
It = tensorflowvisu.tf_format_mnist_images(X, Y, Y_, 1000, lines=25) # 1000
    images on 25 lines
datavis = tensorflowvisu.MnistDataVis()

# init
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# Call this function in a loop to train the model, 100 images at a time
def training_step(i, update_test_data, update_train_data):

    # training on batches of 100 images with 100 labels
    batch_X, batch_Y = mnist.train.next_batch(100)

    # compute training values for visualisation
    if update_train_data:
        a, c, im, w, b = sess.run([accuracy, cross_entropy, I, allweights,
            allbiases], feed_dict={X: batch_X, Y_: batch_Y})
        datavis.append_training_curves_data(i, a, c)
        datavis.append_data_histograms(i, w, b)
        datavis.update_image1(im)
        print(str(i) + ": accuracy:" + str(a) + " loss: " + str(c))

    # compute test values for visualisation
    if update_test_data:
        a, c, im = sess.run([accuracy, cross_entropy, It], feed_dict={X:
            mnist.test.images, Y_: mnist.test.labels})
        datavis.append_test_curves_data(i, a, c)
        datavis.update_image2(im)
        print(str(i) + ": ********* epoch " +
            str(i*100//mnist.train.images.shape[0]+1) + " ********* test
            accuracy:" + str(a) + " test loss: " + str(c))

    # the backpropagation training step
    sess.run(train_step, feed_dict={X: batch_X, Y_: batch_Y})


datavis.animate(training_step, iterations=2000+1, train_data_update_freq=10,
    test_data_update_freq=50, more_tests_at_start=True)

print("max test accuracy: " + str(datavis.get_max_test_accuracy()))

# final max test accuracy = 0.9268 (10K iterations). Accuracy should peak
```

```
    above 0.92 in the first 2000 iterations.
```

4.1 Second Iteration

This model is very similar to the last model, except for the fact that it uses 5 layers as opposed to 1. We get decent accuracy on the test data. One possible way to further increase the accuracy of this model would be to spatially encode the model by using CNNs.

```
import tensorflow as tf

import tensorflowvisu

from tensorflow.contrib.learn.python.learn.datasets.mnist import read_data_sets
tf.set_random_seed(0)

# Download images and labels into mnist.test (10K images+labels) and
    mnist.train (60K images+labels)
mnist = read_data_sets("data", one_hot=True, reshape=False, validation_size=0)

# input X: 28x28 grayscale images, the first dimension (None) will index the
    images in the mini-batch
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
# correct answers will go here
Y_ = tf.placeholder(tf.float32, [None, 10])

# five layers and their number of neurons (tha last layer has 10 softmax
    neurons)
L = 200
M = 100
N = 60
O = 30
# Weights initialised with small random values between -0.2 and +0.2

# With RELUs, initializing biases with small *positive* values

W1 = tf.Variable(tf.truncated_normal([784, L], stddev=0.1)) # 784 = 28 * 28
B1 = tf.Variable(tf.zeros([L]))
W2 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B2 = tf.Variable(tf.zeros([M]))
W3 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
B3 = tf.Variable(tf.zeros([N]))
W4 = tf.Variable(tf.truncated_normal([N, O], stddev=0.1))
B4 = tf.Variable(tf.zeros([O]))
W5 = tf.Variable(tf.truncated_normal([O, 10], stddev=0.1))
B5 = tf.Variable(tf.zeros([10]))

# The model
```

```
XX = tf.reshape(X, [-1, 784])
Y1 = tf.nn.sigmoid(tf.matmul(XX, W1) + B1)
Y2 = tf.nn.sigmoid(tf.matmul(Y1, W2) + B2)
Y3 = tf.nn.sigmoid(tf.matmul(Y2, W3) + B3)
Y4 = tf.nn.sigmoid(tf.matmul(Y3, W4) + B4)
Ylogits = tf.matmul(Y4, W5) + B5
Y = tf.nn.softmax(Ylogits)

# cross-entropy loss function (= -sum(Y_i * log(Yi)) ), normalised for batches
    of 100 images
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits,
    labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy)*100

# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# matplotlib visualisation
allweights = tf.concat([tf.reshape(W1, [-1]), tf.reshape(W2, [-1]),
    tf.reshape(W3, [-1]), tf.reshape(W4, [-1]), tf.reshape(W5, [-1])], 0)
allbiases = tf.concat([tf.reshape(B1, [-1]), tf.reshape(B2, [-1]),
    tf.reshape(B3, [-1]), tf.reshape(B4, [-1]), tf.reshape(B5, [-1])], 0)
I = tensorflowvisu.tf_format_mnist_images(X, Y, Y_)
It = tensorflowvisu.tf_format_mnist_images(X, Y, Y_, 1000, lines=25)
datavis = tensorflowvisu.MnistDataVis()

# training step, learning rate = 0.003
learning_rate = 0.003
train_step = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)

# init
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)


# Call this function in a loop to train the model, 100 images at a time
def training_step(i, update_test_data, update_train_data):

    # training on batches of 100 images with 100 labels
    batch_X, batch_Y = mnist.train.next_batch(100)

    # compute training values for visualisation
    if update_train_data:
        a, c, im, w, b = sess.run([accuracy, cross_entropy, I, allweights,
            allbiases], {X: batch_X, Y_: batch_Y})
        print(str(i) + ": accuracy:" + str(a) + " loss: " + str(c) + " (lr:" +
            str(learning_rate) + ")")
```

```
        datavis.append_training_curves_data(i, a, c)
        datavis.update_image1(im)
        datavis.append_data_histograms(i, w, b)

    # compute test values for visualisation
    if update_test_data:
        a, c, im = sess.run([accuracy, cross_entropy, It], {X:
            mnist.test.images, Y_: mnist.test.labels})
        print(str(i) + ": ********* epoch " +
            str(i*100//mnist.train.images.shape[0]+1) + " ********* test
            accuracy:" + str(a) + " test loss: " + str(c))
        datavis.append_test_curves_data(i, a, c)
        datavis.update_image2(im)

    # the backpropagation training step
    sess.run(train_step, {X: batch_X, Y_: batch_Y})

datavis.animate(training_step, iterations=10000+1, train_data_update_freq=20,
    test_data_update_freq=100, more_tests_at_start=True)

print("max test accuracy: " + str(datavis.get_max_test_accuracy()))

# Some results that were achieved:

## learning rate = 0.003, 10K iterations
# final test accuracy = 0.9788 (sigmoid - slow start, training cross-entropy
    not stabilised in the end)
# final test accuracy = 0.9825 (relu - above 0.97 in the first 1500 iterations
    but noisy curves)

## now with learning rate = 0.0001, 10K iterations
# final test accuracy = 0.9722 (relu - slow but smooth curve, would have gone
    higher in 20K iterations)
)
```