CSC3423 Bio-Computing
Coursework 2: Biologically-Inspired Computing for Machine Learning

B9052919 Ryota Fuwa
Word Count (1997)

1 Neural Networks

I used python to implement neural network algorithms since I thought it would be easier to change operations and parameters.

1.1 Description

Multi-Layer Perceptron is one type of neural networks. Neural Network is optimization algorithm inspired by how neurons in a human brain are connected with each other. It has an input layer, which takes data, and pass it to hidden layers, after that it comes out from output layer. In the calculation, it usually goes through matrix multiplications and non-linear activation functions. I present an abstract sketch in Figure1.

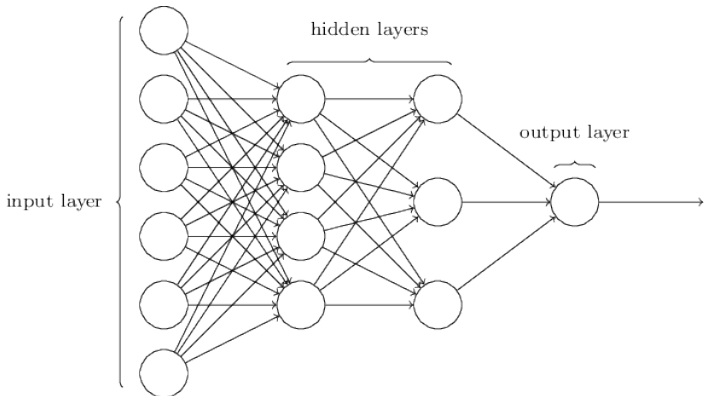

Figure1 Overview of MLP

In supervised learning, networks are first trained with prepared data. By comparing output of the network with provided label data and calculating a fitness value, weights in the network can be optimized to reduce the fitness value so that the output of the network can be matched or fairy close.

1.2 Reason

Due to the fact that MLP has hidden layers and non-linear activation functions, it can be used also in complex non-linear classification problems unlike linear classifications. Moreover, if I have access to GPU, matrix multiplications can be done quickly enough even if I have a relatively large network.

1.3 Tuning Process and Performance

As this case study is a binary classification problem with 2-dimensional input (x, y), some part of the structure of this network can be fixed from the beginning. Obviously, input layer has two inputs, and some number of hidden layers. At the end, it has two outputs, and each represent the possibility of the class, so it means the sum of the two outputs will always be 1.0. To do this, I use one-hot encoding for label data and Softmax activation function at the output layer. In addition, I employ binary cross entropy for the fitness function unless I specifically mention other fitness functions. I listed all fixed details for the network in Table1. Red Rows are the configurations that I'll change to improve the network later on, and the number of outputs will always be 2 because of one-hot encoding.

Table1 Pre-Defined Configuration of the Network

Number of Inputs	2
Number of Outputs	2
Activation Function of Output Layer	Softmax
Fitness Function	Binary Cross Entropy
Activation Function	Relu
Optimizer	SGD
Batch size	256

First, I tried to figure out the number of hidden layers and weights inside each layer to get a fairy high accuracy on test data. It's easier to get good accuracy with bigger networks in general, but it increases the required time for training and classification. I tried with different configurations showed in Table2, and I plotted fitness values over epochs for each in Figure2.

Table2 Statistics for Different Layers & Weights Network

weights	5	10	20	5-5	10-10	20-20
Fitness Value	0.3305	0.3320	0.3334	0.3263	0.1631	0.0635
Accuracy (%)	0.84211	0.82631	0.84211	82.105	89.474	96.316
Execution Time (In Training) (sec)	14.658	13.738	14.574	14.781	14.883	18.320
Classification Rate (%)	100.0	100.0	100.0	100.0	100.0	100.0

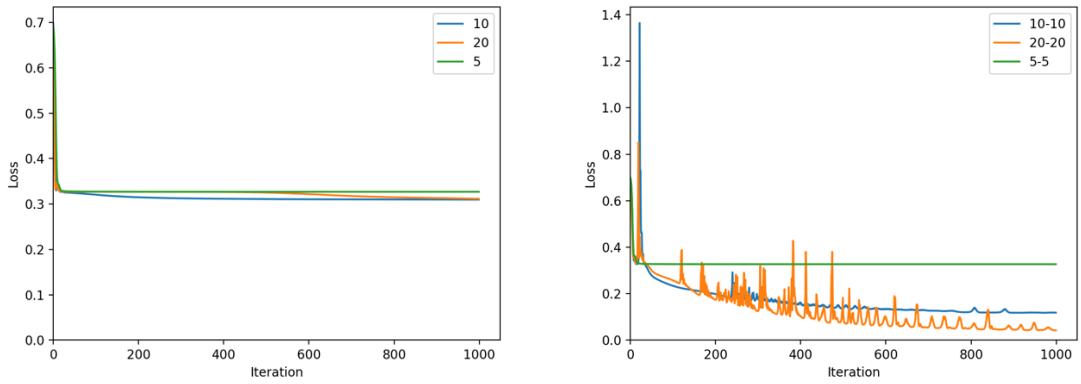

Figure2 Numbers of Layers and Weights(Left: one hidden layer, Right: two hidden layer)

Figure2 suggests that 1-hidden-layer model don't produce good result. If I tune other parameters and use different optimizers, it might also work. However, I'll use 2-hidden-layer model with 20-20 weights from now on even though it fluctuates in this figure.

Second, I tried with three different activation functions. I plotted each performance in Figure2.

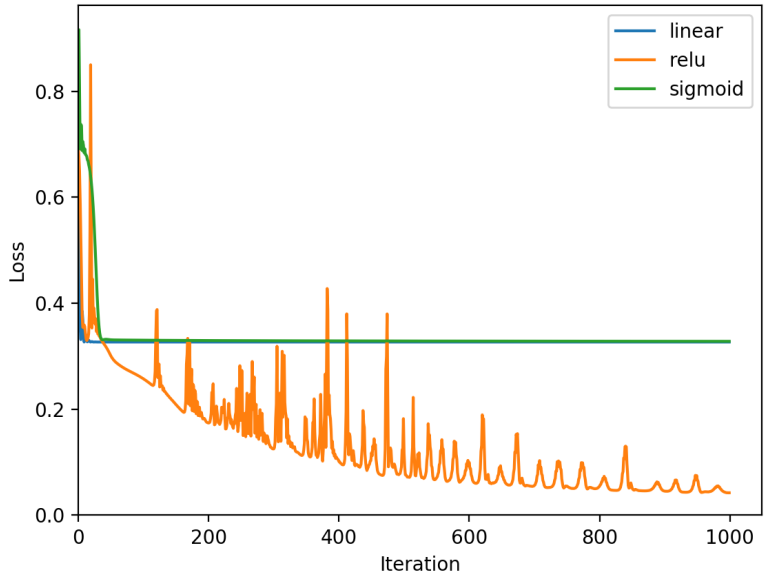

Figure3 Use of Different Activation Functions

It makes sense that the network using linear activation function didn't improve since all networks that use linear activation function only can classify data linearly; however, I don't know why sigmoid also didn't improve itself much. That might be because I haven't modified all other parameters related to the network such as the numbers of layers and weights.

Next, I focused on optimizers. I measured performance with two types of optimizers, Stochastic Gradient Decent and Adaptive Moment Estimation. When it comes to SGD, if learning rate λ is less than 1.0, it stagnates around accuracies of 83 %. I depicted two classification graphs in Figure4. The right one is for 83% accuracy, and the left one is for 96.842%, which I've got by using Adam. Red dots mean that the network classified wrong for the samples.

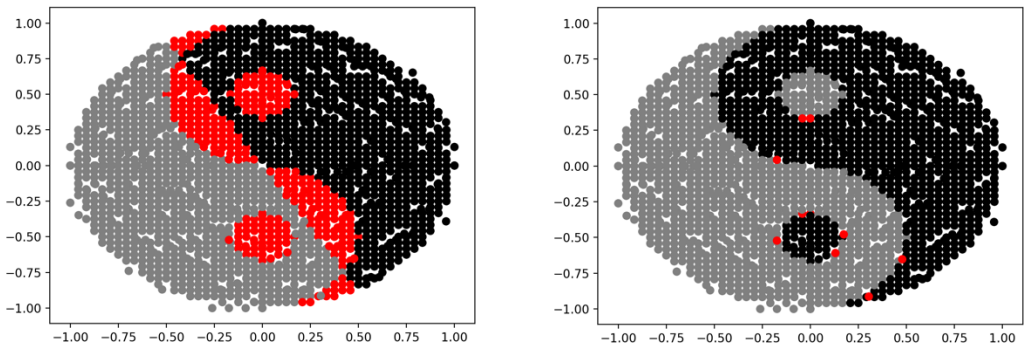

Figure4 Classification (L: Not Good Result with Sigmoid, R: Great Result with Adam)

I figured that the learning rate λ was small enough that the network can't search the optimal value to get it down. Thus, I made λ bigger, and also had it decay while training.

Adam embeds momentum and λ decay into its expression whereas SGD just takes derivative and decent repeatedly unless designers specify momentum, hyper parameters and etc. I listed statistics for each optimizer in Table3.

Table3 Execution with Different Optimizers

Optimizer	SGD	Adam
Accuracy (%)	91.578	0.98421

.

Finally, I tried with different batch size to make the execution faster. I plotted execution times for different batch sizes in Figure5.

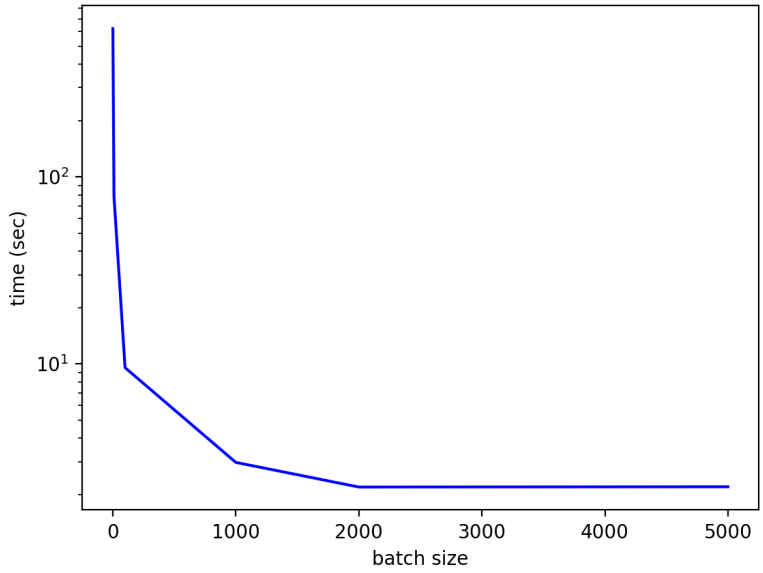

Figure5 Batch size vs Execution Time

As we can see from Figure5, the bigger the batch size is, the shorter execution time will be in general: However, it doesn't change from a certain size depending on the hardware. In my case, it was around 2000. Most part of the calculation is matrix multiplication, so if it is executed on GPU, the execution time are expected to be shorter.

I present the configuration and statistics of the network which produced the best result in Table4.

Table4 Best Performance and The Configuration

Network Structure	20 - 20
Activation Function	Relu
Optimizer	Adam (λ=0.1, λ decay=0.002, β1=0.75, β2=0.752)
Epochs	500

Batch Size	2000
Accuracy (%)	98.421
Execution Time (sec)	3.7297

I also did an experiment to make accuracy as small as possible with a big network. However, I couldn't manage to get 100 % accuracy on test data. After iterating in some amounts of epoch, accuracy for training data gets 100 %, and fitness value becomes as low as 1.0e-5; However, accuracy for test data never reached 100 % in my experiments, instead I only could get the accuracy of 98.421% on test data set, which was even not stable. I confirmed there wasn't overfitting by randomly allocating 20 % of the training data for evaluation. I showed moves of the training accuracy and the evaluation accuracy over epochs in Figure6.

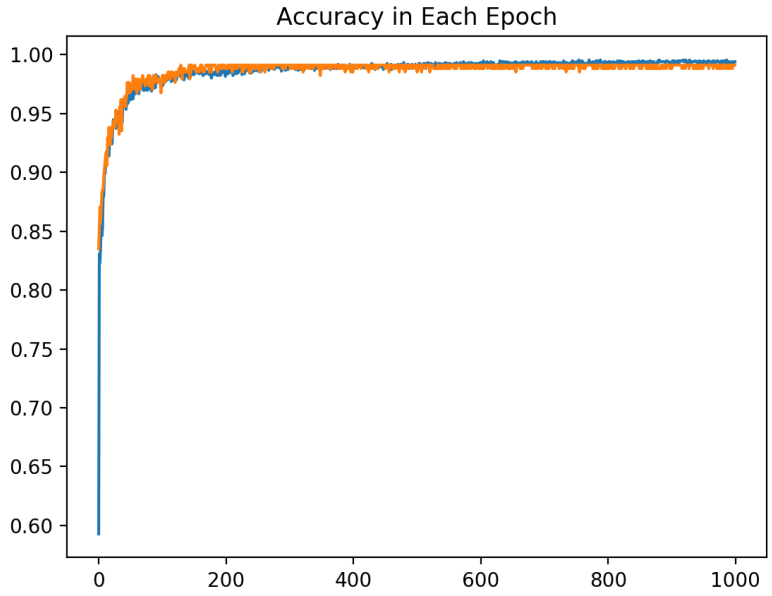

Figure6 Accuracy on Training and Evaluation Data

Figure6 demonstrates that accuracy on evaluation data doesn't get lower over iteration, which means there is no overfitting.

2 Genetic Programming or Genetic Algorithm (800 words)

I used Java codes provided for this coursework to implement this algorithm. Performance metrics is accuracy, classification rate, and execution time.

2.1 Description

Genetic Programming is similar to Genetic Algorithm in that it's inspired by natural selection. However, it uses a tree structure, which has operations at the nodes and values at the leaves as individual (showed in Figure7) rather than using chromosome, which is just array or matrix of values imitating DNA. The depth of the tree and types of operations allowed in trees are defined by the designer of GP program, but all the rest

such as values and operations in each node will be determined through the optimization procedure. The optimization process of GP also comes with selection, crossover, mutation just like GA. All stages work pretty much the same way as in GA. Selector chooses individuals by referring to fitness value for each, crossover mixes individuals together to create new one, and mutator changes some part of an individual also to make changes. Then, the program iterate the cycle over and over until it get an individual which can produce a desirable result.

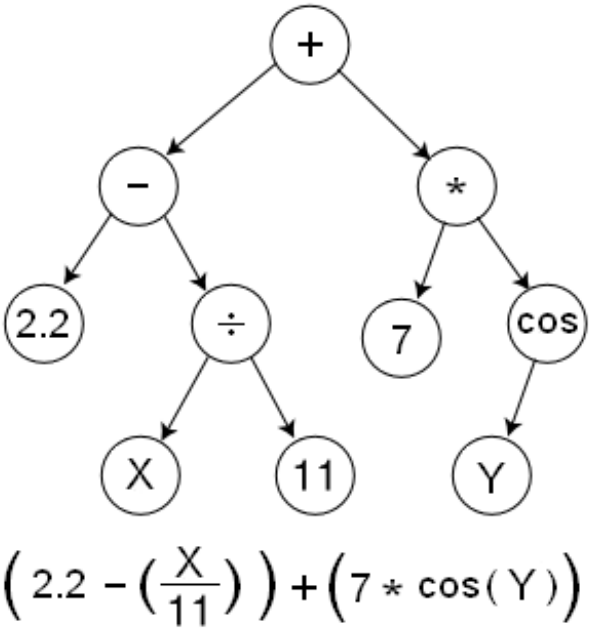

Figure7 Tree Structure In GP

2.2 Reason

GP uses only one classifier for this coursework just like neural network classifier, so I thought it would be much fair to compare their results rather than using algorithms that has multiple classifiers. Moreover, GP is often used for classification problems [1], and also considering that this coursework problem is just a simple binary classification, I thought GP could produce a good result. Moreover, I haven't tried to use only this algorithm out of our choices (GA, PSO, NN, GP), so it seems to be a good practice to choose this one.

2.3 Tuning Process and Performance Report

First, I run the code with all operations to see if which operations affect the performance of this algorithm in this classification problem. After I ran it with 500 generation 20 times, which took more than one hour, I found the most likely used operations by calculating the rate of the use of each operation (shown in Figure8); In addition to the 4 basic operations(add, subtract, multiply and divide), Cube, Inv and Pow operations are used quite a lot whereas all other operations didn't appear in the tree much. Thus, I decided to use 4 basic operations and cube, inv, and pow operations based on Figure8 and my observation that that the program gets really slow if GP has GREATER_THAN and SMALLER_THAN operations.

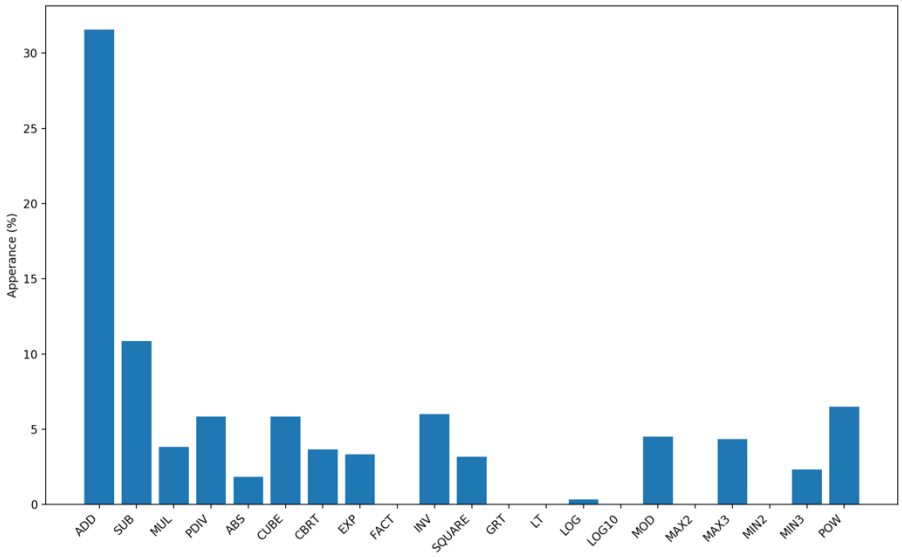

Figure8 Rate of Each Operation In GP

Next, I changed the fitness function to get accuracy on test data higher. I tried with different functions using sigmoid, power, and so on. What I aimed to do here was to make fitness value get much higher as error rate goes up. With default fitness value, it just uses error number as it is linearly, which I thought was not efficient. However, all I tried didn't change the accuracy much, so I just decided to use this fitness function (Math.pow(errorNum, 3) instead, and I plotted fitness values over Generation in Figure9 to see if the program progresses.

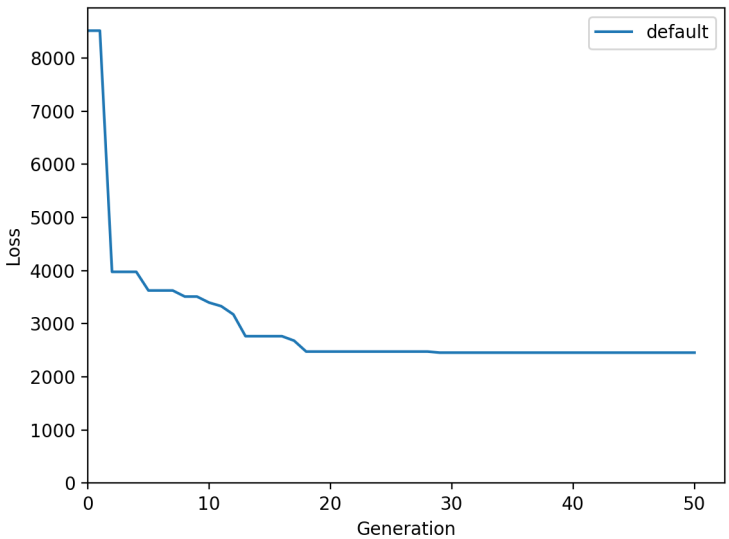

Figure9 Fitness Over Generations with Default Config.

After that, I changed Initialiser first. Instead of FullInitialiser, I used RampedHalfAndHalfInitialiser because if I use FullInitialiser and set the max depth of trees to high value, I figured that it wouldn't search the potential solutions with low depth trees

enough. The best individual doesn't have to be the deepest one. I left all others as they were (, which were SubTree operation) since I'd like it to be more explorative based on what I could see from Figure9. Moreover, I chose random generator over fixed constants because of the same reason.

Then finally, I modified parameters to optimize the algorithm. To make the algorithm more explorative, I set crossover and mutation probabilities to 0.3 and 0.25 respectively. Also, I set the number of elite trees and tournament size to lower values. I listed parameters I changed and the results for them in Table5.

Table Comparison of Default and The Best Configuration

Configuration	Default	The Best
Number of Elites	10	5
Tournament Size	10	2
Crossover	0.2	0.3
Mutator	0.2	0.25
Number of Constants	0	2
Accuracy on Test Data (%)	87.89	92.11
Execution Time (sec)	2.997	4.773

Execution Time differs in every execution. Sometimes it becomes much faster than that in Table5, but generally speaking, the one with default parameters was faster than the other. When it comes to accuracy on test data, I couldn't make it higher than 94% even though I tried many times also with a large population size and long generation (500, 500 respectively). I think the reason for this is the classifier fails at classifying the instances in both circles inside the other classes (see Figure4). And the shape of the best trees is also different every time. It sometimes has less than 6 depth.

3 Performance Comparisons

I show you the performance result from the two algorithms in Table6 again.

Table6 Performance Review for Neural Network and Genetic Programming

Algorithm	Neural Network	Genetic Programming
Accuracy (%)	98.421	92.11
Execution Time (sec)	3.7297	4.773

According to Table6, it goes without saying that Neural Network is better for both accuracy and execution time. And also, Neural Network was much more stable in training. When I trained a network with the exact same configuration, the performance of the network was basically the same whereas Genetic Programming create totally different trees that produces different accuracies on test data. Even if I used the same configuration

from crossover, selector and mutator to all parameters, the result tree sometimes only can classify 88% of test data correctly.

As for the execution time, it makes sense that training for neural network is much faster than the other because it is calculated as matrices operations and I used tensorflow, which is library written in C++ and specialized in deep learning operations, for instance using back propagations for updating each weights, so it works pretty quickly even on CPU. On the other hand, GP has to deal with tree structures for each individual in every generation. Thus, making trees longer even by a little heavily affects the execution time of GP. Furthermore, I think neural network performed better than GP in this classification problem since the neural network for this coursework uses more than 1000 weights interconnected with each other while GP trees can't have more than 1000 weights practically at least on my computer because it makes execution unbearably slow and more than 7^{th} depth trees will immediately be banished by only one variable trees like y = var1.

Both of the two algorithm has 100 % classification rate, which is sometimes not desirable. 100 % classification implies that the classifiers try to classify instances even if there are not sure, so they tend to have a higher error rate.

Reference

[1] Neuralnetworksanddeeplearning.com. (2019). [online] Available at: http://neuralnetworksanddeeplearning.com/images/tikz11.png [Accessed 8 Dec. 2019].

[2] Upload.wikimedia.org. (2019). [online] Available at: https://upload.wikimedia.org/wikipedia/commons/7/77/Genetic_Program_Tree.png [Accessed 8 Dec. 2019].

[3] Keras.io. (2019). *Home - Keras Documentation*. [online] Available at: https://keras.io/ [Accessed 8 Dec. 2019].

[4] En.wikipedia.org. (2019). *Sigmoid function*. [online] Available at: https://en.wikipedia.org/wiki/Sigmoid_function [Accessed 8 Dec. 2019].

[5] En.wikipedia.org. (2019). *Genetic programming*. [online] Available at: https://en.wikipedia.org/wiki/Genetic_programming [Accessed 8 Dec. 2019].

[6] Epochx.org. (2019). *EpochX: genetic programming software for Java*. [online] Available at: https://www.epochx.org/ [Accessed 8 Dec. 2019].