Assignment 5 Group Report - Neural Network

Group Members:
Che-Pai Kung 5999-9612-95
Chenqi Liu 2082-6026-02
Mengyu Zhang 3364-2309-80

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 by 960 and 184 by 1 respectively.

Layer is the main component in the model. Except the input layer, all the other 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 the 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 we were unsatisfied with the accuracy rate, 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.

<u>Challenge</u>

Implementing the backpropagation was the biggest challenge. Although Professor Satish has provided us with the formula and the proof of it during 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 the shape of matrices and the formula of the algorithm as well as the desired results. Then, we figured out the error. We were initially unaware that weight should be updated with the average of gradients in each batch and the wrong shape of the matrix was used to update the weight matrix. We managed to run the program successfully after revising the codes.

<u>Results</u>

The results are presented below. The batch size and the learning rate of both results are 5 and 0.1. Figure 1 shows that the loss decreases significantly within the hundredth epochs if using adagrad and the final loss is smaller than the outcome of using stochastic gradient descent. Nevertheless, though the accuracy of the training data is 100% for 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 the loss of the training data becomes too low.

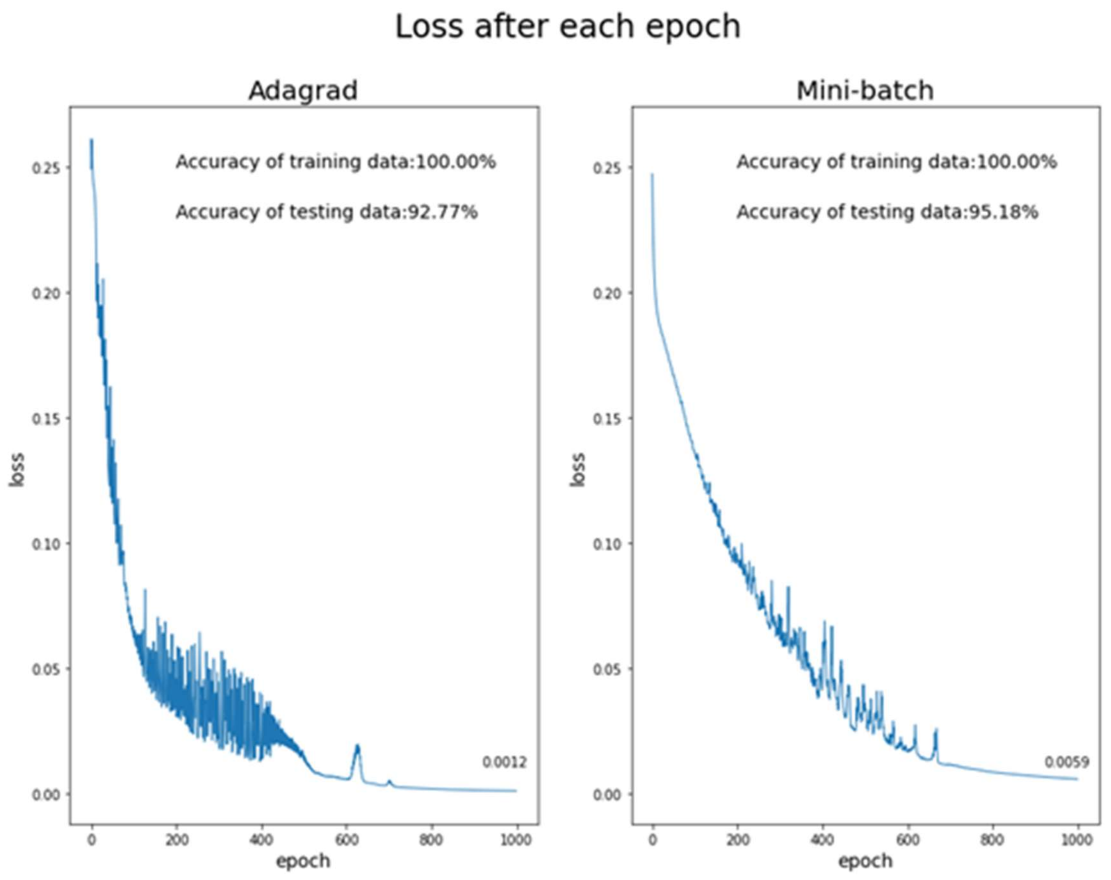
The predictions using mini-batch gradient descent of testing data are

[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] (Accuracy rate: 95.18%)

where the points marked red are misclassified, and “1” represents “down” gesture, while “0” represents other gestures.

Additionally, it is noted that there appears to be numerous highs and lows in both curves. It is due to the fact that we updated 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



Part 2: Software Familiarization

Implementation

The library we used this time is scikit-learn.neural_network.MLPClassifier. This library is used for the multi-layer perceptron classifier which is exactly what we are looking for in this assignment. The function has several parameters that are related to our research. 'hidden_layer_sizes' is used for setting the number of neurons and the number of hidden layer, 'activation' determines the activation function for the hidden layer, 'solver' is for weight optimization, 'learning_rate' schedules the weight updates, 'max_iter' determines the maximum number of iterations, 'batch_size' is for setting the size of minibatches for stochastic optimizers, and 'tol' is the tolerance for the optimization.

In the first part of the program, we try to match the function of python library with that of our algorithm implementation for convenience of comparison. As a result, we use 'hidden_layer_sizes' of (100,1), 'solver of 'sgd', 'activation' of 'logistic' for sigmoid function, 'learning_rate' of 0.1, 'max_iter' of 1000, 'batch_size' of 10, and 'tol' of 1e-7. The result and accuracy show as following:

Prediction: [1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0]

Accuracy: 0.7951807228915663

Then, we try to optimize the performance of the function like what we do in the implementation part. We use the Hyper-parameters to accomplish this objective. Hyper-parameters are parameters that are not directly learnt within estimators. In scikit-learn they are passed as arguments to the constructor of the estimator classes. Learning from the scikit-learn site, it is possible and recommended to search the hyper-parameter space for the best cross validation score. We keep the parameters that have specific requirements unchanged and let the program search for the best values for other parameters.

The result is as following:

Prediction: [1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 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]

Accuracy: 0.9397590361445783

The result and accuracy perform much better than the previous try. The accuracy number rises above 93% which is within a satisfiable range.

<u>Comparison and Inspiration</u>

Comparing the function of scikit-learn library with our implementation, the initial performances of them are very similar when they set the same values for parameters. Then for the implementation part, we try two approaches to improve the gradient descent function to optimize final performance. For the software familiarization part, as for the function is already written inside, we cannot do much about it, we turn to Hyper-parameter space to let the program search for the best parameters comprehensively to optimize the final performance. Both approaches reach our expectations. Inspiring from the python library function, besides gradient function, we can also try improving other parameters in our algorithm instead of setting fixed numbers. This would be our next step for optimization if we make further study on this topic.

Part 3: Applications

The neural network has been trending in not only Computer Science, but also various fields like biology, physics and etc. So does the feed forward neural network because of its advantage in adopting complex functions easily.

For example, in Computer Science, the feed forward neural network did a great job on X-ray image fusion.In article *The application of feed-forward neural network for the X-ray image fusion* published by Zhang Jian and Wang Xue Wu, they designed a three-layer feed-forward neural network in the field of X-ray fusion with a complex system and nonlinear function. Compared with traditional methods, the method with a neural network does a much better job based on the image qualities.

Also, Mani and Srinivasan published another research called *Application of Artificial Neural Network Model for Optical Character Recognition* in 1997. They successfully increase the noisy rate from 70% to 99% by implementing the feedforward neural network. Optical character Recognition has an essential influence in image recognition such as digitalizing the historical reference, newspaper and etc, which enable searching easier no matter for academic use or other convenience on research.

Part 4: Individual Contributions

● Model discussion: Che-Pai Kung, Chenqi Liu, Mengyu Zhang

● Model implementation: Che-Pai Kung

● Model optimization: Che-Pai Kung

● Software Familiarization: Chenqi Liu, Mengyu Zhang

● Applications: Chenqi Liu, Mengyu Zhang