Intelligent Systems HW3 Part 2

# Problem Description

Given a data set of 24x24 pixel hand written digits

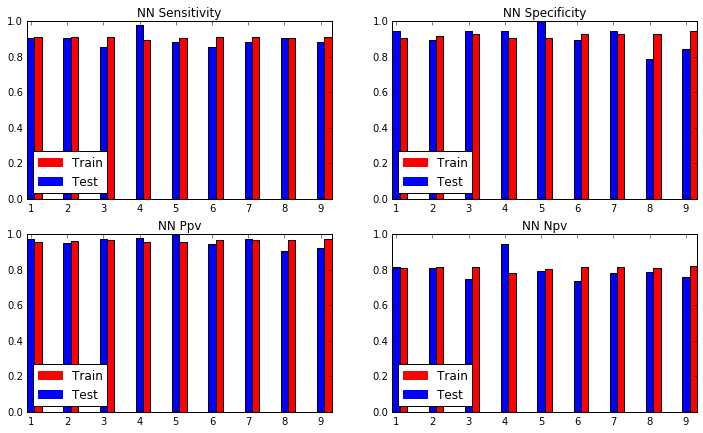
patient sodium levels and blood pressure, labeled with a binary disease presence value, we are asked to develop a model for predicting the presence of the disease given a new patient’s sodium and blood pressure levels. The model used to perform this task was a neural network that used a input layer of size 2, a hidden layer of size 20, and an output layer of size 2. The learning rate for each layer was .05, momentum was 0, and the number of epochs was 200. The neural network model was written as a modular library, with all operations vectorized to take advantage of the numpy BLAS fortran code linkage for speed up during prediction and backpropagation calculations.

# Approach

The neural network code was developed so that an arbitrary number of layers can be used. New layers are added to the model using the add() function, and can specify size, learning rate, and momentum. The ability to use a different loss function and optimizer for the network was added, but is currently set to stochastic gradient descent and a least mean square loss function. Each layer is represented by a “Layer” class, and contains a numpy matrix representing the input weights to each neuron, as well as parameters defining the layer such as learning rate and momentum. Each layer is independent of the other layers, and the model class controls the propagation of data through the layers to classify, as well as to backpropogate errors. Through quick trial and error, it was determined that a hidden layer of 20 neurons, a learning rate of 0.05, no momentum, and 200 epochs was enough to get accuracies of 90% and above.

The attribute values were also scaled so that both attributes would fall into the range of 0 to 1. The formula used was, . This kind of scaling is called range scaling, and this kind of scaling ensures that similarities found in data are due to relative distances, and not due to the inherent magnitude of attributes. An alternative method that could have been used is Z-score scaling, since the data is approximately normal for each attribute.

# Performance on Individual Trials

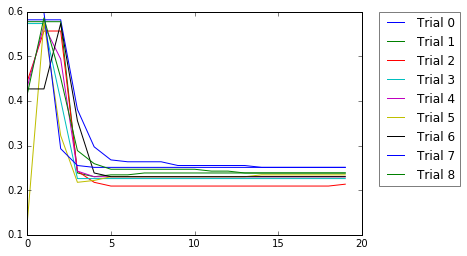
**NN Metrics **

# Average Performance

**NN Metrics**

|  |  |  |
| --- | --- | --- |
|  | **Test** | **Train** |
| **Sensitivity** |  |  |
| **Specificity** |  |  |
| **Ppv** |  |  |
| **Npv** |  |  |

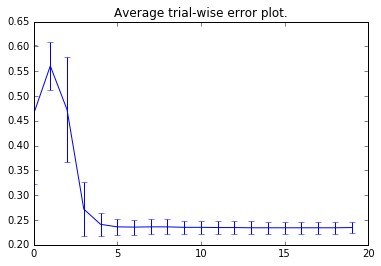
# Trial-Wise Training Error for the NN



Error

Epoch % 10

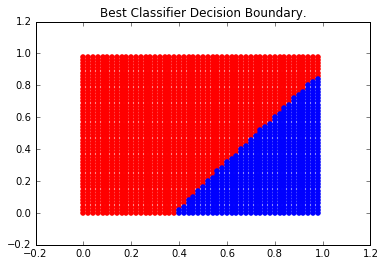
# Mean Training Error for the NN



Error

Epoch % 10

# Best NN Decision Boundary



# Analysis of Results

**Part A:** Your analysis of what the results over the 9 trials

for the classier indicate about the suitability of the classifier for the problem;

Looking at the graphs of metrics for the neural net, it is evident that the model has consistently high metrics. Indeed, looking at the table averages, the neural net model scores in the range of 90% for almost all metrics. While the boundaries of the neural net classifier look to be almost the same as the linear perceptron used to classify this data in homework 2, the decision boundary is consistently found, regardless of the data subset used. This means that the neural net is more robust to selection of training data subsets than the perceptron model.

**Part B:** Your opinions about the pros and cons of this classifier.

The neural network model has the ability to be very expressive, and can construct simple or highly non-linear boundaries. While the model can be accurate for many data distributions, it is prone to overfitting due to its large number of parameters and may draw overly complicated decision boundaries. Furthermore, with more tunable parameters, it can be more difficult to determine the precise value of each parameter that will give the best performance.

# Appendix: Programs

# example.py code

*# -\*- coding: utf-8 -\*-  
"""  
Created on Wed Mar 30 18:39:35 2016  
  
This program shows how to use the NN\_library. it  
  
  
@author: DAN  
"""***import** numpy **as** np  
**import** pandas **as** pd  
**from** NN\_library **import** NNModel  
**import** random  
**from** sklearn.cross\_validation **import** train\_test\_split  
**import** matplotlib.pyplot **as** plt  
**import** matplotlib.patches **as** mpatches  
**from** pylab **import** \*  
  
  
**def** buildSmallExampleNet():  
 *# Build model.* mModel = NNModel.Model()  
 mModel.add(layer\_size=2, learning\_rate=1, isInput=**True**)  
 mModel.add(layer\_size=3, learning\_rate=1, momentum\_factor=.3)  
 mModel.add(layer\_size=2, learning\_rate=1, momentum\_factor=.3)  
 print(**"Created Model."**)  
  
 *# Train model.* testData = np.array([[1,1]])  
 labelData = np.array([[1,0]])  
 mModel.train(testData, labelData, epochs=10000)  
 *# Predict data.* output = mModel.predict(testData[0])  
 print(**"Model output is: "**)  
 print(output)  
  
**def** calculateAccuracy(ypredicted, yactual):  
 metrics = {}  
 metrics[**"tp"**] = 0  
 metrics[**"tn"**] = 0  
 metrics[**"fp"**] = 0  
 metrics[**"fn"**] = 0  
 **for** i **in** range(0, len(yactual)):  
 **if** ypredicted[i] == 0 **and** yactual[i] == 0:  
 metrics[**"tn"**] += 1  
 **elif** ypredicted[i] == 1 **and** yactual[i] == 0:  
 metrics[**"fp"**] += 1  
 **elif** ypredicted[i] == 0 **and** yactual[i] == 1:  
 metrics[**"fn"**] += 1  
 **elif** ypredicted[i] == 1 **and** yactual[i] == 1:  
 metrics[**"tp"**] += 1  
  
 accuracy = (metrics[**"tp"**] + metrics[**"tn"**]) / (metrics[**"tp"**] + metrics[**"tn"**] + metrics[**"fp"**] + float(metrics[**"fn"**]))  
  
 **return** accuracy  
  
**def** calculateMetrics(ypredicted, yactual):  
 metrics = {}  
 metrics[**"tp"**] = 0  
 metrics[**"tn"**] = 0  
 metrics[**"fp"**] = 0  
 metrics[**"fn"**] = 0  
 **for** i **in** range(0, len(yactual)):  
 **if** ypredicted[i] == 0 **and** yactual[i] == 0:  
 metrics[**"tn"**] += 1  
 **elif** ypredicted[i] == 1 **and** yactual[i] == 0:  
 metrics[**"fp"**] += 1  
 **elif** ypredicted[i] == 0 **and** yactual[i] == 1:  
 metrics[**"fn"**] += 1  
 **elif** ypredicted[i] == 1 **and** yactual[i] == 1:  
 metrics[**"tp"**] += 1  
  
 metrics[**"sensitivity"**] = float(metrics[**"tp"**]) / (float(metrics[**"tp"**]) + metrics[**"fn"**])  
 metrics[**"specificity"**] = float(metrics[**"tn"**]) / (float(metrics[**"tn"**]) + metrics[**"fp"**])  
 metrics[**"ppv"**] = float(metrics[**"tp"**]) / (float(metrics[**"tp"**]) + metrics[**"fp"**])  
 metrics[**"npv"**] = float(metrics[**"tn"**]) / (float(metrics[**"tn"**]) + metrics[**"fn"**])  
  
 **return** metrics  
  
**def** runNetTrial():  
 *# Build model.* mModel = NNModel.Model()  
 mModel.add(layer\_size=2, learning\_rate=.05, isInput=**True**)  
 mModel.add(layer\_size=20, learning\_rate=.05)  
 mModel.add(layer\_size=2, learning\_rate=.05)  
 print(**"Created Model."**)  
  
 data = pd.read\_table(**'./hw2\_dataProblem.txt'**, sep=**" +"**, engine=**'python'**)  
 *#Range scale the P data.* data[**"P"**] = data[**"P"**].apply(**lambda** item: (item - data.P.min()) / (data.P.max() - data.P.min()))  
 *#Range scale the L data* data[**"L"**] = data[**"L"**].apply(**lambda** item: (item - data.L.min()) / (data.L.max() - data.L.min()))  
  
 *#Split the data into training and test data sets.* train0, test0 = train\_test\_split(data[data.D == 0].values, test\_size = 0.2, random\_state=random.randint(0, 100000))  
 train1, test1 = train\_test\_split(data[data.D == 1].values, test\_size = 0.2, random\_state=random.randint(0, 100000))  
  
 *#Combine and shuffle the test and train examples.* testSet = np.vstack((test0, test1))  
 np.random.shuffle(testSet)  
 trainSet = np.vstack((train0, train1))  
 *#trainSet = np.vstack((trainSet, train0))* np.random.shuffle(trainSet)  
  
 testSetData = testSet[:,0:2]  
 testSetLabels = NNModel.labelToOneHotEncoding(testSet[:,2])  
 trainSetData = trainSet[:,0:2]  
 trainSetLabels = NNModel.labelToOneHotEncoding(trainSet[:,2])  
  
 print(**"Starting training."**)  
 trialWiseErrorList = mModel.train(trainSetData, trainSetLabels, epochs=200)  
 print(**"Training finished."**)  
  
 *# Predict the test set metrics* predictedLabels = mModel.predictAll(testSetData)  
 predictedLabels = NNModel.oneHotEncodingToLabels(predictedLabels)  
 accuracy = calculateAccuracy(predictedLabels, testSet[:,2].reshape((len(testSet), 1)))  
 testSetMetrics = calculateMetrics(predictedLabels, testSet[:,2].reshape((len(testSet), 1)))  
 testSetMetrics[**"accuracy"**] = accuracy  
  
 *# Predict the train set metrics* predictedLabels = mModel.predictAll(trainSetData)  
 predictedLabels = NNModel.oneHotEncodingToLabels(predictedLabels)  
 accuracy = calculateAccuracy(predictedLabels, trainSet[:,2].reshape((len(trainSet), 1)))  
 trainSetMetrics = calculateMetrics(predictedLabels, trainSet[:,2].reshape((len(trainSet), 1)))  
 trainSetMetrics[**"accuracy"**] = accuracy  
 trainSetMetrics[**"accuracyList"**] = trialWiseErrorList  
  
  
 *# Print model metrics.  
 # print("Predicted Labels:")  
 # print(predictedLabels)  
 # print("Accuracy on test set is: " + str(accuracy))  
 # print("Sensitivity: " + str(metrics["sensitivity"]))  
 # print("Specificity: " + str(metrics["specificity"]))  
 # print("ppv: " + str(metrics["ppv"]))  
 # print("npv: " + str(metrics["npv"]))* **return** mModel, trainSetMetrics, testSetMetrics  
  
  
**def** plotMetrics(metricList, modelName=**""**, numTrials=0):  
 fig = plt.gcf()  
 fig.set\_size\_inches(12, 7)  
 red\_patch = mpatches.Patch(color=**'red'**, label=**'Train'**)  
 blue\_patch = mpatches.Patch(color=**'blue'**, label=**'Test'**)  
  
 *#NN PLOTS.  
 #Plot the sensitivity* mplt = fig.add\_subplot(2,2,1)  
 xVals = np.arange(1, 10, 1)  
 testMetrics = []  
 trainMetrics = []  
 **for** i **in** range(0, numTrials):  
 testMetrics.append(metricList[i][2])  
 trainMetrics.append(metricList[i][1])  
 *#Make perceptron plot.* mplt.set\_title(modelName + **" Sensitivity"**)  
 mplt.xaxis.set\_ticks(xVals)  
 mplt.bar(xVals - 0.1, [item[**"sensitivity"**] **for** item **in** testMetrics], width=.2, color=**'b'**)  
 mplt.bar(xVals + 0.1, [item[**"sensitivity"**] **for** item **in** trainMetrics], width=.2, color=**'r'**)  
 mplt.legend(handles=[red\_patch, blue\_patch], loc=3)  
  
 *#Plot the Specificity* mplt = fig.add\_subplot(2,2,2)  
 *#Make perceptron plot.* mplt.set\_title(modelName + **" Specificity"**)  
 mplt.xaxis.set\_ticks(xVals)  
 mplt.bar(xVals - 0.1, [item[**"specificity"**] **for** item **in** testMetrics], width=.2, color=**'b'**)  
 mplt.bar(xVals + 0.1, [item[**"specificity"**] **for** item **in** trainMetrics], width=.2, color=**'r'**)  
 mplt.legend(handles=[red\_patch, blue\_patch], loc=3)  
  
 *#Plot the ppv* mplt = fig.add\_subplot(2,2,3)  
 *#Make perceptron plot.* mplt.set\_title(modelName + **" Ppv"**)  
 mplt.xaxis.set\_ticks(xVals)  
 mplt.bar(xVals - 0.1, [item[**"ppv"**] **for** item **in** testMetrics], width=.2, color=**'b'**)  
 mplt.bar(xVals + 0.1, [item[**"ppv"**] **for** item **in** trainMetrics], width=.2, color=**'r'**)  
 mplt.legend(handles=[red\_patch, blue\_patch], loc=3)  
  
 *#Plot the npv* mplt = fig.add\_subplot(2,2,4)  
 *#Make plot.* mplt.set\_title(modelName + **" Npv"**)  
 mplt.xaxis.set\_ticks(xVals)  
 mplt.bar(xVals - 0.1, [item[**"npv"**] **for** item **in** testMetrics], width=.2, color=**'b'**)  
 mplt.bar(xVals + 0.1, [item[**"npv"**] **for** item **in** trainMetrics], width=.2, color=**'r'**)  
 mplt.legend(handles=[red\_patch, blue\_patch], loc=3)  
  
 plt.show()  
  
**def** averagePerformance(metricList, numTrials=0):  
 testMetrics = []  
 trainMetrics = []  
  
 **for** i **in** range(0, numTrials):  
 trainMetrics.append(metricList[i][1])  
 testMetrics.append(metricList[i][2])  
  
 *#Metrics on the test set.* sen = np.array([item[**"sensitivity"**] **for** item **in** testMetrics])  
 spec = np.array([item[**"specificity"**] **for** item **in** testMetrics])  
 ppv = np.array([item[**"ppv"**] **for** item **in** testMetrics])  
 npv = np.array([item[**"npv"**] **for** item **in** testMetrics])  
 print(**"Test metrics."**)  
 print(**"Sensitivity: "** + str(sen.mean()) + **", "** + str(sen.std()))  
 print(**"Specificity: "** + str(spec.mean()) + **", "** + str(spec.std()))  
 print(**"ppv: "** + str(ppv.mean()) + **", "** + str(ppv.std()))  
 print(**"npv: "** + str(npv.mean()) + **", "** + str(npv.std()))  
 print(**"\n"**)  
  
 *#Metrics on the train set.* sen = np.array([item[**"sensitivity"**] **for** item **in** trainMetrics])  
 spec = np.array([item[**"specificity"**] **for** item **in** trainMetrics])  
 ppv = np.array([item[**"ppv"**] **for** item **in** trainMetrics])  
 npv = np.array([item[**"npv"**] **for** item **in** trainMetrics])  
 print(**"Train metrics."**)  
 print(**"Sensitivity: "** + str(sen.mean()) + **", "** + str(sen.std()))  
 print(**"Specificity: "** + str(spec.mean()) + **", "** + str(spec.std()))  
 print(**"ppv: "** + str(ppv.mean()) + **", "** + str(ppv.std()))  
 print(**"npv: "** + str(npv.mean()) + **", "** + str(npv.std()))  
 print(**"\n"**)  
  
**def** plotTrialError(metricList, numTrials=0):  
 plt.clf()  
 trialAccuracyList = []  
 *# Get the accuracy list from the training metrics dict in the metricsList obj.* **for** i **in** range(0, numTrials):  
 trialAccuracyList.append(metricList[i][1][**"accuracyList"**])  
  
 *#Plot the trial wise accuracy over time.* **for** i **in** range(0, numTrials):  
 pltLabel = **"Trial %s"** % str(i)  
 plt.plot(np.arange(0, len(trialAccuracyList[i])), trialAccuracyList[i], label=pltLabel )  
 legend(bbox\_to\_anchor=(1.05, 1), loc=2, borderaxespad=0., framealpha=0.5)  
 plt.title(**"Trial accuracy over time."**)   
 plt.show()  
  
 *#Plot the mean trial wise error, and plot the std dev as error bars.* avgList = []  
 stdList = []  
 **for** i **in** range(0, len(trialAccuracyList[0])):  
 temp = np.array(trialAccuracyList)  
 avgList.append(temp[:, i].mean())  
 stdList.append(temp[:, i].std())  
 plt.title(**"Average trial-wise error plot."**)  
 plt.errorbar(np.arange(0, len(trialAccuracyList[0])), avgList, yerr=stdList)  
 plt.show()  
  
**def** plotNN(mModel):  
 plt.clf()  
 print(**"Beginning best knn..."**)  
 *#Create a grid to classify over.* testSet = []  
 **for** x **in** np.arange(0, 1, 0.02):  
 **for** y **in** np.arange(0, 1, 0.02):  
 testSet.append([x, y])  
 testSet = np.array(testSet)  
  
 *#Classify over the grid.* predictedLabels = mModel.predictAll(testSet)  
 predictedLabels = NNModel.oneHotEncodingToLabels(predictedLabels)  
  
 *#Group together to be filtered by color.* data = pd.DataFrame(testSet, columns=[**'L'**, **'P'**])  
 data[**'D'**] = predictedLabels  
 posData = data[data.D == 1]  
 negData = data[data.D == 0]  
 plt.scatter(posData.L, posData.P, color=**"red"**)  
 plt.scatter(negData.L, negData.P, color=**"blue"**)  
 plt.title(**"Best Classifier Decision Boundary."**)  
 plt.show()  
  
**def** main():  
 **global** metricList  
 print(**"In main."**)  
 *#buildSmallExampleNet()  
  
 # Run program for trial wise metrics.* metricList = []  
 **for** i **in** range(0, 9):  
 print(**"Trial: "** + str(i))  
 metricList.append(runNetTrial())  
  
 *# Performance on individual trials.* print(**"Plotting metrics."**)  
 plotMetrics(metricList, **"NN"**, 9)  
  
 *# Average performance.* print(**"Printing average performance."**)  
 averagePerformance(metricList, 9)  
  
 *# Trial wise error and perceptron mean training error.* print(**"Plotting trial error."**)  
 plotTrialError(metricList, 9)  
  
 *# Plot decision boundary for NN.* print(**"Plotting decision surface."**)  
 bestAccuracy = 0  
 bestModel = **None  
 for** tup **in** metricList:  
 **if** tup[2][**'accuracy'**] > bestAccuracy:  
 bestAccuracy = tup[2][**'accuracy'**]  
 bestModel = tup[0]  
 plotNN(bestModel)  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

# NN\_library code

*'''  
This library has some parameters indicating the possibility of implementing backpropagation  
with alternate loss functions, activation functions, and optimizers. Currently, only  
backpropagation with SGD and an activation of sigmoids is used.  
'''***from** functools **import** \*  
**import** numpy **as** np  
**import** \_pickle **as** pickle  
**import** math  
  
**'''  
This class is used to build a neural network model.  
'''  
class** Model:  
 **def** \_\_init\_\_(self):  
 self.layers = []  
 self.inputSize = 0  
  
 **'''  
 Add a new fully connected layer to this model.  
 '''  
 def** add(self, layer\_size=1, learning\_rate=0.1, momentum\_factor=0, loss\_function=**"lms"**, optimizer=**"sgd"**, isInput=**False**):  
 *# Set input size and return.* **if** isInput:  
 self.inputSize = layer\_size  
 **return  
 else**:  
 newLayer = Layer()  
 **if** len(self.layers) == 0:  
 *# First layer, so use inputSize as input size value.* newLayer.setParams(input\_size=self.inputSize, size=layer\_size)  
 **else**:  
 newLayer.setParams(input\_size=self.layers[-1].size, size=layer\_size)  
 self.layers.append(newLayer)  
  
 **'''  
 Using the training set of data, run through each data example, and backpropogate the errors.  
  
 train\_set: (m x k) numpy array with m examples of dimension k  
 label\_set: (m x o) numpy array with m outputs of dimension o  
 '''  
 def** train(self, train\_set, label\_set, epochs=1):  
 accuracyList = []  
 actualLabels = oneHotEncodingToLabels(train\_set)  
  
 **for** epoch **in** range(0, epochs):  
 print(**"Epoch: "** + str(epoch))  
 *# Generate the trial wise error and add it to a list to return every 10th epoch.* **if** epoch % 10 == 0:  
 predictedLabels = self.predictAll(train\_set)  
 predictedLabels = oneHotEncodingToLabels(predictedLabels)  
 accuracy = calculateAccuracy(predictedLabels, actualLabels)  
 accuracyList.append(1.0 - accuracy)  
  
  
 **for** train\_index **in** range(0, len(train\_set[:])):  
 **'''  
 1) Predict current example.  
 2) Calculate error for last level.  
 3) For each level before last level:  
 (In vector form)  
 A) levelError = (levelOutput)\*(1 - levelOutput)\*(Sum of next level's weights \* next level's errors)  
 B) Calculate delta wji (With momentum)  
 C) Update weight wji as wji(n) = wji(n-1) + delta wji  
 '''** prediction = self.predict(train\_set[train\_index])  
 *# Calculate error term for every output neuron. Dims (1 x o)* error = np.array([(label\_set[train\_index] - prediction)])*#np.array([(prediction)\*(1.0 - prediction)\*(label\_set[train\_index] - prediction)])* errorMat = error  
 *# Backpropogate errors for each layer.* **for** layer\_index **in** range(len(self.layers) - 1, -1, -1):  
 error = self.layers[layer\_index].backpropogateErrors(errorMat)  
 errorMat = self.layers[layer\_index].generateErrorMat(error)  
 *# Continue to next example.* print(**"Finished Training."**)  
 **return** accuracyList  
  
 **'''  
 Generate an output prediction from the NN model using the training data and current network weights.  
 The data should be an np array with dims (1 x k), where k is the number of inputs specified in the input  
 layer when the model was being built.  
 '''  
 def** predict(self, data):  
 *'''  
 1) Set current data numpy matrix.  
 2) For each layer in the net:  
 Expand inputs so that they can be passed into the layer.  
 Pass inputs to the layer.  
 The layer will apply a dot product and activation function to generate the outputs, and store the output vector.  
 3) Return the final output vector, and threshold if necessary.  
 '''  
  
 # The input data is not of the proper size, so error out.* **if** len(data) < self.inputSize:  
 **raise** AttributeError(**"Input data size not equal to weight input size."**)  
  
 *# Initialize the pLayerOutput to the input data\_set for the looped dot product code.* pLayerOutput = data  
 nLayerOutput = **None** *# For each layer, compute the dot product of the pLayerOutput and the input weights of each neuron of the layer.* **for** cLayer **in** self.layers:  
 nLayerOutput = cLayer.generateOutput(pLayerOutput)  
 pLayerOutput = nLayerOutput  
  
 **return** pLayerOutput  
  
 **def** predictAll(self, data):  
 labels = []  
 **for** entry **in** data[:]:  
 labels.append(self.predict(entry))  
  
 **return** np.array(labels)  
  
**'''  
Convert a one-hot encoding of the classes to a numerical number from 0 to the number of classes - 1.  
'''  
def** oneHotEncodingToLabels(labels):  
 newLabels = np.zeros((labels.shape[0], 1))  
 **for** index **in** range(0, labels.shape[0]):  
 argMax = np.argmax(labels[index])  
 newLabels[index] = np.array([argMax])  
 **return** newLabels  
  
**'''  
Convert a set of labels into a one-hot encoding with smallest number in bit position 0, and largest  
number in the last bit position.  
'''  
def** labelToOneHotEncoding(labels):  
 uniqueValues = sorted(list(set(labels)))  
 newLabels = np.zeros((labels.shape[0], len(uniqueValues)))  
 **for** label\_index **in** range(0, len(labels[:])):  
 value\_index = uniqueValues.index(labels[label\_index])  
 *# Flip the bit corresponding to the position of the element. Values are encoded in descending order.  
 # Aka, smalles value is bit in first position, and largest value is bit in last position.* newLabels[label\_index, value\_index] = 1  
 **return** newLabels  
  
**'''  
Given a predicted and actual set of labels, determine the accuracy of the list.  
'''  
def** calculateAccuracy(ypredicted, yactual):  
 metrics = {}  
 metrics[**"tp"**] = 0  
 metrics[**"tn"**] = 0  
 metrics[**"fp"**] = 0  
 metrics[**"fn"**] = 0  
 **for** i **in** range(0, len(yactual)):  
 **if** ypredicted[i] == 0 **and** yactual[i] == 0:  
 metrics[**"tn"**] += 1  
 **elif** ypredicted[i] == 1 **and** yactual[i] == 0:  
 metrics[**"fp"**] += 1  
 **elif** ypredicted[i] == 0 **and** yactual[i] == 1:  
 metrics[**"fn"**] += 1  
 **elif** ypredicted[i] == 1 **and** yactual[i] == 1:  
 metrics[**"tp"**] += 1  
  
 accuracy = (metrics[**"tp"**] + metrics[**"tn"**]) / (metrics[**"tp"**] + metrics[**"tn"**] + metrics[**"fp"**] + float(metrics[**"fn"**]))  
  
 **return** accuracy  
  
**'''  
  
'''  
class** Layer:  
  
 **def** \_\_init\_\_(self):  
 *# Each column represents the weights of a neuron. Column 0 are the input weights of neuron 0. Column 1 are the input weights  
 # of neuron 1 and so on.* self.input\_weights = **None** self.input\_weight\_deltas = **None** self.output = **None** self.input = **None** self.size = 0  
 self.momentum = 0  
 self.learning\_rate = 0.1  
  
 **def** setParams(self, input\_size, size, momemtum=0, learning\_rate=0.1, activation\_function=**'sigmoid'**):  
 *# Weight matrix. (# weights or inputs, # neurons). (k x H).* self.size = size  
 self.input\_weights = np.random.rand(input\_size, size)*#np.zeros((input\_size, size))* self.input\_weight\_deltas = np.zeros((input\_size, size))  
 self.output = np.zeros((size, 1))  
 self.momentum = momemtum  
 self.learning\_rate = learning\_rate  
 *#* ***TODO: Potentially allow an activation function to be passed, or set using the activation function param.* '''  
 1) Do dot product of input (1xk) and weight matrix (kxH)  
 2) Store output as copy in layer output ndarray.  
 3) Return output in the form or (1xH)  
 '''  
 def** generateOutput(self, input):  
 *# Set the input for the backprop to use later. (1 x k) vector.  
 #* ***TODO: Make sure input isn't being changed by any other func.*** self.input = np.array([input])  
 *# Generate output.* output = np.dot(input, self.input\_weights)  
 *# Apply sigmoid function, and reset values of output so that mem doesn't have to be allocated.* **for** col **in** range(0, len(output)):  
 *# Set the output with dim (1 x H) values to the layer's output var with dims (H X 1)* output[col] = 1.0 / (1.0 + math.exp(-1.0 \* output[col]))  
 *# Set the output to the output term that will be used later in backprop.* self.output[col] = output[col]\*(1.0 - output[col])  
  
 **return** output  
  
 **'''  
 Return the input weight vectors for this layer, stacked horizontally.  
 '''  
 def** getInputWeights(self):  
 **return** self.input\_weights  
  
 **'''  
 Return the input weight vector for a neuron in this layer, from neuron 0  
 through neuron (layer\_size - 1).  
 '''  
 def** getInputWeightsForNeuron(self, neuron):  
 **pass  
  
 '''  
 The error matrix is the transpose of the input matrix, with each column multiplied by the  
 error term for that output neuron. It has dims (1 x H)  
 '''  
 def** backpropogateErrors(self, errorMat):  
 *# Calculate the new delta's for this layer. It should be (H x 1) \* (1 x H).T  
 # Note, these intermediate numpy arrays are necessary for the transpose operations to work.  
 # mErrorMat = errorMat.reshape((len(errorMat), 1))  
 #mInput = self.input.reshape((len(self.input), 1))* error = self.output \* errorMat.T  
 *# Calculate the new weight deltas along with momentum. Make input of form (k x 1) and error of form (1 x H)* self.input\_weight\_deltas = (self.learning\_rate \* np.dot(self.input.T, error.T)) + (self.momentum\*self.input\_weight\_deltas)  
 *# Update the weights. input\_weight\_deltas should still be a (k x H) weight matrix.* self.input\_weights = self.input\_weights + self.input\_weight\_deltas  
  
 *# Return the error.* **return** error.T  
  
 **def** generateErrorMat(self, error):  
 *# Calculate the errorMat to use for backpropagation.* errorMat = np.dot(error, self.input\_weights.T)  
 **return** errorMat  
**'''  
Save the model in the specified file path as a pickled object.  
'''  
def** save(model, file\_path):  
 f = open(file\_path, **'wb'**)  
 pickle.dump(model, f)  
 f.close()  
  
**'''  
Return the model saved in the specified pickled file.  
'''  
def** load(file\_path):  
 f = open(file\_path, **'rb'**)  
 model = pickle.load(f)  
 f.close()  
 **return** model