Training neural networks using Genetic and Simulated annealing

Ashwin R Bharadwaj	PES1201700003
K S Sahazeer	PES1201700949
Anish Sekhar	PES1201700242

Abstract:

The assignment was to train a given neural network for a given dataset, using any weight updating mechanism besides the conventional backward propagation. Further, we were to use a heuristic of some kind for updating weights, the purpose of this being to speed up the fitting of a neural network to training data, as backward propagation takes a lot of resources. We ended up using a combination of a genetic algorithm and simulated annealing, and they involve a degree of randomness in weight updating, in the hope of converting to a reasonably accurate net faster. There is NO guarantee it is optimal, but it is certainly faster, and we keep iterating and updating until we get a reasonable accuracy.

Approach:

The neural network was represented as a set of matrices. Where each matrix represents a layer in the neural network. The updating of the weights was handled by a genetic algorithm. The rate of mutation was dynamic. The mutation rate was similar to simulated annealing, ie: the mutation rate was high at the beginning but slowly reduced over time.

Working of genetic algorithms and simulated annealing:

As in nature, the population is set with random weights at the beginning and thus they have random accuracies. Inspired by evolution in nature, the ones with the highest accuracies are considered the "fittest", and are assigned as the parents for the future generations. The selected parents are "bred" with each other, that is, their "genes" (weight values) are combined taking first half from the first parent and second half from the second parent, to produce an offspring, which is then taken along with the fit parents into consideration in the next generation (iteration).

The remaining offspring from the previous generation are terminated. Then, the offspring are "mutated", that is, random small changes are made in their weights (to imitate small mutations in each generation that weren't present in the earlier generation in nature). The amount of mutation decreases over time as the accuracy of our population of solutions increases, so initially, mutation is a high number (90), and as accuracy increases, fewer, less drastic changes are made, until we reach a pre-determined minimum mutation level. If the accuracy starts plateauing for about 25 generations, we start increasing the mutation rate again. Based on the performance of the generations the mutation changes.

Details on the implementation:

1. Firstly we read the data from the given CSV file. The instances of the data are randomized. The data is further split into training and testing data with a ratio of 20-80. The Data is also normalized, ie: the values of the attributes are between 0.0 and 1.0
2. Once the data was collected the neural networks were made. The matrices made were then converted to a vector. The matrices were changed to vectors and vice versa for convenience's sake. It is easier to predict the outputs using the matrices and easier to mutate and mate the population when using the vector.
3. Once the above is completed the training begins. For every generation, the fitness of the population is calculated. The best-performing entities in the population are chosen to be the parents and the remaining are terminated. Using the parents the new generation is created that includes the parents. This process continues till the threshold accuracy is met or the generation limit is reached.
4. While the model is training the mutation rates continuously change. It begins with a very high value and slowly goes down as the accuracy of the model goes up. As the fact that accuracy is increasing indicates that the model is going in the right direction. If the accuracy plateaus and remains constant, it indicates it reached a local minimum and thus we increase the mutation rate to try to escape the local minima.

Result:

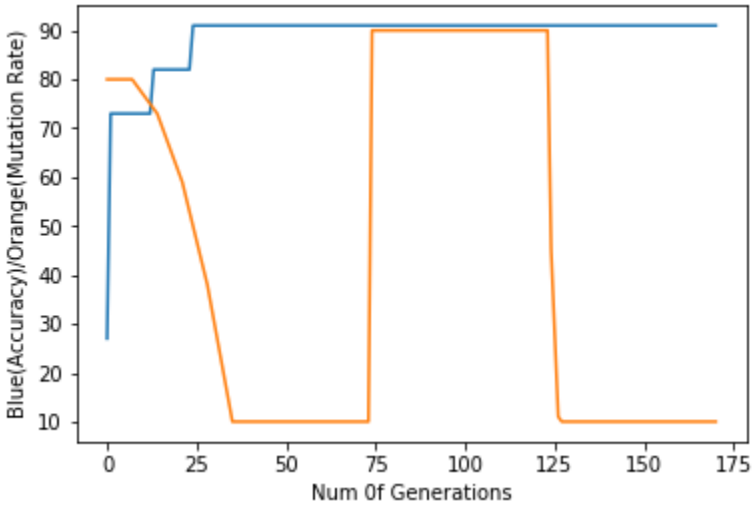
The above algorithm was tested with many variations and the best result for the given data set was obtained with the following hyper parameters:

- Entities in the population: 25
- Number of parents: 10
- Generation: 500
- Largest mutation rate: 80
- Smallest mutation rate: 10.

The time taken for 500 generations was: 49.21s

The training accuracy was: 91.0

The testing accuracy was: 95.75



As can be seen in the graph, as the accuracy of the model increases the mutation rate decreases gradually. After this the model plateaus, thus the model identifies that it may be in a local minima and thus increases the mutation rate. After increasing the mutation rate the accuracy

increases marginally thus the mutation rate is brought down immediately to prevent the model from overshooting.

Conclusion:

The method that has been implemented is a randomized algorithm, i.e. there will be no guarantee that the model will be able to find the best solution. The model relies on the fact that at every generation we take the best performing entities and modifying them (Greedy approach).
The algorithm that has been implemented also tries to avoid going into local minima by increasing the randomness. This is similar to natural evolution, i.e. if there is a drastic change to the environment the entities that would survive will not necessarily be the best among the population that were adapted to the previous environment and thus entities who have a high degree of random traits(ability to quickly mutate) will have a better chance of surviving.