Intelligent Systems HW3 Part 2

# Problem Description

Given a data set of 5000, 28x28 pixel hand written digits, develop a neural network to classify the digits. Use 4000 images for training, and 1000 images for testing.

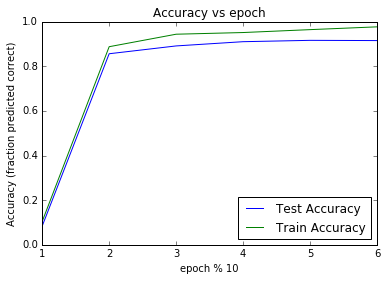
# Network Code Description

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. The initial weights are chosen at random between -0.05 and 0.05 for each layer.

Through much trial and error, the final neural network was set with an input of 784 inputs, 200 neurons in the hidden layer, and 10 neurons in the output layer. The learning rate for the first layer is .01, and the rest of the layers have a learning rate of 0.005. All layers have a momentum of 0. The total number of training epochs was set at 60 epochs, and the batch size for training was set as a random sampling of 2000 images. The image labels 0 through 9 are encoded in a one-hot encoding for training. The predictions given by the network are also in a one-hot encoding form. As an example, 1 is encoded as [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]. It was decided to stop training at 60 epochs because this was where the training and test accuracy started to diverge. The training error continued decreasing, while the testing error started increasing, indicating overtraining.

# Results

The final network error on the training set was 97.77% and the final network error on the test set was 91.8%. It took 8 minutes to train the network on an 8 core machine.



# Analysis of Results

Describe, discuss, and interpret the results you got, and why you think they are as they are.

Initially, the network was not changing accuracy enough, and continually had an accuracy of 10%. So, the learning rate was decreased. The initial momentum was also decreased to 0. As the learning rate was decreased, the accuracy started to change more, but didn't adjust enough. So, the network was made more expressive through adding more layers and larger layers. The data input size for training was also reduced and shuffled during every epoch. After adjusting the network size, the training and testing error started to increase. The network with one hidden layer maxed out at 91.8%. The network performance could potentially be made better by increasing the number of layers and reducing the size of each layer, while also reducing the number of connections between neurons in consecutive layers. This would allow for more local feature learning, as well as less of a chance of over training. The current network architecture is limited by the issue of over training. To reduce this concern, the number of parameters in the model could be decreased, along with further optimization of the network parameters.

# Code

**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** \*  
**import** time  
  
  
**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=784, learning\_rate=.01, isInput=**True**)  
 mModel.add(layer\_size=200, learning\_rate=.005, momentum\_factor=0)  
 *#mModel.add(layer\_size=100, learning\_rate=.005, momentum\_factor=0)* mModel.add(layer\_size=10, learning\_rate=.005, momentum\_factor=0)  
 print(**"Created Model."**)  
  
 *# Read data from file.* xData = pd.read\_csv(**'./MNISTnumImages5000.txt'**, sep=**'\t'**, header=**None**)  
 yData = pd.read\_csv(**'./MNISTnumLabels5000.txt'**, header=**None**, names=[**'labels'**])  
 xData[**'labels'**] = yData.values  
 *# Break data into train and test sets.* **import** random  
 trainSet, testSet = train\_test\_split(xData, test\_size=0.2, random\_state=random.randint(0, 100000))  
  
 *# Break data into train and test sets.* originalTrainLabels = trainSet[**'labels'**].values  
 originalTestLabels = testSet[**'labels'**].values  
 trainLabels = NNModel.labelToOneHotEncoding(originalTrainLabels)  
 testLabels = NNModel.labelToOneHotEncoding(originalTestLabels)  
 trainData = trainSet[trainSet.columns[:-1]].values  
 testData = testSet[testSet.columns[:-1]].values  
  
 print(**"Starting training."**)  
 trialWiseErrorList = mModel.train(trainData, trainLabels, validation\_data\_set=testData, validation\_label\_set=originalTestLabels, epochs=60)  
 print(**"Training finished."**)  
  
 *# Predict the test set metrics* predictedLabels = mModel.predictAll(testData)  
 predictedLabels = NNModel.oneHotEncodingToLabels(predictedLabels)  
 accuracy = NNModel.calculateAccuracy(predictedLabels, originalTestLabels)  
 *#testSetMetrics = calculateMetrics(predictedLabels, originalTestLabels)* testSetMetrics = {}  
 testSetMetrics[**"accuracy"**] = accuracy  
  
 *# Predict the train set metrics* predictedLabels = mModel.predictAll(trainData)  
 predictedLabels = NNModel.oneHotEncodingToLabels(predictedLabels)  
 accuracy = NNModel.calculateAccuracy(predictedLabels, originalTrainLabels)  
 *#trainSetMetrics = calculateMetrics(predictedLabels, originalTrainLabels)* trainSetMetrics = {}  
 trainSetMetrics[**"accuracy"**] = accuracy  
 trainSetMetrics[**"accuracyList"**] = trialWiseErrorList  
  
 print(**"Orig labels: "**)  
 print(originalTrainLabels[0:20])  
 print(**"Pred labels: "**)  
 print(predictedLabels[0:20])  
  
 **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** mModel, trainMetrics, testMetrics  
 print(**"In main."**)  
  
 *# Run program for trial wise metrics.* initTime = time.time()  
 print(**"Training net..."**)  
 mModel, trainMetrics, testMetrics = runNetTrial()  
 print(**"Finished training!"**)  
 finTime = time.time()  
 print(**"Total time: "** + str((finTime - initTime) / 60.0) + **" min."**)  
  
 print(**'Train metrics:'**)  
 print(trainMetrics)  
 print(**'Test metrics: '**)  
 print(testMetrics)  
  
 NNModel.save(mModel, **'./MINST\_MODEL.pk'**)  
 mModel = NNModel.load(**'./MINST\_MODEL.pk'**)  
  
 print(**"Saving metrics."**)  
 NNModel.save(trainMetrics, **'./MINST\_MODEL\_TRAIN\_METRICS.pk'**)  
 NNModel.save(testMetrics, **'./MINST\_MODEL\_TEST\_METRICS.pk'**)  
 trainMetrics = NNModel.load(**'./MINST\_MODEL\_TRAIN\_METRICS.pk'**)  
 testMetrics = NNModel.load(**'./MINST\_MODEL\_TEST\_METRICS.pk'**)  
 print(**"Saved and reloaded metrics."**)  
   
   
 trainA = list(map(**lambda** item: item[**'train\_accuracy'**], trainMetrics[**'accuracyList'**]))  
 testA = list(map(**lambda** item: item[**'test\_accuracy'**], trainMetrics[**'accuracyList'**]))  
 plt.plot(np.arange(1, 7), testA, label=**'Test Accuracy'**)  
 plt.plot(np.arange(1, 7), trainA, label=**'Train Accuracy'**)  
 plt.xlabel(**'epoch % 10'**)  
 plt.ylabel(**'Accuracy (fraction predicted correct)'**)  
 plt.title(**'Accuracy vs epoch'**)  
 legend(loc=0)  
 plt.show()  
   
  
 print(**"End of main."**)  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

*'''  
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, validation\_data\_set=**None**, validation\_label\_set=**None**, epochs=1):  
 accuracyList = []  
  
 **for** epoch **in** range(0, epochs):  
 randomIndecies = np.random.random\_integers(0, train\_set.shape[0] - 1, 2000)*#train\_set.shape[0])* mTrainSet = train\_set[randomIndecies]  
 mLabelSet = label\_set[randomIndecies]  
 actualLabels = oneHotEncodingToLabels(mLabelSet)  
  
 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(mTrainSet)  
 predictedLabels = oneHotEncodingToLabels(predictedLabels)  
 accuracy = calculateAccuracy(predictedLabels, actualLabels)  
 **if** validation\_label\_set **is None**:  
 accuracyList.append(1.0 - accuracy)  
 **else**:  
 *# Calculate the accuracy of the network on the validation set.* predictedLabels = self.predictAll(validation\_data\_set)  
 predictedLabels = oneHotEncodingToLabels(predictedLabels)  
 validationAccuracy = calculateAccuracy(predictedLabels, validation\_label\_set)  
 accuracyList.append({**'train\_accuracy'**: accuracy, **'test\_accuracy'**: validationAccuracy})  
 print(**"Accuracy: "**)  
 print(accuracyList)  
  
  
 **for** train\_index **in** range(0, len(mTrainSet[:])):  
 **'''  
 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(mTrainSet[train\_index])  
 *# Calculate error term for every output neuron. Dims (1 x o)* error = np.array([(mLabelSet[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] == yactual[i]:  
 metrics[**"tp"**] += 1  
  
 accuracy = (metrics[**"tp"**]) / (len(ypredicted))  
  
 **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  
 posNegArr = [1, -1]  
 initRandWeights = np.random.rand(input\_size, size)  
 **for** row **in** range(0, initRandWeights.shape[0]):  
 **for** col **in** range(0, initRandWeights.shape[1]):  
 initRandWeights[row, col] = initRandWeights[row, col]\*posNegArr[np.random.randint(0, 2)]\*0.05  
  
 self.input\_weights = initRandWeights*#np.random.rand(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