Journal of Machine Learning Research Format (2019) 7 Submitted 10/11; Published 10/11

CSCI 447: Machine Learning – Project 4

Peter Ottsen PETER.K.OTTSEN@GMAIL.COM
Bruce Clark BRUCEWESTONMT@GMAIL.COM
Forest Edwards FORESTJEDWARDS@GMAIL.COM
Justin McGowen MCGOWEN.JUSTIN.P@GMAIL.COM

Editor: None

Abstract

For this project, we were tasked with implementing three different evolutionary algorithms, the Genetic Algorithm (Whitley, 1994), Differential Evoultion (V. L. Huang, 2009), and Particle Swarm Optimization (Bai, 2010). We used these to train a Multilayered Perceptron (D. Montana, 1989). These algorithms performed classification and regression on the following data sets : the Abalone data set (Paulo Cortez, 2009), Car data set (Bohanec, 1997), Forest Fires data set (Cortez, 2007), Machine data set (Phillip Ein-Dor), image data set (Vision Group, 1990), Wine data set (Paulo Cortez, 2009). Our design used a population manager class to handle the three algorithms, as well as multiple instances of a MLP. Each of our evolutionary algorithms are designed in separate classes inheriting from PopulationManager which control our popuation of MLPs. Similarly to project 3, we ran multiple tests on MLPs with different numbers of hidden layers to see if any one gives us a better result than the others. We will use the F-score and Mean Squared Error loss functions to assess the performance of each algorithm. Based on the fitness functions we picked for the different algorithms, we see that they are able to preform regression with a good measure, while not being able to preform discrete classification very well.

1. Problem Statement

Some computational problems in this world appear to be too complex to solve using conventional methods. Many may think that in order to solve a complex problem, a complex solution is needed, but this isn't always the case. Evolutionary algorithms are a rather simple way to find optimal solutions to a wide range of problems. They are based on the concept of natural selection. Organisms who are more suited to a given environment thrive and reproduce at a higher rate, while those who have less desirable traits die off and reproduce at a lower rate. The algorithms implemented, Genetic Algorithm (GA), Differential Evolution (DE), and Particle Swarm Optimization (PSO), carry out this concept but in different ways. A population of candidate solutions is tested on the problem and evaluated for fitness (accuracy/error). The more fit or accurate individuals are chosen to as the basis for the next iteration of the algorithm. One major advantage to using this method is that it performs better overtime. Each subsequent generation has an overall fitness greater than the last, meaning it is better at making correct decisions.

For this project, evolutionary algorithms will be used to solve multi-layer perceptron neural networks with 0, 1, and 2 hidden layers. Based on our findings from project 3, we expect to see a similar pattern of results depending on the number of hidden layers in the

MLPs. We hypothesize that a MLP that has one or two hidden layers should have better performance than one with no hidden layers.

2. Description of Algorithms implemented

2.1 The Genetic Algorithm

The Genetic Algorithm is one of the more straightforward evolutionary algorithms. It is an iterative process that improves with each generation by combining characteristics of the highest performing individuals (D. Montana, 1989). The algorithm first takes a population of members, and evaluates the fitness of each member. In our case, each member is represented by a MLP, with attributes as the inputs and the classification/regression guesses as the outputs. Fitness for each member is then evaluated on the accuracy of that guess. The GA then selects the fittest individuals in the current generation and performs crossover and mutation to breed genes, producing more optimal members for the next generation. Selection is done by prioritizing better fitnesses, but is also semi-random to ensure population diversity. This process is then repeated multiple times, and the MLPs continue to be replaced and improve, making better predictions over subsequent generations.

2.2 Differential Evolution

The idea for Differential Evolution comes from the idea that we can find an optimum solution to a problem by iteratively replacing members of the population that preform poorly with better members. If we perform this operation enough times, we should expect to have fitter members converging at an optimum to the problem, although this is not guaranteed. To find better members to replace the worse ones with, we have many options. We went with a classic approach, where for every member in the population, 3 unique members were chosen at random. We take a combination of these members: the first member is added to the difference of 2 and 3, multiplied by a scalar D (differential weight usually between 0-2) which is a tunable parameter (Storn, 1996). We then randomly select the genes of this compound member or the original member of the population to generate a mutated member. If this mutated member preforms better than the original, it replaces the original.

The method above shows population selection, mutation, and crossing. Because we randomize the genes of our population to begin with, our population is quite diverse, so we can explore and exploit the space given the above method, hopefully well enough to find a good solution to our problem.

2.3 Particle Swarm Optimization

Particle swarm optimization uses cognitive and social components to find an optimal solution. The cognitive component comes from the candidate solution being updated based on the best solution that the candidate has seen across the iterations, while the social component comes from the candidate solution also being updated based on the best solution that the entire candidate population has seen. The candidate's best known solution is the known as the particle best, or pBest, and the population's best solution is known as the global best, or gBest. After each iteration the fitness of each individual is checked to determine if the candidates reached a new pBest and if any of them reached a new gBest. The amount

that each candidate is updated after each iteration is called its velocity. The velocity, $V(t)$ is calculated by the below equation:

$$V(t) = \omega * V(t-1) + c_1 * r_1 * (pBest - X(t)) + c_2 * r_2 * (gBest - X(t)) \quad (1)$$

where ω is a tunable inertia coefficient, $V(t-1)$ is the velocity of the previous iteration, c_1 and c_2 are tunable coefficients, r_1 and r_2 are random number between 0 and 1, pBest and gBest are the individual and global best respectively, and $X(t)$ is the current position of the candidate. The position of the candidate is then updated with this velocity by the equation below:

$$X(t+1) = X(t) + V(t) \quad (2)$$

where $X(t+1)$ is the updated position, $X(t)$ is the current position, and $V(t)$ is the velocity calculated from the above equation. This update process is carried out each iteration until the optimal solution is found or an iteration limit is reached.(Bai, 2010)

3. Experimental Approach

The evolutionary algorithms, Particle Swarm Optimization, Differential Evolution and Genetic Algorithm, along with Backpropogation, were tested 0, 1, and 2 hidden layers. We did not go over 2 hidden layers as we would potentially jump into the world of deep learning and would have to deal with the vanishing gradient problem. (**?**)

Each multi-layer perceptron was implemented with a set number of hidden nodes in each hidden layer based on the data set (Abalone - 30, Car - 10, Segmentation- 14, Machine - 15, Forest Fires - 20, and Wine - 18). Each genetic algorithm was run with a population of 50 and an generation/iteration limit of 1000 so that we would be able to directly compare performance.

The mean squared error was used to evaluate the fitness of each candidate solution of the population. Mean squared error is given by the equation below:

$$MSE = \frac{\sum_{j=1}^{n}(ActualResult_j - PredictedResult_j)^2}{n}$$

Mean-squared-error gives us a positive value that represents the average squared value between the actual and predicted value. Mean-squared-error penalizes more heavily for occasionally guessing values that are further away from the average than consistently guessing values that only vary slightly from the actual value.

In implementation, we used slight tweaks of this formula in order to try to incentivize finding an optimum to the problem correctly.

Each of the algorithms have multiple parameters to tune. Tuning these parameters is incredibly important in order to get the best performance from our network. We had limited time to do extensive testing, but were able to test a few different values for each algorithm.

For the GA there were a few parameters that we were interested in tuning: the replacement rate, the mutation rate, and the mutation value (how much we change a gene by when it is selected for mutation). We changed the mutation rate and mutation value multiple times and watched how our our MLP progressed from generation to generation.

We saw that when these values were too high (around 0.1) the fitness measurements would jump all over the place but, in general, our MLP was slower to progress. Once we changed these rates down to about 0.001, we saw a slower but more positive increase in the performance of our MLP. The replacement rate (the amount of population that is replaced in each generation) we have at 0.5. For our selection we chose to use Roulette Wheel Selection (Adam Lipowskia, 2012). This selection method randomly selects individuals within our population and has a higher probability of selection the most fit individuals. We then use these selected individuals to breed, mutate the offspring, and then use the mutated offspring replace the weakest members of the population. We performed the above for 1000 generations with 0, 1, and 2 hidden layers over all the data sets.

To tune the Differential Evolution algorithm, we have two main parameters to optimize. We have our mutation rate, which is a probability of how often a mutant child will pick the original parent's gene or the generated parent's gene that comes from the 3 randomly selected members. We can also tune the differential weight which affects how much the generated parent is influenced by the second and third randomly selected members. We will mainly be testing values between 0 and 2 to ensure that enough, but not too much exploration is happening when mutation occurs. For mutation rate, we ended up using 0.5, although more testing could have been done to find an optimum rate.

For the PSO, a couple tunable constants for inertia (ω), c_1 (pBest) and c_2, gBest were tested. Inertia values of 0.9 and 0.4 were used, and 0.9 was observed to give unstable convergence and was oscillating between iterations. c_1 the coefficient for pBest was tested with values of 1 and 2, while c_2 the coefficient for gBest was tested with values of 2 and 3. The combination of c_1 and c_2 with values of 2 was observed to give larger incremental increases in fitness from generation to generation.

These algorithms have multiple candidates searching for the optimum with some element randomness (stochastic learning) like the GA, DE, and PSO. Even with understanding how these algorithms work, it is was hard to predict their performance on the data sets a priori, but we expected a few general trends to hold. Overall, the two-layer models were expected to perform better that the single-layer model, and the single layered model was expected to perform better than the no-layer model. This is simply because, if we give an MLP enough layers, we can get around solving linearly inseparable problems.

The models were run with 5 fold cross validation and scored based on macro averaged F-score or mean squared error depending on the data set. Macro averaged F-score was used for the Abalone, Car, and Segment data sets as they deal with classification into a finite set of classes. Mean squared error was used to calculate the variance we have between our model's prediction and the actual discrete values of the Machine, Forest Fire, and Wine data sets.

For the data sets with categorical classes, an f-score was calculated based on how frequently we made a correct guess on the class. To do this, we created a confusion matrix. To calculate the f-score from our confusion matrix, we do the following:

$$Fscore = 2 * \frac{precision/recall}{precision+recall}$$

$$precision = \frac{TruePositive}{TruePositive+FalsePositive}$$

$$recall = \frac{TruePositive}{TruePositive+FalseNegative}$$

Because confusion matrices are calculated on a per-class basis for multi-class problems, we computed precision, recall, and f-score for each class and then used a macro - averaged f-score over the folds.

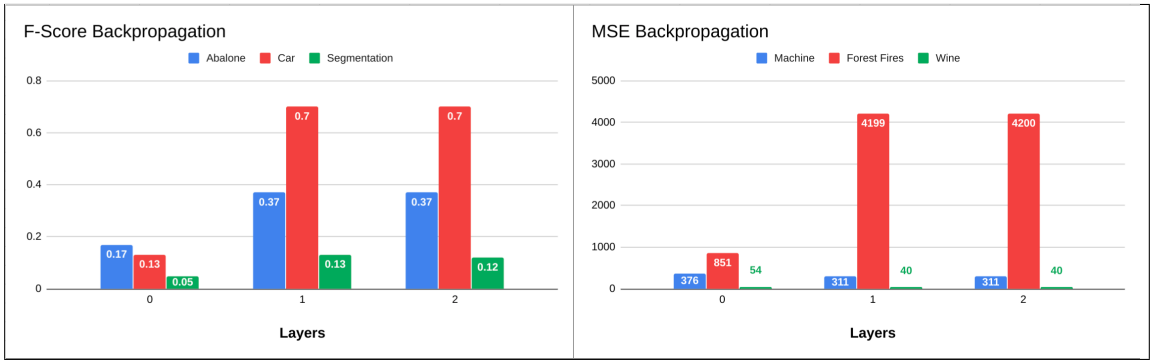
For the data sets with have non-discrete class values, the mean squared error was calculated to measure the accuracy:

$$MSE = \frac{\sum_{j=1}^{n}(ActualResult_j - PredictedResult_j)^2}{n}$$

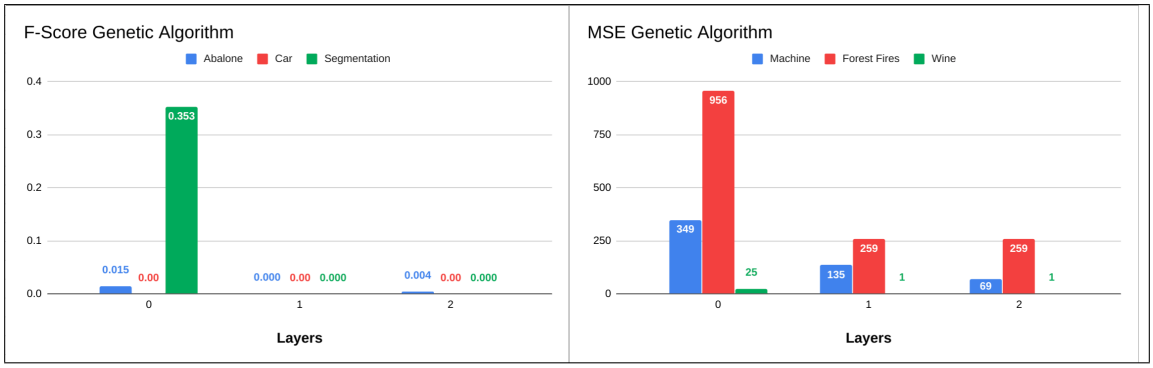
This is the same metric given earlier in the paper that was used to evaluate fitness of the candidate solution. It penalizes more heavily for predicting values that are further away from the average than predicting values that only vary slightly from the actual value.

4. Results

In assignment 3, we implemented a neural network using backpropogation as the training method. Backpropagation showed the best results for the Car and Wine data sets. Overall f-score increased as the number of hidden layers as expected. This also held true for the regression data sets, except for the Forest Fire data set, where the performance got worse as the number of layers increased. We are using this model as a baseline for comparing against our genetic algorithms.

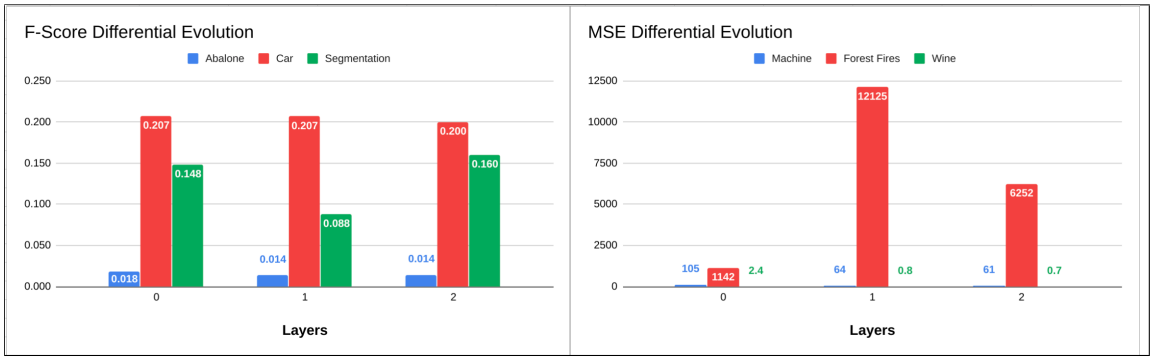


The Genetic Algorithm had trouble predicting any of the classification data sets. Which is discussed more below in the Discussion section. The regression data sets performed as expected, since the mean squared error improved as the number of layers increased. The Forest Fires had the worst accuracy while the Wine data set showed the best accuracy.



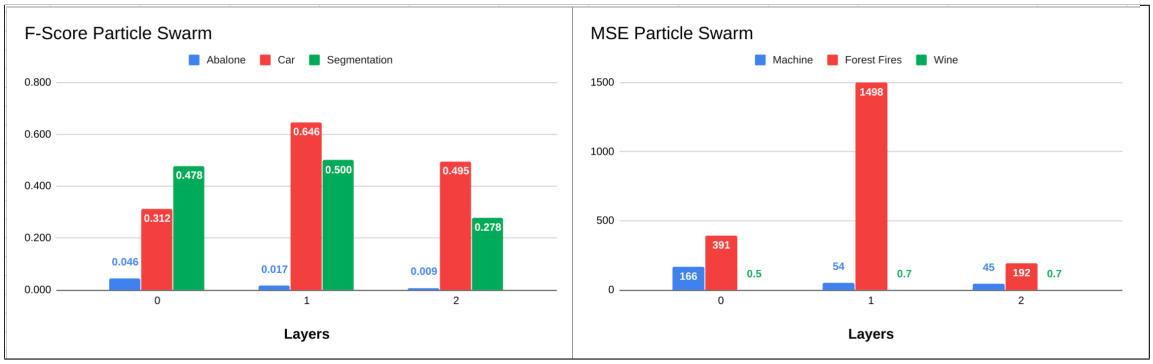
Accuracy is not shown from Abalone, Car, and Segmentation, but those results are a bit more telling for what the DE was doing when classifying. If we look at accuracy, the average accuracy of predicting the test data correctly is about even with the proportion of the common class of that data. This means that for classification, our Differential Evolution's population has evolved to predict the common class. This is not the desired output, but it is probably due to the fitness function calculation.

Forest Fires is the outlier for regression. It is hard to explain how the DE arrived at this solution. Possibly, we did not give it enough generations or members of the population to evolve to a good enough solution when running. Looking at Machine and Wine however, we can begin to see that as the number of layers increases, we are able to get better results from the DE using the same population size and generation limit.



On the classification data sets Particle Swarm optimization performed the best on Car, while it performed the worst on Abalone. Segmentation showed the best performance with 0 and 1 hidden layers, while Car had better performance with 1 and 2 hidden layers.

For the regression data sets, Particle Swarm optimization performed very accurately on the Wine data set. The Machine data set showed over 50% improvement with the increase from 0 to 1 hidden layer and the 2 hidden layer model showed incremental improvement over 1 hidden layer. Forest Fires did not perform well with any number of hidden layers with the 1 hidden layer model being far worse than the 0 and 2 hidden layer model.



If we compare the results of our Particle Swarm algorithm to our Differential Evolution, we see that the distributions are very similar. We may be able to make similar assumptions about the Particle Swarm algorithm then. We do notice slightly better results here though for regression, compared to the DE.

5. Discussion

Overall the algorithms showed better performance on the regression datasets and worse performance on the classification datasets. This seems to be highly due to how we measured fitness of the individuals during the evolution. Our first approach to calculating fitness was to take a mean squared error approach to our nodes. The node with the highest output value would represent which class would be predicted, so for a perfect perceptron, for the actual class node, we would output 1 and we would output 0 for all the other nodes. We noticed that doing a MSE approach made our models evolve to not predict much at all, as it usually needed to predict mostly zeros and only a single one. This caused the solution to favor the most common class. The most obvious example of this is for the Car data set. This population starts randomly, but evolves over time to only ever predict "unacc". If we look at the class distributions for Car, we notice that "unacc" represents 70% of our data and our DE predicts this data set correctly with about a 70% accuracy. We attempted to mitigate this by decentivising the correct prediction of a zero, which is the model we eventually used. We also attempted a fitness measure for classification similar to accuracy, to completely remove the value in correctly predicting a zero. There wasn't much of a difference in performance from this either. The regression datasets did not have the problem described above. This could be due to them only having a single solution making the mean squared error is a very direct method for fitness assessment.

When making our initial population we generated MLPs with entirely random weights which were between 0 and 1. Without seeding the weights at all, the MLP could be incredibly slow in improving. So, although we did 1000 generations, at our rate of improvement, this would not nearly be enough to move close to any decent results. This could be seen most significantly in our Genetic Algorithm. We believe that this lead to some of the zeros that we see in our data. If we got lucky and had a had a fit individual from the start, maybe 1000 generations would have been enough (like we can see in some of our better results), however, it does not seem to be enough generations for all of our datasets.

A bigger population could also improve performance. We experimented with 50 individuals in our population, but upon more research in the optimal number of individuals we found this may be low. For example, Genetic Algorithms depending on the data set and problem, hundreds and thousands of individuals are sometimes needed. (Vincent A. Cicirello, 2000) This means our small population size could have also contributed to the poor performance, since it only had a small pool of individuals competing.

Another thing we observed was the performance changes depending on the number of hidden layers in our MLP. It is hard to know if the performance changes are due to the number of layers, but for Machine and Wine we saw consistently better results with more hidden layers. The Forest Fires data set is the exception, where it doesn't look like our algorithms found a good model since all of our MSEs were very high. However, for Abalone, we saw performance decreases when adding hidden layers. This could be due to the need for a higher generation limit to train the the MLPs giving them more time to find a better solution.

Parameter tuning could also improve our algorithms. We were able to test a few different values but each dataset could be optimized to find the solution. With more time we would also like to explore this as well.

In comparison with our MLP that used Backpropagation, all of our genetic algorithms preformed regression much better. Contrarily, they preformed classification worse. This seems to suggest that or fitness measure for classification does not emphasize the desired outcomes of our prediction model and Backpropagation is a better method method for training classification datasets, but further testing is needed because there are multiple options to improve performance of both.

6. Summary

In summary, we set out to implement three different genetic algorithms: a Genetic Algorithm, a Differential Evolution algorithm, and a Particle Swarm algorithm. Each of these use feed forward neural networks as members of their populations. In reviewing the results, our models were generally unable to preform classification with much efficiency. We were able to find promising results using these models to predict non-discrete classes.

Looking at all of our results, it seems that we would be better off testing different fitness measures, as well as running the algorithms with higher population sizes.

For our regression sets, on average, our genetic algorithm models are able to predict these values better when more layers are present. We often had difficulty predicting values for the Forest Fire data set, so in order to improve this in the future, we would need to examine the reason why we had difficulty with this one.

References

Dorota Lipowskab Adam Lipowskia. Roulette-wheel selection via stochastic acceptance. 2012.

Qinghai Bai. Analysis of particle swarm optimization algorithm. 2010.

Marko Bohanec. Car evaluation database. 1997.

Paulo Cortez. Forest fires. 2007.

L. Davis D. Montana. Training feedforward neural networks using genetic algorithms. 1989.

Marine Resources Division. Abalone data. 1995.

S. Haykin H. Leung. The complex backpropagation algorithm. 1991.

N. Overgaard J. Solem. A geometric formulation of gradient descent for variational problems with moving surfaces. 2005.

C. McCormick. Radial basis function network. 2013.

Fernando Almeida Telmo Matos Jose Reis Paulo Cortez, Antonio Cerdeira. Wine quality. 2009.

Jacob Feldmesser Phillip Ein-Dor.

R. Storn. On the usage of differential evolution for function optimization. 1996.

P. N. Suganthan V. L. Huang. Differential evolution algorithm with strategy adaptation for global numerical optimization. 2009.

Stephen F. Smith Vincent A. Cicirello. Modeling ga performance for control parameter optimization. 2000.

University of Massachusetts Vision Group. Image segmentation data. 1990.

Darrell Whitley. A genetic algorithm tutorial. 1994.