CARLETON UNIVERSITY

COMP4107 - Final Project

Neural Networks Learning Through Genetic Algorithms

Author:

Brandon Marshall

Professor:

Dr. Tony White

December 16, 2017

Introduction

Once a neural network is created it is regularly desirable to modify its weights and biases to have it learn an appropriate output in a given situation. One of the most popular ways to teach a network the correct weights and biases is through backpropagation with gradient descent. This process of teaching a network works extremely well, but backpropagation requires pre-existing knowledge of what the network should output in the proposed situation. Therefore, backpropagation can require a database full of network inputs and desired outputs for the network to train on. This is undesirable as gathering a database full of inputs and desired outputs takes time and resources, and it may not always be possible. Training a neural network with genetic algorithms eliminates the need to have a database of inputs and outputs. Instead the network trains through a “survival of the fittest” competition.

Genetic algorithms are inspired by natural evolution in biology. The generic genetic algorithm is a loop that starts by having models solve a problem. The models are then assigned a fitness rating determined by the quality of their solution. The next step is to select winner models to breed and create children. The higher fitness rating a model has, the more of a chance it will have to be selected to breed. The final step is mutation, each of the children are slightly modified in some way to provide variety and introduce new information into the models. The loop then restarts with the children models. The idea is that over many iterations the best solution will be found.

Problem

The problem for which my neural networks will be creating solutions is a recreation of the game Flappy Bird. In this game, players control a dot called the “bird.” The bird can move up and down on the Y axis but has a constant forward movement on the X axis. The bird is constantly being pulled to the floor by gravity, but players can jump overcoming gravity and rise upward to a max height. The players’ objective is to get the bird to the correct height so they pass through gaps in walls. If they miss the gap and hit the wall, they die.

Methodology

The first step in solving this problem is to decide how the neural networks of my players will be structured. In Flappy Bird there are two actions a player can take at any given moment. The first option is to do nothing, allowing gravity pull the bird downwards. The second is to jump, giving the bird a short burst of upward velocity. These two actions will make up our output nodes: one node to indicate a jump, the other to indicate no action. As the goal of Flappy Bird is to make it through the gap of the wall, the features I decided to use as input nodes are the X and Y distance components from the player to the middle of the gap. Finally, the middle layers of our network are arbitrary as they are hyperparameters.

Now that we have chosen our network architecture, we must develop a fitness algorithm to determine which network has performed the best. The objective of my fitness algorithm is longest distance traveled. As the distance traveled increases the fitness of a solution does as well. However, I made a few modifications to this algorithm. The fitness algorithm must be able to distinguish between each of the birds, even if their solution only varies in a small way. If I did not do this, then I would run into the problem of all birds running into the same wall, thus all having the same fitness, no matter

how close they were to the gap. I avoided this by calculating the distance traveled minus the distance to the closest gap. This modification means that the birds that die closest to the gap have higher fitness than the birds that die further away. The next modification I made was to freeze the bird's distance traveled counter while the bird is at the maxheight or minheight. This change was made because due to the games random nature of having a gap in the wall, spawn anywhere between the maxheight and the minheight, networks that were only outputting one type of action (always jumping or performing no action) would sometimes be accidently rewarded. This was because the gap would happen to spawn at the max or min height and the bird would pass through it. Essentially this change rewarded networks that performed more than one action.

Now that we have our fitness for each of our networks we move onto the selection process. Here we must decide who the winners are and how to breed them. Elitism is the name of the method that selects the a few of the best performers and passes them directly onto the next generation. Using this method helps prevent the network from relearning good solutions. In my genetic algorithm I use this method and pass on the most fit network of every generation. Then I choose an arbitrary amount of winners out of the population (including the one passed on by the Elitism method) and save the networks. Each bird in the population (excluding the one passed on by Elitism method) then has a high chance (known as crossover rate) of being bred with one of the four winners, and a child is passed onto the next generation.

Breeding is the process of creating children. With neural networks this is done by swapping the weights and biases at an arbitrary number of crossover points between two parent networks, thus creating two children. A crossover point is a point chosen randomly between zero and the number of weights or biases. For example, imagine we have two arrays A = [1,2,3,4] and B = [A,B,C,D] and we have one crossover point at position 2. This would result in two children A' = [1,2,C,D] and B' = [A,B,3,4].

Mutation is the final step after breeding is completed. Each of the networks that are moving on to the next generation (excluding the network selected from Elitism method) are subjected to mutation. Mutation is when several variable weights and biases are selected, then each of the selected weights and biases has a chance (known as mutation rate) of being varied by a small degree (known as a mutation step).

One of the major problems with genetic algorithms is that it's easy to fall into a local maximum. One of the problems I had during early testing was that sometimes the birds were falling into a local maximum, and they had determined that the best solution was to perform no action at all. To avoid this I implemented simulated annealing and diversity selection, both of which were solutions proposed by Patrick Winston, an instructor at MIT.

The idea of simulated annealing is simple: start with a large search area and reduce it as time progresses. To do this in a genetic algorithm, you start by having a large mutation step, then reduce the nudge every generation. This ensures that your networks have lots of variation between them, and because the best are chosen as winners each time you will eventually hone in on the the best solution.

Diversity selection is the idea of considering diversity along with fitness in choosing the winners in the selection process. Because you're looking for diversity along with fitness, your population will be very diverse and some of your population will be in the optimal maximum. Once this happens or after an arbitrary number of epochs you stop looking for diversity and only focus on fitness. This will make your population converge on the optimal maximum. To consider diversity with fitness you must first develop a way of finding diversity. In this case, to determine diversity, I chose to

calculate the euclidean distance of the weights and biases of the network in question to each of the birds in the winners list, then divide by the length of the winners list to get the networks diversity score. If the winners list is empty, the network with the highest fitness is added to the winners list (no diversity calculations are made). Once you have made your diversity calculations you must determine which network belongs in the winners list. Start by assigning a diversity rank to each of your networks, with one being the most diverse. Then assign a fitness rank to each of the networks, with one being the most fit. The final score of the a network would be the addition of the diversity and fitness ranks. Then you add the network with the lowest final score to the winners list, because it is the most diverse and the most fit. You repeat these steps of calculating diversity and assigning ranks until your winners list is full.

Experimentation

With my experimentation, I wanted to know how simulated annealing and diversity selection stacked up against each other. I'm interested in how many times each method falls into a local maximum and how fast the networks learn. I will test four variations of my genetic algorithm, the first variation will implement both simulated annealing and diversity selection. The second variation will only implement simulated annealing. Variation three will only use diversity selection, and the fourth variation will have neither simulated annealing nor diversity selection. I will run each variation ten times with a varied amount of generations, and measure two things. First, I will measure how many birds make it past five walls for any given generation. This is because if it can pass five walls I consider it to be a champion and to have learnt the game. Once the birds are accounted for, I will kill them and move on to the next generation. The second measurement I will make is the number of times that after forty generations, no birds make it past any wall. If no birds make it past any walls, it indicates that we have fallen into a local maximum.

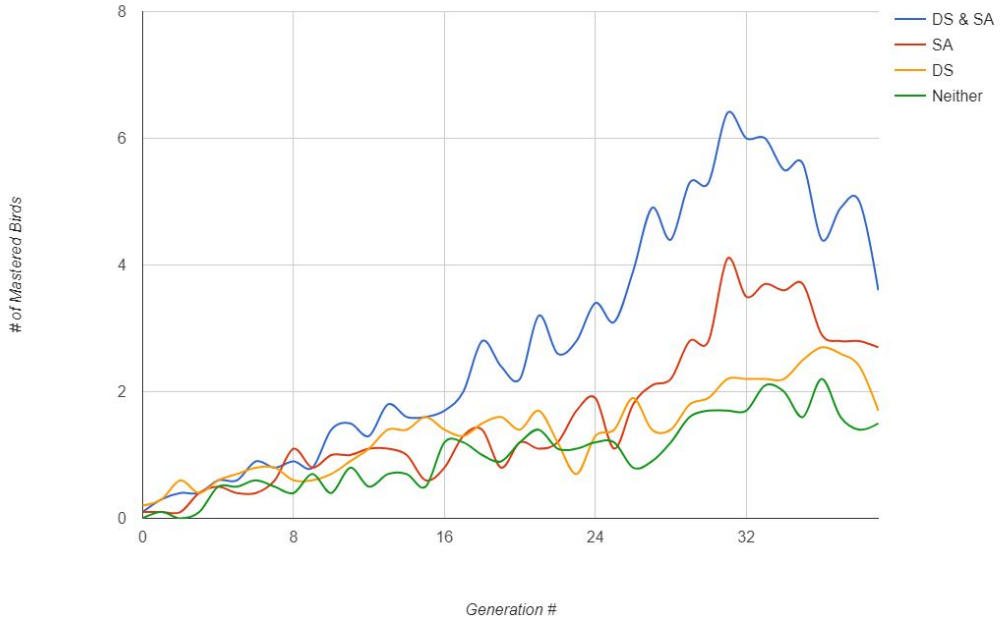
The network architecture used in my tests have two layers of hidden nodes: one with fifteen nodes the next with five. I have chosen to have a large amount of hidden nodes because this will reduce the chance of the network being randomly initialized to a winning solution, thus increasing the role learning has.

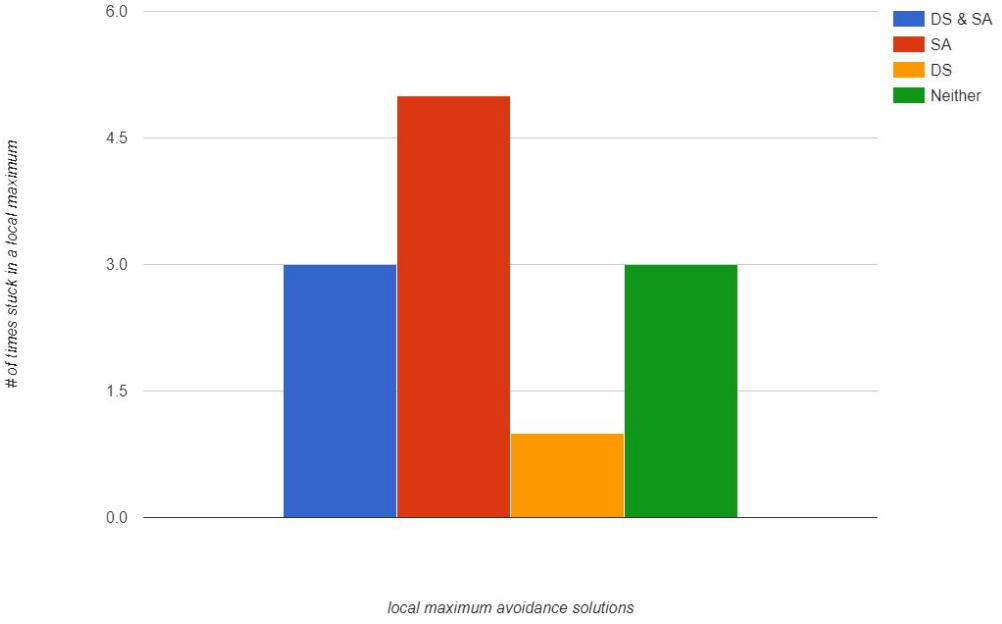
I performed three tests, the hyperparameters of the genetic algorithm for each test are shown in the table below. Over the three tests I varied four of these hyperparameters (Generations, Mutation Step, Generations Until Mutation Step is Negligible, and Generations Until Only Fitness is Considered). My reasoning for modifying these variables will be explained in the next section.

Hyperparameter	Test 1	Test 2	Test 3
Iterations	10	10	10
Generations	40	40	**100**
Population	10	10	10
Number of Winners	4	4	4
Crossover Points	1	1	1
Crossover Rate	70%	70%	70%
Number of Mutations	10	10	10
Mutation Rate	70%	70%	70%
Initial Mutation Step	+/- 0.25	**+/- 0.50**	+/- 0.25
Generations Until Mutation Step is Negligible (Only Enabled With Simulated Annealing)	30	30	**85**
Generations Until Only Fitness is Considered (Only Enabled With Diversity Selection)	25	25	**80**

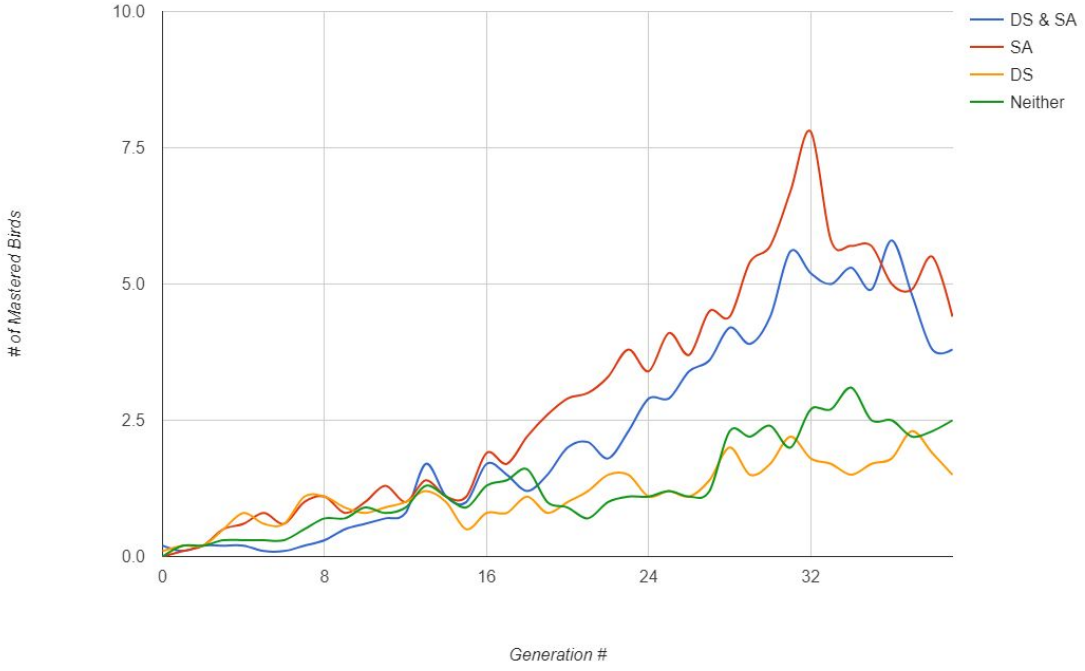
Results

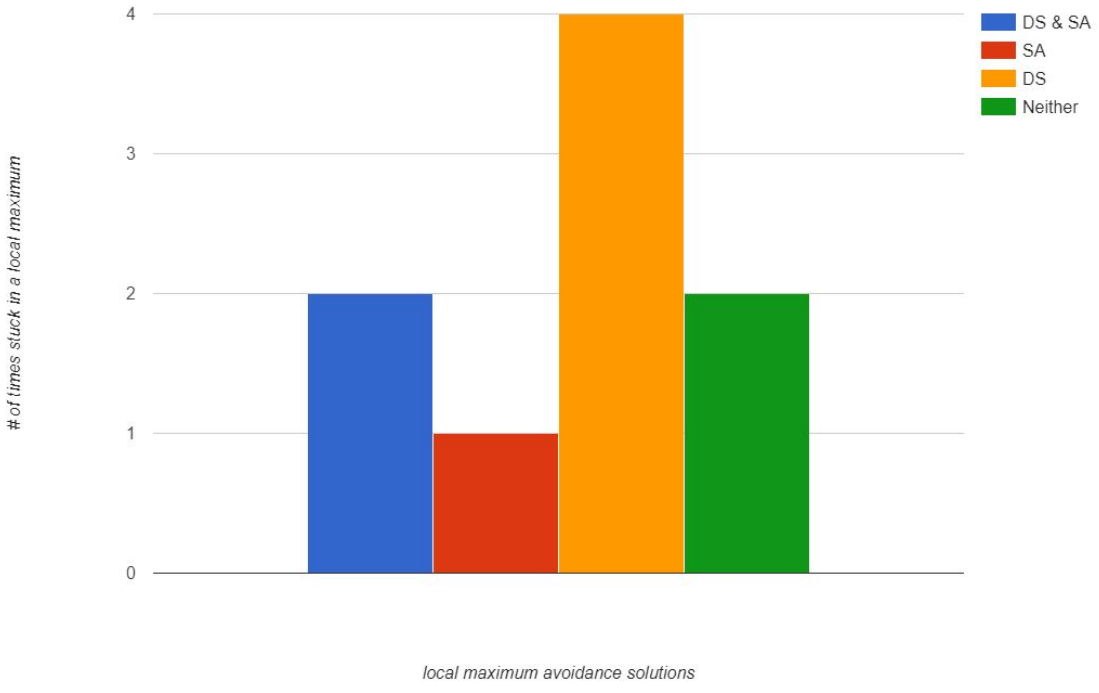
Test 1





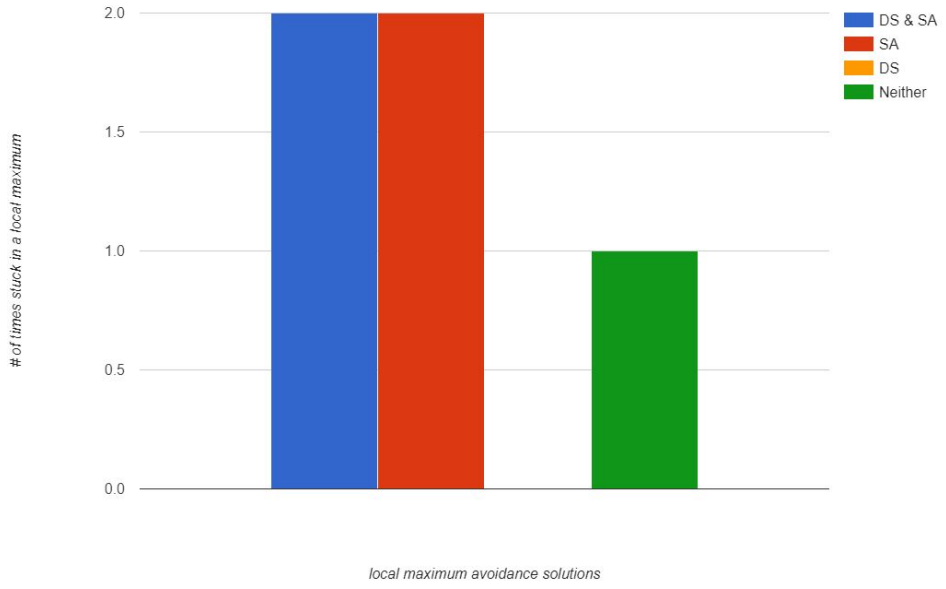
Test 2


10.0
7.5
5.0
2.5
0.0
0
8
16
24
32
DS & SA
SA
DS
Neither
# of Mastered Birds
Generation #


4
3
2
1
0
DS & SA
SA
DS
Neither
# of times stuck in a local maximum
local maximum avoidance solutions

Test 3


12
8
4
0
-4
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
# of Mastered Birds
Generation #
DS&SA
SA
DS
Neither


2.0
1.5
1.0
0.5
0.0
# of times stuck in a local maximum
local maximum avoidance solutions
DS & SA
SA
DS
Neither

Results Analysis

In test 1, we can see that simulated annealing was the worst and got stuck in a local maximum five times, diversity selection got stuck once, and the combination of simulated annealing and diversity selection did no better than our basic genetic algorithm. This is surprising because one would initially think the combination of simulated annealing and diversity selection would do the best, while both local maximum avoidance techniques on their own would do better than the basic genetic algorithm. Since simulated annealing requires a large initial mutation step, I figured that +/- 0.25 wasn't high enough for this technique to work properly.

In test 2 I set the mutation step to +/- 0.50 which effectively doubled the mutation step from test 1. In the results we can see that this change drastically helped simulated annealing, but hurt diversity selection. In Patrick Winston lecture he implied that diversity selection should be the most resistant from falling into a local maximum. I estimated that due to the fact that diversity selection takes diversity into account as well as fitness every generation, it may take longer for diversity selection to find an optimal solution.

In Test 3 I increased the number of generations from 40 to 100 and returned the mutation step back to +/- 0.25. The results show that this helped all variations of our genetic algorithm as opposed to the results we got in test 1. The most important takeaway from test 3 is that we have shown diversity selection never fell into a local maximum. This is significant because this provides evidence that diversity selection is indeed the best technique out of the two, to avoid falling into a local maximum.

Conclusion

The Flappy bird problem can indeed be solved by training a neural network through genetic algorithms. The major pitfall in using genetic algorithms as a training tool is the possibility of falling into a local maximum. Techniques such as diversity selection and simulated annealing exist to combat this problem, and can be implemented into the genetic algorithms. Through our testing, we found evidence that the hyperparameters given to the genetic algorithm largely dictate how each of these techniques perform. With a large enough mutation step simulated annealing works quite well, falling into a local maximum around 10% of the time. When given a large amount generations the diversity selection technique worked the best, as it never fell into a local maximum. In short, if you are restricted by number of generations you can perform, use simulated annealing, otherwise, diversity selection is the better technique to use.

References

Winston. P. (Jan, 10, 2014). 13 Learning: Genetic Algorithms.
https://www.youtube.com/watch?v=kHyNqSnzP8Y&t=1897s

Kemp. R. An introduction to genetic algorithms for neural networks.
https://www.phase-trans.msm.cam.ac.uk/2003/MP9/MP9-5.pdf

Mahajan. R. Kaur. G. (Sept 2013). Neural Networks using Genetic Algorithms.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.403.1652&rep=rep1&type=pdf

External Libraries

pygame. https://www.pygame.org