Intro to AI - Final Project Report

Names & RUIDs:
Ananya Balaji - 215007455
Shatakshi Ranjan - 210005870
Sahana Senthilkumar - 218000767

May 4, 2024

Perceptron

Algorithm Implementation

- The pixels in each digit and face image are fed directly into the perceptron algorithm via basicFeatureExtractorDigit and basicFeatureExtractorFace in dataClassifier.py. Each face image is inputted as a 60 x 70 feature image and each digit image is inputted as a 28 x 28 feature image. basicFeatureExtractorDigit returns the color of each pixel (1 for black, 0 for white) and basicFeatureExtractorFace returns whether a pixel is an edge (1) or not (0).
- Weights are initialized randomly between 0 and 1.
- MNIST data: The train method iterates through the training data by the number of times inputted in the command line following "-t". Each iteration, the scoring functions $f(x)$ or dot products of the features (trainingData[i]) and weights (self.weights[label]) are computed for each digit, and the digit label with the highest dot product is selected as the predicted label. If the predicted label is not the same as the actual label, then the weights are updated accordingly. Specifically, the weights are incremented for features that are in the actual label but not in the predicted and decremented for features that are not in the actual label but are in the predicted.
- Face data: If the scoring function $f(x)$ incorrectly predicts that an image is a face, then the corresponding features are removed from the weight so that it is less likely to predict the same outcome during the next training iteration. If $f(x)$ predicts that there is no face when there is, then the corresponding features are added to the weight so that it is more likely to predict that there is a face during the next training iteration.

- Each digit has a separate scoring function when training on MNIST data, whereas for face data, all images use the same scoring function with different weights.
- The classify method predicts labels for images by calculating the dot product of trained weights (self-weights[l]) and the input image's features (datum). The label with the highest result is finally returned as the predicted label.
- The findHighWeightFeatures method returns a list of the 100 features with the greatest weight for a particular label.

Results

- Example command line input for 90% digit training data, trained on 5 iterations, outputting standard deviation and mean accuracy rates:
  `python dataClassifier.py -c perceptron -d digits -r -i 5 -t .9`
- Command line perceptron testing results:
  https://docs.google.com/document/d/1Z6xaz2en0rGRddws3EyN1mHm4XXS0sxGBKWRel8yY3k/edit?usp=sharing
- Visualizations of perceptron results:
  https://docs.google.com/spreadsheets/d/10GFKrVResPh5sb1PmPgtFGFvXAjQquQPO94NMHVH4jY/edit?usp=sharing

Two-layer Neural Network

Algorithm Implementation

- Results yielding an accuracy rate of higher than 70% were observed when using 20 nodes per hidden layer, so we initialized the network with 20 hidden nodes. We observed higher accuracy rates (closer to 80%) with 30 hidden nodes, but we ran our code with 20 to keep the runtime relatively low and manageable.
- Weights are initialized randomly for the nodes connecting the input layer to the hidden layer (weights_hidden) and nodes connecting the hidden layer to the output layer (weights_output).
- The sigmoid method defines the calculation for the activation function ($\frac{1}{1+e^{-x}}$) and the sigmoid_derivative method calculates the derivative of sigmoid with respect to its input. The derivative is used to calculate the activation of dot products during the forward pass inside the training function.
- Each iteration of training consists of a forward and backward pass:

  - The forward pass computes the dot product of input features with their corresponding weights and applies the sigmoid activation function. It also computes the dot product of the hidden node values with their corresponding weights and applies sigmoid to obtain the output nodes.
  - The backward pass calculates the error between the predicted and actual results and updates the weights from the hidden nodes to output nodes and from the input nodes to hidden nodes depending on the error.

- The classify method uses the trained weights to construct the optimal network. It computes the sigmoid of the dot products of input features with their weights and hidden nodes with their weights and returns the label with the highest output value as the predicted label.

- findHighWeightFeaturesHidden returns a list of features with the greatest weight connected to a hidden unit and findHighWeightsFeaturesOutput does the same for the greatest weight connected to an output label.

Results

- Example command line input for 90% digit training data, trained on 5 iterations, outputting standard deviation and mean accuracy rates:
  `python dataClassifier.py -c two-layer-network -d digits -r -i 5 -t .9`

- Command line NN testing results:
  https://docs.google.com/document/d/1gMwr6ygpIkXCE16NJiODzTQlDDyANS2dj8Gc1Mmovnk/edit?usp=sharing

- Visualizations of NN results:
  https://docs.google.com/spreadsheets/d/14xPXATkBWizxMMSYma504SpN6anAKMR3zqNQNJ3l_JA/edit?usp=sharing

Observations and Analysis

- For both classifiers, the standard deviation in the testing accuracy rate decreases, the mean in accuracy rate increases, and the average training time increases as the percentage of training data used increases.

- The training data is randomly selected before training the classifiers each time the training and testing cycle occurs. Therefore, the smaller the sample of data, the more likely it is that there will be a large variance and standard deviation in the testing accuracy. This explains the decrease in standard deviation of testing accuracy as the percentage of training data used increases.

- The mean accuracy is expected to increase as the percentage of data increases as using more data will prepare the classifier better for a variety of testing data. The average training time is also expected to increase as the weights must adjust to a larger pool of training data.
- The neural network generally took longer to train than the perceptron classifier, likely due to the need to perform a number of calculations during each weight update. During each training iteration, calculations are performed for each weight connecting each input node to each of the 20 hidden nodes and for each weight connecting each hidden node to each output node.
- Testing accuracy is generally higher for both classifers when using the face data, likely because multiclass classification is more complicated and involves more calculations than binary classification.
- Training on face data took less time than training on digit data due to the larger number of data points provided (5000 for digit data, 451 for face data).