CAP5615 Justin Johnson
Summer 2018 Z23136514

Question 1

Use own language to explain the following concepts:

Gradient Descent:
Popular machine learning optimization method (learning rule) which is used for updating a model's weights during the training process. Using Gradient Descent, weight updates are calculated using the negative of the squared error function's gradient. This error is calculated by passing training examples through network of linear units (no activation functions, therefore continuous) and calculating error as difference between expected outcome and predicted outcome. The negative derivative of the model's squared error will point downhill, and Gradient Descent will use the learning rate parameter to determine how far to step in the downhill direction. Gradient descent can update weights after calculating the training error for all training instances, or alternatives can update weights after a random subset of instances.

Neural Network learning rate:
A machine learning hyperparameter that determines the size of the step to take when updating weights. In neural networks, the learning rate is multiplied by the negative gradient of the network's squared error, and the result is used to update the weights of the network. If the learning rate is too small, learning will take longer, but if it is too high then it is possible that the network will oscillate or even diverge.

Multi-Layer Feed Forward Neural Network:
At least one hidden layer is required for a multi-layer network, a layer which sits between the input and output layers. With the exception of the last layer, each layer l_k must be connected to the next layer in the network l_{k+1} to satisfy requirement of feed forward network. In addition, feed forward networks can not contain loops or connections to previous layers. Therefore, multi-layer feed forward network is a neural network with at least one hidden layer and where every layer is connected to the following layer, with no loops to same layer or previous layers. Multi-layered networks solve a major problem encountered with single layer networks – they are able learn non-linearly separable data.

Hidden Nodes in Neural Network:
Hidden nodes are nodes of the network's hidden layer. Hidden nodes receive their input from an input layer or another hidden layer, and their output is fed into either another hidden layer or an output layer.

Output Nodes in Neural Network:
The final layer in the neural network. The number of output nodes will determine the size of the network's output vector, and is usually determined to be the total number of possible classes expected.

Backpropagation Rule:
The rule which is used to update weights in a multilayer network, it determines how to distribute weight adjustments throughout the network based on the error of the network and gradient descent. There are two steps – forward pass, and backward pass. The forward pass computes the output of all units in the network and calculates the network's error. Then the backward pass, starting with output layer and moving back through the network, recursively computes the local gradient of each neuron and uses this to update the network's weights.

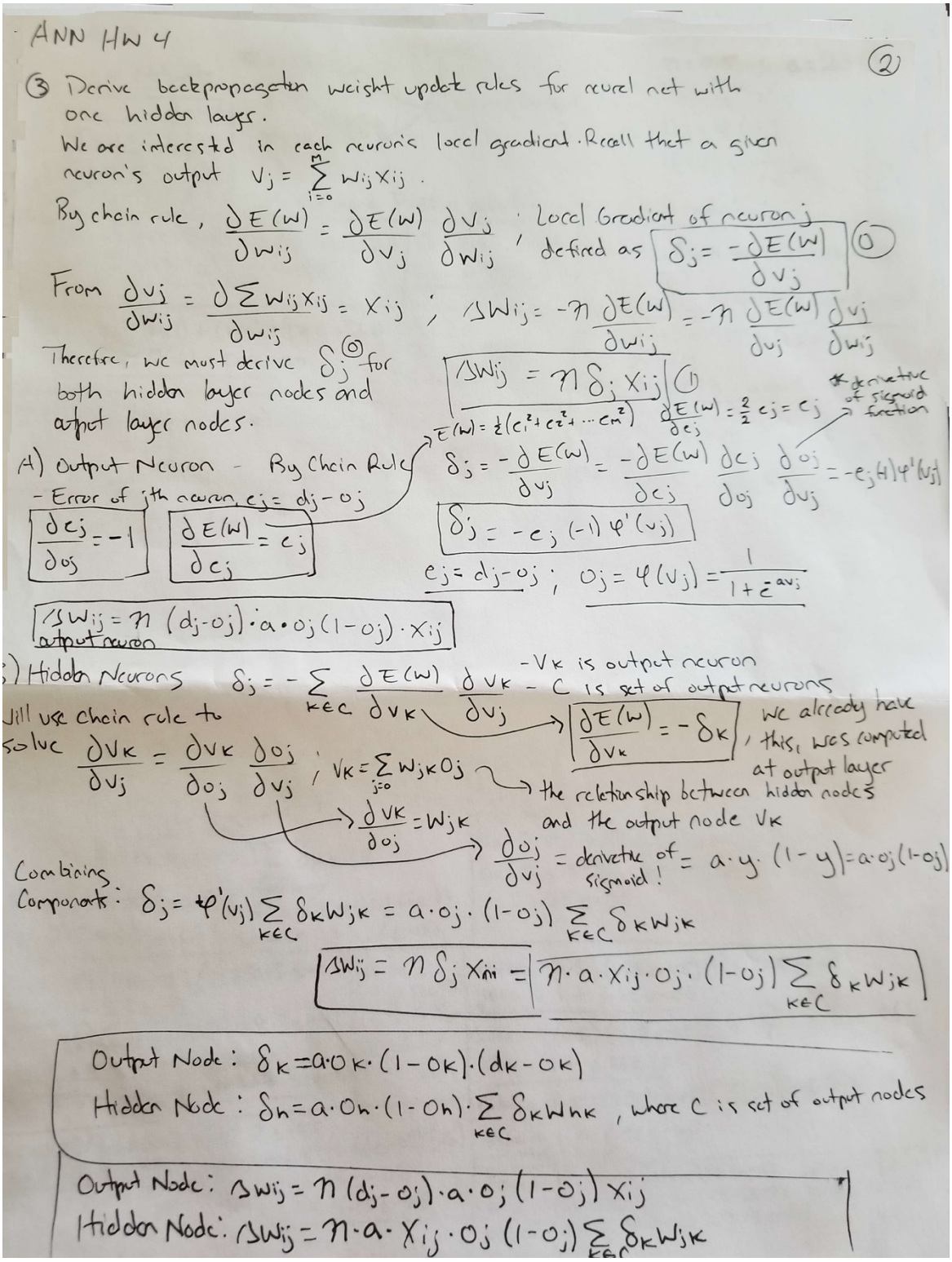
Question 2

What is the gradient at point (2, 4) for equation $y = x^2$
The derivative of $y = x^2$ with respect to x is 2x. At (2, 4), the gradient = (2)(2) = 4.

Following gradient descent, find the next movement towards the global minimum:
With a learning rate of 0.1, the next step will be the negative of the gradient multiplied by the learning rate = - 0.4

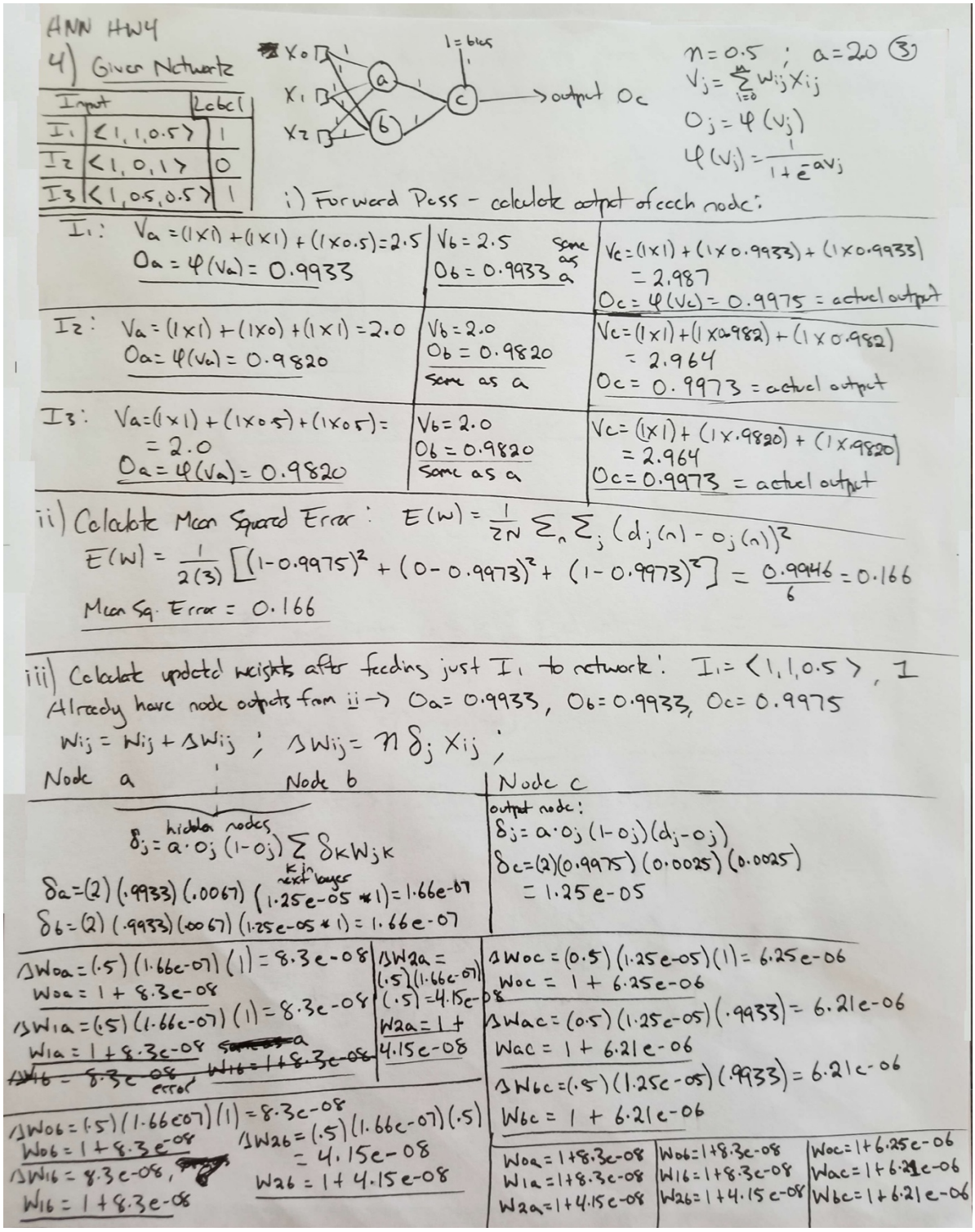
Question 3

Derive back propagation update rule for neural network with one hidden layer and one output layer, including BP rule for hidden and output nodes. Derivation included in below image:



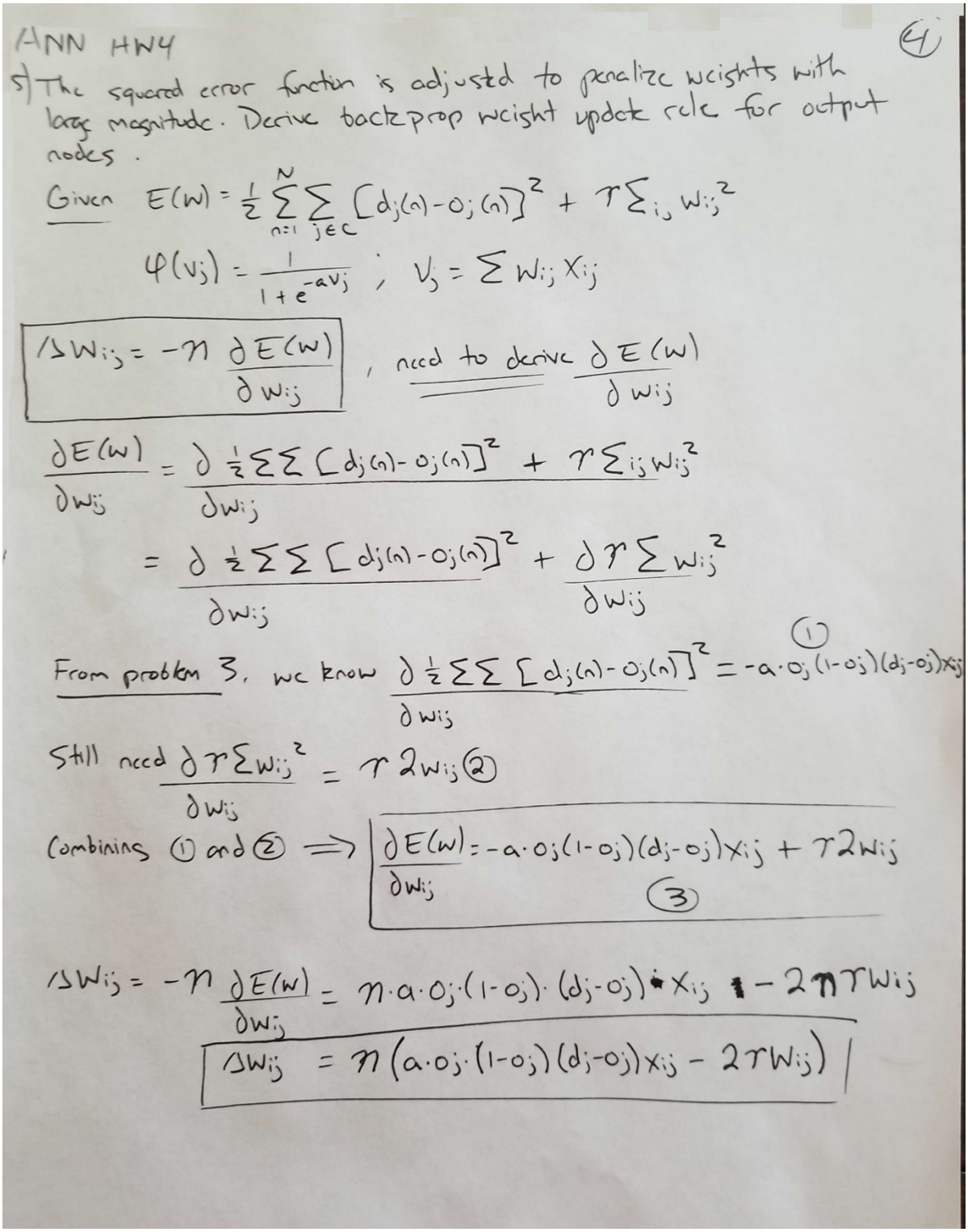
Question 4

What is mean squared error of the network with respect to the 3 instances? Update the weights using instance I_1. The following image includes work and solution:



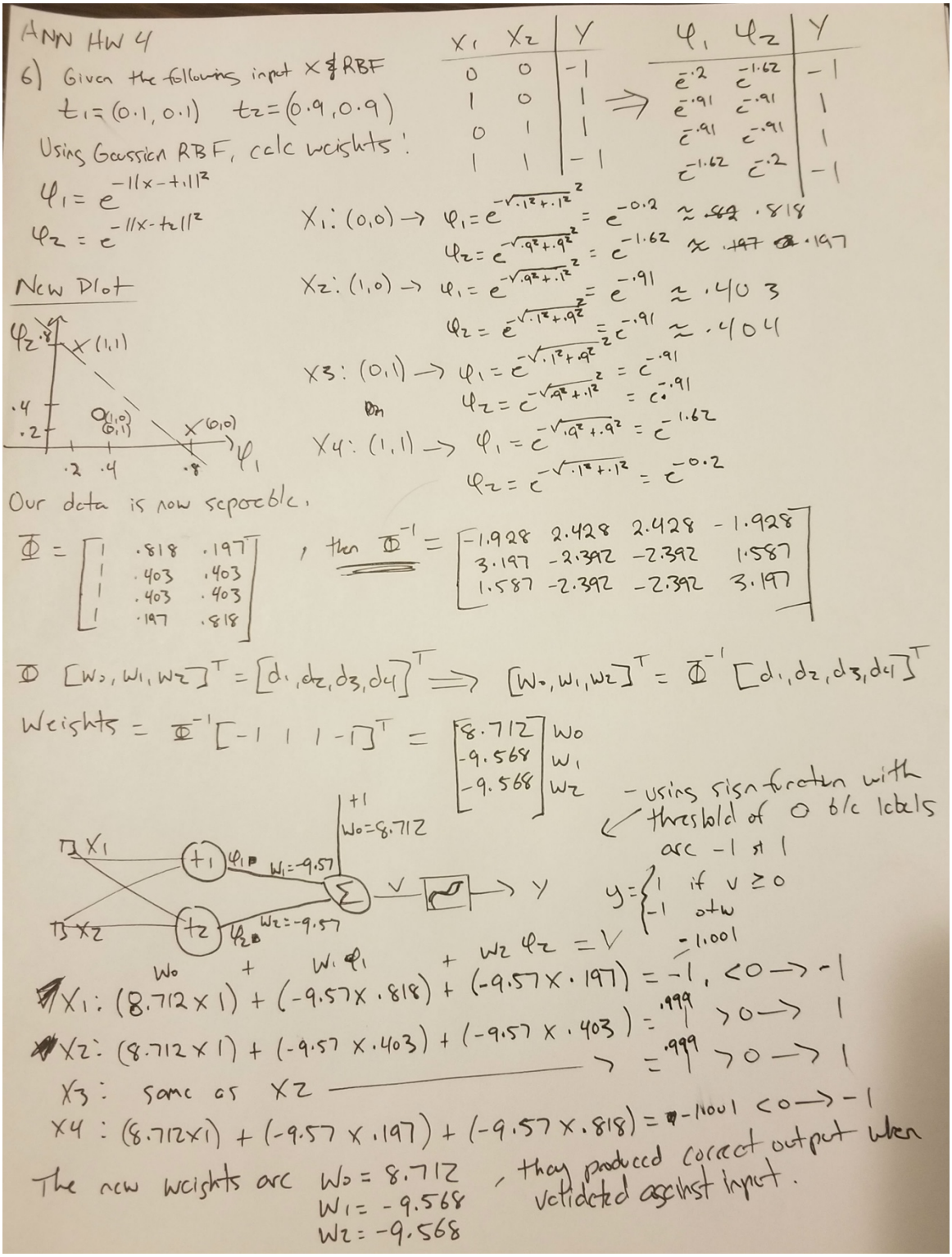
Question 5

Derive back propagation update rule given the new squared error function which penalizes weights with high magnitudes:



Question 6

Assume RBF network using Gaussian RBF function. Use pseudo-inverse to calculate weight values of the output node, validate results with respect to the 4 instances provided:



Question 7

Design and implement the following face recognition task using Neural Networks:

DL 100 face images from CMU to build two class classification tasks. Must specify faces in positive of negative classes (left vs right). Include all faces in final submission.

Face images were downloaded from CMU (http://www.cs.cmu.edu/~tom/faces.html) in low resolution format. For this model, we are only interested in two classes of face images: heads turned left (positive) and heads turned right (negative). The original data set images are organized by person, and contain images with heads facing left, right, up, and down. Left and right facing images were combined to create two new directories, each appropriately containing only positive (left) or negative (right) images.

```
SB-FL-46ZTHTD6:Desktop justin.johnson$ cp faces_4/
.DS_Store  an2i/      boland/    ch4f/      choon/     glickman/  kawamura/  megak/     night/     saavik/    sz24/
Positive/  at33/      bpm/       cheyer/    danieln/   karyadi/   kk49/      mitchell/  phoebe/    steffi/    tammo/
SB-FL-46ZTHTD6:Desktop justin.johnson$ cp faces_4/*/*left*.pgm FaceData/Positive/
SB-FL-46ZTHTD6:Desktop justin.johnson$ cp faces_4/*/*right*.pgm FaceData/Negative/
SB-FL-46ZTHTD6:Desktop justin.johnson$ ls FaceData/Positive/
an2i_left_angry_open_4.pgm              cheyer_left_angry_open_4.pgm            kawamura_left_angry_sunglasses_4.pgm    phoebe_left_happy_open_4.pgm
an2i_left_angry_sunglasses_4.pgm        cheyer_left_angry_sunglasses_4.pgm      kawamura_left_happy_open_4.pgm          phoebe_left_happy_sunglasses_4.pgm
an2i_left_happy_open_4.pgm              cheyer_left_happy_open_4.pgm            kawamura_left_happy_sunglasses_4.pgm    phoebe_left_neutral_open_4.pgm
an2i_left_happy_sunglasses_4.pgm        cheyer_left_happy_sunglasses_4.pgm      kawamura_left_neutral_open_4.pgm        phoebe_left_neutral_sunglasses_4.pgm
an2i_left_neutral_open_4.pgm            cheyer_left_neutral_open_4.pgm          kawamura_left_neutral_sunglasses_4.pgm  phoebe_left_sad_open_4.pgm
an2i_left_neutral_sunglasses_4.pgm      cheyer_left_neutral_sunglasses_4.pgm    kawamura_left_sad_open_4.pgm            phoebe_left_sad_sunglasses_4.pgm
an2i_left_sad_open_4.pgm                cheyer_left_sad_open_4.pgm              kawamura_left_sad_sunglasses_4.pgm      saavik_left_angry_open_4.pgm
an2i_left_sad_sunglasses_4.pgm          cheyer_left_sad_sunglasses_4.pgm        kk49_left_angry_open_4.pgm              saavik_left_angry_sunglasses_4.pgm
```

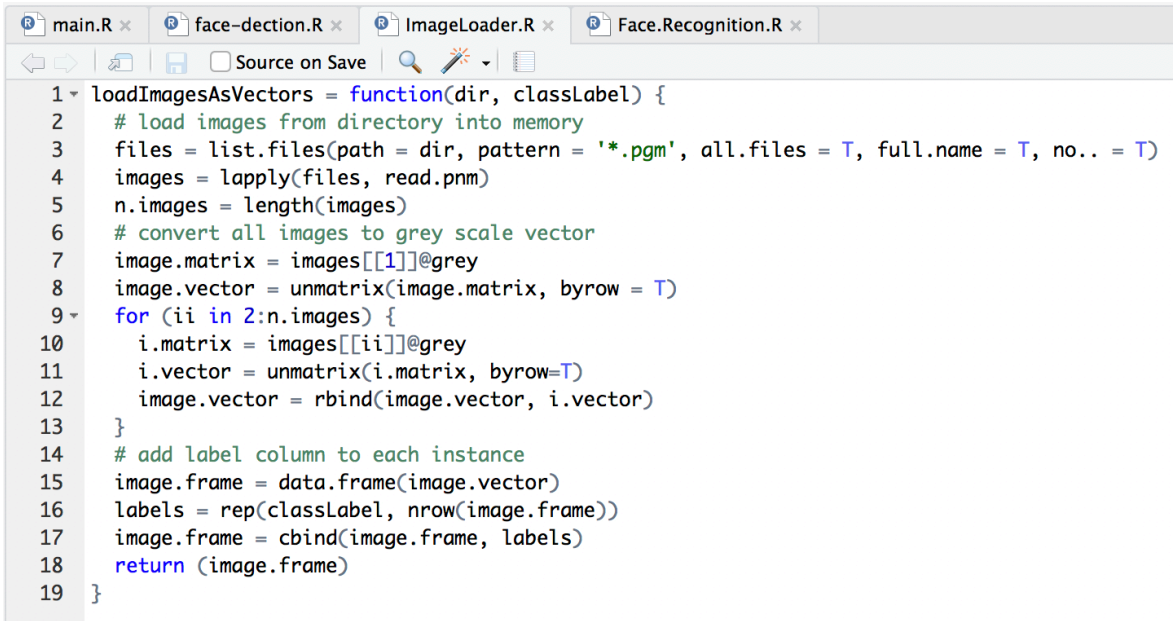
The above image displays the splitting of data into positive/negative classes. A preview of the positive file names are also displayed. There are 157 positive images and 155 negative images.

Two images, one positive and one negative, were loaded into R and plotted:

Left image is positive (head turned left) and right image is negative (head turned right). Images are low resolution (32 x 30) to reduce computation cost.
Below image contains the code required to read and plot images:

```r
# Homework 4
# Problem 7

# load dependencies
library(pixmap)
library(gdata)
library(neuralnet)
source('ImageLoader.R')


# define data path
positiveDir = '../FaceData/Positive'
negativeDir = '../FaceData/Negative'


# plot 1 positive and 1 negative sample, to preview
posFile = list.files(positiveDir)[1]
posFile = read.pnm(paste(positiveDir, posFile, sep='/'))
plot(posFile)
negFile = list.files(negativeDir)[1]
negFile = read.pnm(paste(negativeDir, negFile, sep='/'))
plot(negFile)

```

A helper function was created to load images from file into memory, convert each file to a matrix of grey scale pixel intensities, flatten matrix to a vector, and add a column for class label. The help function then takes all image instances (as vectors) and row binds them, producing a matrix where each row is an image and each column is a pixel intensity value. This helper function is based off the one presented in lecture.

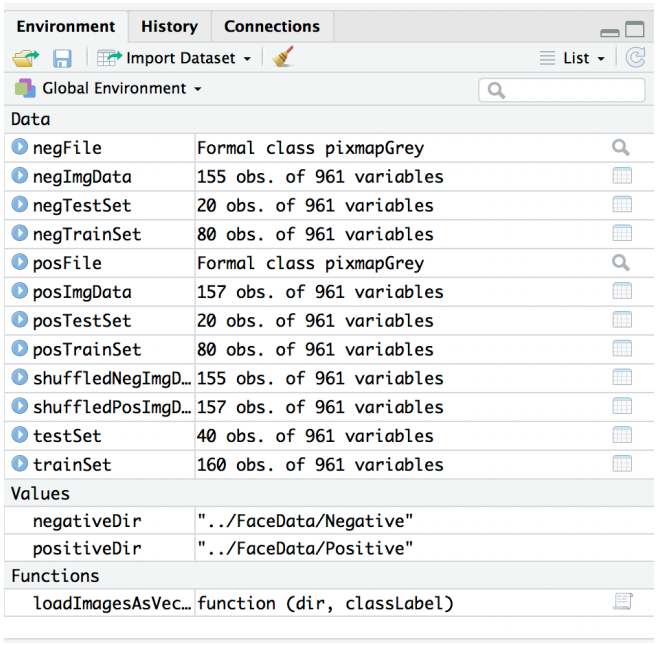



```r
loadImagesAsVectors = function(dir, classLabel) {
  # load images from directory into memory
  files = list.files(path = dir, pattern = '*.pgm', all.files = T, full.name = T, no.. = T)
  images = lapply(files, read.pnm)
  n.images = length(images)
  # convert all images to grey scale vector
  image.matrix = images[[1]]@grey
  image.vector = unmatrix(image.matrix, byrow = T)
  for (ii in 2:n.images) {
    i.matrix = images[[ii]]@grey
    i.vector = unmatrix(i.matrix, byrow=T)
    image.vector = rbind(image.vector, i.vector)
  }
  # add label column to each instance
  image.frame = data.frame(image.vector)
  labels = rep(classLabel, nrow(image.frame))
  image.frame = cbind(image.frame, labels)
  return (image.frame)
}
```

The below screenshot displays the code required to load all images as vectors and randomly partition into train and test data sets. Using a balanced data set of 200 images, 160 will be used for training and 40 will be used for evaluation.

```
25  # load image data as vectors
26  # each row is instance
27  # each column is grey intensity for given pixel
28  posImgData = loadImagesAsVectors('../FaceData/Positive', 1)
29  negImgData = loadImagesAsVectors('../FaceData/Negative', 0)
30
31
32  # we will use 100 positive and 100 negative instances for train/test sets
33  # will split the data 80/20 for training and testing
34  shuffledPosImgData = posImgData[sample(1:nrow(posImgData)), ]
35  shuffledNegImgData = negImgData[sample(1:nrow(negImgData)), ]
36  posTrainSet = shuffledPosImgData[1: 80, ]
37  negTrainSet = shuffledNegImgData[1: 80, ]
38  posTestSet = shuffledPosImgData[81: 100, ]
39  negTestSet = shuffledNegImgData[81: 100, ]
40
41  trainSet = rbind(posTrainSet, negTrainSet)
42  testSet = rbind(posTestSet, negTestSet)
43
```

I've included a screenshot of the R Studio workspace, allowing easy confirmation of data set that has been loaded into memory:



Our training set contains 160 instances, each with 961 variables. The first 960 variables are the pixel intensities, and the last is the instance label. Similarly, our test set contains 40 instances, each with 961 variables.

Next the R neuralnet package is used to train 5 different multi-layer feed forward neural network architectures using the training set. All neural networks will include 960 input nodes, that is one node for each pixel of the image. They will also include 1 output node, allowing for binary classification of images as positive (head turned left) or negative (head turned right). Just one hidden layer will be used, and the total number of hidden nodes will be varied between 3 and 11.

```
# specify formula (a symbolic description of the model to be fitted)
n = names(trainSet)
myform = as.formula(paste('labels ~ ', paste(n[!n %in% 'labels'], collapse = ' + ')))

# mark the label index
labelIdx = length(trainSet)

# we don't need to add bias, the neuralnet library will do for us

# 1st network - 3 hidden nodes
face.3.classifier <- neuralnet(myform, trainSet, hidden=3, rep=500, linear.output=FALSE, threshold=0.1)
face.3.prediction<-compute(face.3.classifier, testSet[,-labelIdx])
face.3.results = face.3.prediction$net.result
face.3.results = lapply(face.3.results, round)
face.3.results = unlist(face.3.results)

# 2nd network - 5 hidden nodes
face.5.classifier <- neuralnet(myform, trainSet, hidden=5, rep=500, linear.output=FALSE, threshold=0.1)
face.5.prediction<-compute(face.5.classifier, testSet[,-labelIdx])
face.5.results = face.5.prediction$net.result
face.5.results = lapply(face.5.results, round)
face.5.results = unlist(face.5.results)

# 3rd network - 7 hidden nodes
face.7.classifier <- neuralnet(myform, trainSet, hidden=7, rep=500, linear.output=FALSE, threshold=0.1)
face.7.prediction<-compute(face.7.classifier, testSet[,-labelIdx])
face.7.results = face.7.prediction$net.result
face.7.results = lapply(face.7.results, round)
face.7.results = unlist(face.7.results)


# 4th network - 9 hidden nodes
face.9.classifier <- neuralnet(myform, trainSet, hidden=9, rep=500, linear.output=FALSE, threshold=0.1)
face.9.prediction<-compute(face.9.classifier, testSet[,-labelIdx])
face.9.results = face.9.prediction$net.result
face.9.results = lapply(face.9.results, round)
face.9.results = unlist(face.9.results)


# 5th network - 11 hidden nodes
face.11.classifier <- neuralnet(myform, trainSet, hidden=11, rep=500, linear.output=FALSE, threshold=0.1)
face.11.prediction<-compute(face.11.classifier, testSet[,-labelIdx])
face.11.results = face.11.prediction$net.result
face.11.results = lapply(face.11.results, round)
face.11.results = unlist(face.11.results)
```

Once training is complete, the 5 networks are evaluated using the test set. The output of the network is a value in the range (0, 1), as produced by the output node's activation function. For each network, this result is converted to a 0 or 1 by applying the round function to each result. This is the equivalent to a function which returns 1 if the probability of being equal to 1 is greater than 0.5. These results can now be compared to the test set's labels to generate a confusion matrix.

```
88
89  # generate confusion amtrixes for each network
90  face.3.confMatrix = table(testSet$labels, face.3.results)
91  face.5.confMatrix = table(testSet$labels, face.5.results)
92  face.7.confMatrix = table(testSet$labels, face.7.results)
93  face.9.confMatrix = table(testSet$labels, face.9.results)
94  face.11.confMatrix = table(testSet$labels, face.11.results)
95
```

The following table compares the results from each network:

Network #	Hidden Nodes	TP	FP	TN	FN	Accuracy	Error
1	3	20	0	20	0	100%	0%
2	5	20	0	20	0	100%	0%
3	7	20	0	20	0	100%	0%
4	9	20	0	20	0	100%	0%
5	11	20	2	18	0	95%	5%

When trained with 160 images and tested with a new set of 40 images, the model's performed shockingly well. Additional iterations were run, and since the images are sampled at random, the results varied with each iteration as expected. In general, all models performed very well with 100% accuracy. In several cases, networks with 9 or 11 hidden nodes yielded higher errors on the test set. It is likely that if test and training error were visualized, overfitting would be observed on the networks with more hidden nodes, therefore causing errors on the test set.