**Part 1: Implementation**

Data structure

First, each image file is transformed into a row vector with 960 attributes. As there are 184 training data, the dimensions of input array and target array are 184 times 960 and 184 times 1, respectively.

Layer is the main component in the model. Except the input layer, all the layers, including the output layer, are of class layer. As input of each layer is output of the previous layer as well as output and weight of each layer are needed when doing backpropagation, it is essential that we store them properly. Class layer contains several instance variables, such as weight, z (output) and diff (gradient of the layer), which are all stored as numpy array structures. In addition, a layer has two methods, activationFunction and diffOfactFunc, to track its activated function. It is designed to solve the problem when each layer has a different activated function, but only the sigmoid function is used in every layer in this assignment.

Output (z) of each layer is calculated after applying sigmoid function on s, which is multiplication of output of previous layer and weight of current layer. The dimension of weight is size of previous layer times size of current layer; the dimension of s and z is number of training data times size of current layer.

Furthermore, the gradient of each layer is computed backwardly, so instance variable “diff” is created to store gradients of each data point and each perceptron. It is a numpy array with dimension number of training data times the size of the current layer.

Code-level optimizations

At first, the weight was updated after calculating the average of gradient of every training data, which is known as Vanilla gradient descent. However, it was inefficient as it calculated the gradients of the whole dataset for just an update. After 1000 epochs, as the accuracy rate was not satisfying, we then implemented mini-batch gradient descent that updates the weights after calculating a (random) batch of dataset. In this assignment, we found that batch size of five leads to faster, stable convergence and gets 100% accuracy rate of training data after 1000 epochs or less.

Moreover, we tried to optimize gradient descent function by implementing “adagrad”. The option can be selected when initializing the model through parameter “grad\_opt”. It is discovered that adagrad works well after scaling the dataset. The loss dropped dramatically within hundred epochs and is smaller than the previous model after 1000 epochs.

Challenge

Implementing the backpropagation was the biggest challenge. Although Professor Satish has provided us with the formula and the proof of it during the class, it took us several hours to derive it ourselves. After totally understanding how the formula is derived, the next task was how to program it with matrix computation.

Initially, errors occurred when executing the backpropagation part, and they were about wrong matrix shapes when doing matrix multiplication. Once part of the code was revised, it turned out that the other part of the programming couldn’t execute successfully. After a few days, we wrote down the shape of matrices and the formula of the algorithm as well as the desired results, and then figured out what the wrong parts were. It was mainly due to the ignorance of that weight should be updated by the average of gradients in each batch and the wrong shape of matrix was used to update the weight matrix. We managed to run the program successfully after revising the codes.

Results

The results are presented as below. The batch size and the learning rate of both results are five and 0.1. Figure 1 shows that the loss decreases significantly within the hundred epochs if using adagrad and the final loss is smaller than the outcome of using stochastic gradient descent. Nevertheless, though the accuracy of training data is 100% of both methods, the predictions of mini-batch gradient descent achieve better accuracy rate, 95.18%, on the testing data. I think that is due to the overfitting happening when loss of the training data becomes too low.

The predictions using mini-batch gradient descent of testing data are

[1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0]

where the red color point is misclassified, and “1” means “down” gesture, “0” means other gestures.

Additionally, it is noticed that there appear numerous highs and lows in both curves. It is due to the reason we update the weights after each batch instead of the whole dataset. The curve will be smooth if the weights are updated using vanilla gradient descent.

Figure 1

