CSCI447 - Assignment 4 - Design Document

Chris Major CHRISMICHELMAJOR@HOTMAIL.COM
Farshina Nazrul-Shimim FARSHINA287LEO@GMAIL.COM
Tysen Radovich RADOVICH.TYSEN@GMAIL.COM
Allen Simpson ALLEN.SIMPSON@MOTIONENCODING.COM

Editor: (none)

1. Overview

The purpose of this document is to detail the design of three evolutionary algorithms to train a feedforward neural network. A genetic algorithm, differential evolution, and particle swarm optimization are to be implemented and compared to backpropogation as a means of training. The neural network will perform classification or regression on a series of six data sets, provided by the UCI Machine Learning repository (Dua and Graff (2017)). Experiments focused on the performance of the various training methods, with regards to convergence conditions, are performed.

2. Description

The assignment will be coded in F#, building on the previously constructed feedforward neural network and backpropogation features.

2.1 Experimental Design

The experiment will consist of four training trials for the feedforward neural network. The first will use backpropogation, serving as a control experiment and using existing code from the previous assignment. The second will use a genetic algorithm with real-valued chromosomes for training, the third will use differential evolution, and the fourth will use particle swarm optimization (PSO). Each trial will consist of three tests with 0, 1, and 2 hidden layers within the feedforward neural network.

Validation of the classification and regression conducted by the neural networks will be conducted via Mean Square Error (MSE) for all output nodes of the network. Determination of a correctly classified or regressed value will depend on the comparison between an observation's given class/regression value and the network-determined class/regression value. The results of these comparisons will be collected over various trials as defined by 10-fold cross-validation, then averaged to determine the algorithm's performance. Hypotheses regarding the terms of convergence, in the context of acceptable network performance, will be tested and analyzed in the official project document.

2.2 Major Decisions

2.2.1 Feedforward Neural Network

The feedforward neural network to be used is the same network developed for Assignment #3, based on Kulkarni and Harman (2011). Our design intends to incorporate Single Instruction, Multiple Data (SIMD) features to reduce computation times, especially in regards to particle swarm optimization.

2.2.2 Backpropogation

The backpropogation functionality to be used is the same as was developed for Assignment #3, based on Rumelhart et al. (1985). Our activation function will be a Padé approximation of the logistic function, to aid in quick calculation. There is a threshold where values fall off, which will be attenuated accordingly. This will allow for the use of SIMD to accelerate the computation.

2.2.3 Genetic Algorithm

A genetic algorithm will be implemented as described by Booker et al. (1989), in order to train the feedforward neural network weights. A population of neural networks is taken from the connection matrix and set as a population. Our implementation intends to generate the next generation of networks through a multi-level process. 8% of a following generation will be composed of the previous generation's most fit members selected via elitism, 50% will be selected tournament-style, and 40% will be randomly selected to allow for a stochastic element. The remaining 2% will consist of randomly generated "immigrant" members, to ensure some degree of diversity as generations evolve. the populations selected by the first three methods will be selected without replacement so that we do not have the top 8% of members represented too often. Ideally, these methods will ensure the improvement of the generations of networks, while allowing a diverse subset of the population to continue growth. In the case of unexpected performance, these percentages will be tuned accordingly.

2.2.4 Differential Evolution

Differential evolution will will also be implemented as described by Storn and Price (1995), in order to train the feedforward neural network weights. We will be using *DE\Best \1 \Uniform*. Similar in nature to the genetic algorithm, this method will apply uniform crossover, but we will be crossing every member with the best member. This should yield performance similar to Particle Swarm Optimization. One difference vector will be used for simplicity; more may be considered if improved performance is demonstrated. Finally, our crossover method will be uniform crossover - if n-points demonstrates improved performance, again, then further consideration will be given to the optimum crossover method.

2.2.5 Particle Swarm Optimization

Particle Swarm Optimization (PSO) will also be implemented as described by Poli et al. (2007), for the purposes of training the feedforward neural network weights. The weights of our neural network will be represented either as "island" lBest clusters of particles, or as "star" gBest formations allowing complete communication between all particles; this choice will be determined based on performance restrictions that we may find during testing. PSO will be run for a set amount of time to allow for the particles to move towards their optimum values, with tuning parameters to place weight upon personally-observed or group-observed bests.

2.3 Tuning

The parameters that require tuning in this design relate to the populations in play with these algorithms. First, the population size will be tuned to balance the accuracy of larger generations with the speed and performance required by our computers. The mutation rate in population tuning and learning rate in back-propagation will be adjusted to ensure sufficient advancement in learning performance. Variables c_1 and c_2 are used to assign greater or less weight to the pBest and lBest factors of the particle swarm optimization. Additionally, the inertia coefficient ω will likely be left low so that particles will be able to make quicker adjustments to the local swarm's best known location.

Overall performance will depend on the accepted error set at the start of each algorithm run, a factor that will be tuned across the experiment. A cutoff value will determine how many runs with the same value we will accept before we argue that the algorithm has plateaued. Convergence will be calculated such that, if an algorithm's best member reaches a low enough error or no little-to-no diversity remains, the algorithm will count as "converged", a process we hope will make the algorithms finish within reasonable times.

3. UML Diagram

The UML diagram for this assignment is included below in Figure 1. Note that not all of the classes are connected, as F# is a functional programming language each algorithm is implemented as a function that will take a Network and apply its "run" command and then take said network and it's output value, and tune the weights on the network until convergance. as each algorithm is a function and not a class, they are not represented on the UML class diagram.

DataSetMetadata serve as metadata for the dataset we are testing tracking our attributes' real attributes, categorical attributes, input nodes, output nodes, as well as giving functions to return a class, filling the expected output to track error of an algorithm, and checking if a dataset is a classification or regression problem. The input layer for our neural network is stored as a Point from our dataset. This point will be passed to our algorithm's feedforward function to set the networks input layer.

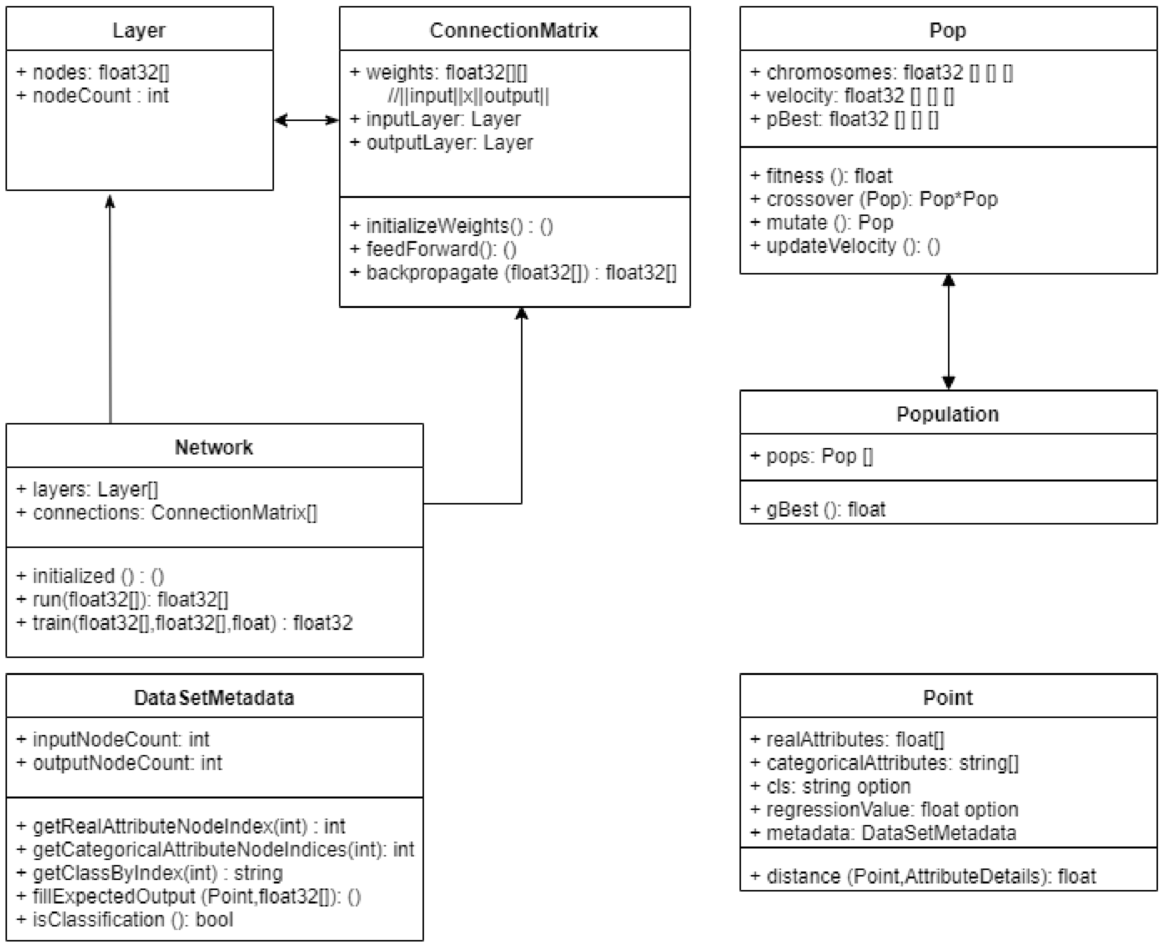
Layer
+ nodes: float32[]
+ nodeCount : int
ConnectionMatrix
+ weights: float32[][]
//||input||x||output||
+ inputLayer: Layer
+ outputLayer: Layer
+ initializeWeights() : ()
+ feedForward(): ()
+ backpropagate (float32[]) : float32[]
Pop
+ chromosomes: float32 [] [] []
+ velocity: float32 [] [] []
+ pBest: float32 [] [] []
+ fitness (): float
+ crossover (Pop): Pop*Pop
+ mutate (): Pop
+ updateVelocity (): ()
Population
+ pops: Pop []
+ gBest (): float
Network
+ layers: Layer[]
+ connections: ConnectionMatrix[]
+ initialized () : ()
+ run(float32[]): float32[]
+ train(float32[],float32[],float) : float32
DataSetMetadata
+ inputNodeCount: int
+ outputNodeCount: int
+ getRealAttributeNodeIndex(int) : int
+ getCategoricalAttributeNodeIndices(int): int
+ getClassByIndex(int) : string
+ fillExpectedOutput (Point,float32[]): ()
+ isClassification (): bool
Point
+ realAttributes: float[]
+ categoricalAttributes: string[]
+ cls: string option
+ regressionValue: float option
+ metadata: DataSetMetadata
+ distance (Point,AttributeDetails): float

Figure 1: UML Diagram for Assignment #3

4. Class Description

4.0.1 Network

The Network class represents the feedforward neural network, containing an array of layers and a matrix representing the connections. initialize() randomizes the weights of the matrix. run() takes an input array and returns the resulting output array. train() takes the input array, the expected output array, and the epsilon (accepted change in error) as parameters and runs the feedforward neural network until convergence, then returns the error.

4.0.2 Layer

The Layer class represents a layer - input, output, or hidden - within a Network. It has an array of nodes, represented by floats, and a nodeCount value representing the number of nodes in that layer.

4.0.3 Connection Matrix

The ConnectionMatrix class represents the connections within the Network. A 2D array of weights, represented by floats, an inputLayer, and an outputLayer make up this class. initializeWeights() randomly assigns initial weights to the ConnectionMatrix. feedForward() takes the input layer and cross-multiplies its nodes with the weights. backpropogate() is the original backpropogation functionality from the previous assignment and simply performs backpropogation by taking the delta values for the outputLayer and using them to modify the weights as need be.

4.0.4 Population

The Population class represents the general population of an evolutionary or population-based algorithm. It is made up of an array of Pops, with a function returning the best of the population through gBest().

4.0.5 Pop

The Pop class represents an individual within the class. It has a 3D array of chromosomes which represent an array of weights used in each algorithm to tune the weights of a Network, a 3D array of velocities to represent each gene in our chromosomes, and a 3D array of pBest "personal best" values. The fitness() function builds a neural network with weights associated with the chromosomes and runs the neural network forward, returning the accuracy of the answer. The crossover() function takes one Pop and crosses it over with another Pop, then returns the children. mutate() applies mutation to the chromosomes of the current Pop. updateVelocity() updates the velocity of the current Pop.

4.0.6 AttributeDetails and Point

AttributeDetails is a structure used to store the calculations required to find the distance between two points. A Point is an object that stores the real attributes, categorical attributes, and the class/regression value of each point fed into the network.

5. Summary

Per these design guidelines, we aim to deliver three evolutionary algorithms and an analysis of their effects on the training of a feedforward neural network. Upon implementation and tuning, experiments will be conducted on provided data sets to determine metrics of performance and convergence. A formal report detailing the results and analysis will follow upon completion of the project.

References

L.B. Booker, D.E. Goldberg, and J.H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40(1):235 – 282, 1989. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(89)90050-7. URL `http://www.sciencedirect.com/science/article/pii/0004370289900507`.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

Sanjeev Kulkarni and Gilbert Harman. An elementary introduction to statistical learning theory. 04 2011. doi: 10.1002/9781118023471.

Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization: An overview. *Swarm Intelligence*, 1, 10 2007. doi: 10.1007/s11721-007-0002-0.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Rainer Storn and Kenneth Price. Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces. *Journal of Global Optimization*, 23, 01 1995.