Name: Zheng Leqian
SID: 55199789

The Report of Project 2

In the project 2, I used the programming language Python with Jupyter Notebook to implement the algorithm.

The Problem Description

The artificial neural network and genetic algorithms are two popular techniques used in optimization and learning. They have evolved for years and each has its own strengths and weaknesses. In this project, we will combine two technology to train a multi-layer perceptron to do classification on the given diabetes dataset and use it to help predicting unseen data. People usually train ANN with gradient-descent-based backpropagation, however, we will use genetic algorithms to train an ANN and observe its classification performance.

The Inputs and Outputs of the ANN

The inputs of the ANN are several 8-dimension vectors[1] (thus, constitute an $n*8$ matrix) in which each dimension represents a measured value with medical significance, and each vector is a record of a person. After processed by its several hidden layers and activation functions, the ANN will output an $n*2$ matrix, in which each row corresponds to each input vector and two columns are probabilities that he/she is/isn't diabetic.

Dataset Operations

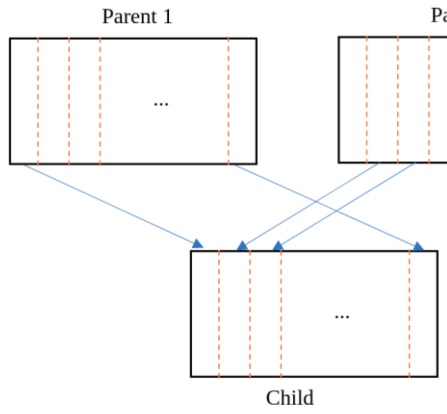
We discard the first column in the given dataset to keep consistent processing in each layer which will be introduced later. We extracte the last two columns as the ground truth labels.
We use k-fold cross validation method with k=6 over the whole dataset to evaluate the ANN to reduce the impact of the overfitting and get an accuracy close to that of predicting unseen data.
The ANN training process will repeat 6 times and 5/6 of the whole dataset will be used as training data and the remaining 1/6 will be used to validation and we record the accuracy of the model on the validation set in each iteration. In this method, each data point will be used in training 5 times and gets to be tested exactly once. At last, we computed the mean value of the accuracy on each validation set.
We also use data batches to train our neural network. We first divide the whole dataset into 600 training data and 168 testing data. Then in each training iteration, we randomly choose a small batch data from the training set to train the ANN.

Implementation Details (Training Method)

Firstly, we define the *forward(Ws, data, label, activation = "sigmoid", dropP=1)* function for ANN to do forward process, where *Ws* is the weight matrices in each layer, *data* is the input data, *label* is the ground truth output. We also provide activation functions *sigmoid*, *tanh*, *Gaussian* and *ReLU*. In each layer, we will add a column consisting of all 1 into the left side of the previous layer output (the input data for the first hidden layer) for conveniently adding bias vector, then we do dropout (will be introduced in ANN Generalization section) with the probability p and matrix

[1] The 1st column in the dataset is to help computing bias, the last two columns are the one-hot encoding label.

multiplication followed by an activation function. After getting the output from the ANN, we computed the L2 loss, the prediction as well as the accuracy.
Secondly, we step into GA training process. We use function *initialPopulation(popSize, NNPar)* where *popSize* is the population size and *NNPar* is a tuple of the ANN parameters to initialize the population. And we will invoke *initIndividual(NNPar)* to produce each ANN weight matrices group, the *hidden_layers* is the list of the number of neurons in each layer, and we will add one row to each weight matrix to as bias vectors as described before.
After getting the initial population, we start to simulate the evolving process. In each generation, we first preserve the top *elitePer*% highest fitness individuals (the fitness is the accuracy of an ANN evaluated on the given data). Then we do crossover on the population. The individual with higher fitness will be more likely to be chosen as the parent. And the crossover process is to randomly choose each column from two parents as a column is the weights linking to a neuron. Here's the illustration:



By doing so, we also omit the cost of flatten a matrix to a vector and reshape a vector back to the matrix. According to the research of David J. Montana[2], the performance of two crossover way is similar.
Then we do mutation on each child with probability *mutation_rate*. The mutation is to randomly use random values chosen from the initialization probability distribution to replace some entries. Though David J. Montana[3] said biased ed-mutate-weights will be much better than unbiased mutate weights. I did 10 experiments in each method respectively and found the accuracy of the two methods are similar, sometime unbiased mutation even did a little bit better.

Mutation Method	*Average Accuracy*
Unbiased mutation	79.63%
Biased mutation	78.72%

Lastly, we put elites back into the offspring, sort them and take top *popSize* individuals.
The ANN is mainly trained with crossover and mutation. Proper weights for each node will gather together with each other by crossover. And some new good weights are introduced by

[2] Montana, D. J., & Davis, L. (1989, August). Training Feedforward Neural Networks Using Genetic Algorithms. In *IJCAI* (Vol. 89, pp. 762-767).
[3] Montana, D. J., & Davis, L. (1989, August). Training Feedforward Neural Networks Using Genetic Algorithms. In *IJCAI* (Vol. 89, pp. 762-767).

mutation/initialization. Good genes will survive by elite preservation and bad genes will be weeded out in each generation.

ANN Generalization

To generalize the ANN, I firstly tried dropout, which is to hide some neurons with a probability *1-p* in the forward process. However, the accuracy by doing so is quite lower than not, and the ANN seems not make progress in each training iteration. The reason is that the dropout is useful in SGD to prevent it gets stuck into a local optimal as it's a hill-climbing process. But in our training method, good entries are mainly obtained by survival and mutation. The dropout thus does not help here.

Then I tried to use small batches data for training iterations. In each iteration, the data batch is randomly chosen from the training set. And lastly, we use testing data to evaluate our ANN. It can help ANN not overfitting as in each round, the training data is different, and the ANN thus cannot overfit a specific training data. We did experiments to show it is helpful to generalize the ANN. We calculated the mean accuracy of 10 experiments for each batch size. And here's the result:

Batch size	**Average Accuracy on Training Data**	**Average Accuracy on Testing Data**
10	64.27%	64.19%
50	65.05%	65.57%
100	77.93%	75.80%
200	78.71%	75.43%
600(The whole training dataset)	78.83%	69.21%

We can find that using a moderate batch size like 100, 200, the ANN can be generalized well. And using small data batches to train can also save time as there's less computation. But if the batch size is too small, the performance of the ANN will be poor.

Comparison with Other Methods

We also tried the random forest classifier and artificial neural network trained with gradient-descent-based backpropagation and compared them with ANN trained with GA.

The random forest classifier with 200 decisions trees can finish training and predict the result within 30 ms while our classifier normally uses tens of seconds to train. From the perspective of time efficiency, the random forest classifier does much better. And the accuracy of random forest classifier is about 78% which is similar with that of the ANN. But ANN with appropriate hyperparameters can achieve about 82% which does make a difference.

The ANN trained with gradient-descent-based backpropagation can achieve the accuracy about 77% as well and it also takes less time to train. However, the main defect of this method is that it's difficult to choose hyperparameters like learning rate. I cannot find a proper learning rate when there're more than two hidden layers, the accuracy or the loss changes little even I used decay learning rate. With an improper learning rate, the performance of the ANN can be extremely poor, I sometimes even got an accuracy around 35%. But to train ANN with GA can relieve this problem. Though the selection of the number of hidden layers, the number of neurons in each layer can influence the performance, they will not affect too much on the performance. With inappropriate hyperparameters, the accuracy is around 70% while with appropriate one, it can achieve greater than 80%.

And we also control the structure of ANN to avoid a complicated structure to prevent overfitting.

Hyperparameters Tuning

Number of Generations

We found that 1000 generations would be sufficient to train a good model and 500 generations will be normally enough to obtain a good model.

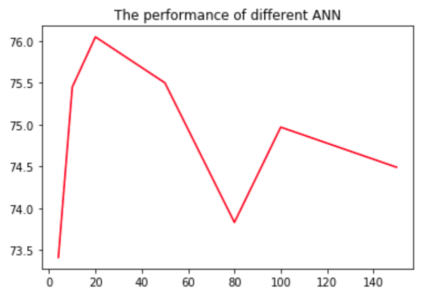
Population Size

We found that a population size greater than 50 will be enough to train. Even with a population size 20, it can generate a not bad model.

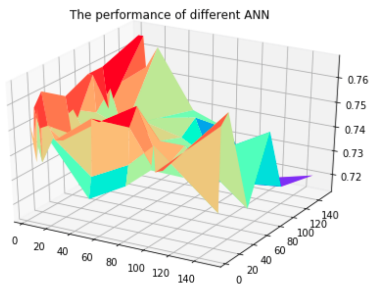
The Number of Hidden Layers and the Number of Neurons in Each Layer

We first tried one hidden layer neural networks with 4~150 neurons and we found that the performance of the ANN first increases and then decreases with the increasing number of neurons. As a complicated structure may cause overfitting and thus get bad performance on unseen data.

The number of neurons	Performance
4	73.41%
10	75.45%
20	76.05%
50	75.5%
80	73.83%
100	74.97%
150	74.49%



We also investigated two hidden layers with neurons in each one varies from 4 to 150. And we plotted the performance of them:



We can observe that the ANN with a complicated structure will not perform well and the best performance ANN has 20 neurons in the 1st hidden layer and 150 neurons in the 2nd hidden layer.

Activation Function

We provide activation functions *sigmoid*, *tanh*, *Gaussian* and *ReLU*. After experiments, we found that *tanh* and *Gaussian* do not perform well on this task (about 3% lower accuarcy). And the classification performance of sigmoid is similar to that of *ReLU*, however, *ReLU* is much faster than *sigmoid*.

Conclusion

ANN trained with GA has its own advantages like less effort to tune some hyperparameters and its own weakness like higher time cost to train compared to other classification algorithms. It can achieve a good result with proper implementations.