2020 Computational Intelligence Coursework

40205163

ABSTRACT

After taking a conventional approach towards the Lunar Lander Genetic Reinforcement Learning Problem, I re-analyse the initial problem. In this paper I created an novel approach to Evolutionary Algorithms, compared to common EA literature I've read - based on my understanding of the problem. EAs usually start with a diverse population of agents and then use various exploration and exploitation methods to try and find a global optima. These techniques help them avoid local optima and search across the space of possible solutions.
In this coursework, I start with an un-diverse population of agents and attempt to mutate them into diversity.

ACM Reference format:
40205163. 2020. 2020 Computational Intelligence Coursework. In *Proceedings of Coursework, Edinburgh Napier University, April 2020 (SET10107),* 4 pages.
DOI: none

1 INTRODUCTION

This is a coursework set by Edinburgh Napier University, in which students were asked to design a genetic algorithm, modifying a neural network, which landed Lunar Landers on a landing pad.

2 APPROACH

In this section I discuss my approach towards the coursework and the operators and algorithms I implemented for the project. I have tried to describe my approach step-by-step, to allow the reader to follow my thought process.

2.1 Background Reading

To help understand Reinforcement Learning and neural networks for this project and my dissertation, I read "Reinforcement Learning: an Introduction" [1] and "The Hundred-Page Machine Learning Book" [2]. To help understand Evolutionary Algorithms, I read Chapters 3-6 of "Introduction to Evolutionary Computing" [3].
The two books about neural networks helped me shape my activation function, my network size and parameters. The Chapters on Evolutionary Computing helped me understand mutation, fitness, selection, some of the popular Evolutionary Algorithm variants and their use-cases, and helps me with my later implementation.

SET10107, Edinburgh Napier University

DOI: none

2.2 Problem Description

For this problem, we were given a Java environment in which a neural network controlled the behaviour of the Lunar Landers. The neural network we were using is called a perceptron [4] - an Artificial neural network with a single hidden layer. It is also an example of a Feedforward neural network, since the connections don't loop back on themselves.
Depending on the problem, they would start in predefined or random locations, and then would attempt to fly to the landing pad (the small green bar at the bottom of the screen). If they hit the landing pad too hard, too fast, or from too great an angle, the rocket would explode. These rockets also had a limited amount of fuel to make the landing pad, and three actions they could take: firing the left thruster, right thruster or the bottom thruster. Note: these actions could be taken at the same time as. Fitness would be determined by how close the rocket landed to the centre of the pad, and how much fuel the rocket had left. If the rocket had more fuel, the fitness would be decreased.
There was an incomplete Evolutionary Algorithm which controlled the mutation of this neural network, controlling the evolution of the fitness of the network. Our job was to complete the Evolutionary Algorithm and update the EA functions, to allow the proper convergence of the network, and the rockets to land. Specifically, we were to update the selection, crossover, replacement and mutation functions, and anything else we felt appropriate. We were given a number of parameters allowing us to tune the network and the EA. Figure 1 shows rockets attempting to land after a set of training runs.

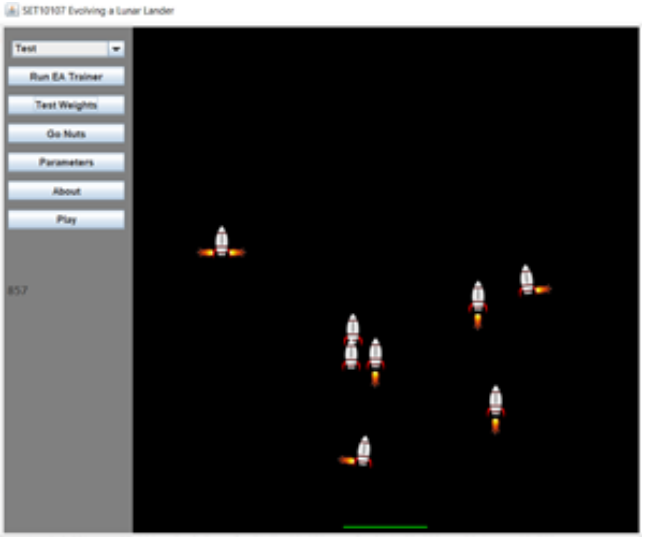

Figure 1: Lunar Lander Environment

2.3 Original Approach

I initially worked with a population of 50 agents across 8,000 runs, as it allowed me to train and re-iterate my model quicker. I originally implemented an algorithmic crossover, but found that I needed a much lower tournament size because it converged too quickly - meaning it had less impact than I was hoping for. As an activation function, I changed the Hyperbolic Tangent (tanh) function to the Rectified Linear Unit (ReLU) function, as it has been shown to speed up performance several times with similar or better results across image processing and other tasks [5].

A big part of the initial implementation was exploring the coding environment, the available classes and the methods we were to code. Throughout the implementation process, I kept varying the weights on the neural network and re-running, anywhere from 2 to 10 weights, to see the effect it had. I found that some of the smaller models had issues representing the complexity in the environment, while the larger models overfit regularly. For some of the largest models, I kept running into the emergent behaviour where all Lunar Landers would converge to a narrow space on the screen and then attempt to land based on a set of X and Y coordinates it was familiar with. For some of the smaller models, the emergent behaviour that kept arising was that all Lunar Landers would fire their blasters at the same time, and then shut them all off at the same time, as they perhaps hadn't got a large enough structure to map the additional environmental complexities.

After training several times on the initial parameters, I switched to 200 population size and 12,000 runs to more deeply analyse my trained models. I was regularly achieving 0.05 or less over 12,000 runs. I had a one-point crossover algorithm, selected from 50 candidates (re-selected if both candidates were identical).

I felt however for the implementation, there was a far more interesting and possibly intuitive approach. Since the outputs were binary (the engines would fire if 0, otherwise wouldn't fire), it felt that gene values of +3 or +3 (or even +1 and -1) may be unnecessary considering they only needed to be slightly positive to fire.

A Quick Example

If there were four hidden units in the hidden layer, three each with 0.001 output weightings and one with -0.003 output weighting. If all fired at the same time, the thruster wouldn't fire, due to it being 0. If the negative weighting unit stopped firing, the thruster would re-fire since the output neuron weighting is now +0.003. This would be the same for three weightings of 1 and one of -3, and three weightings of 10 and one of -30, since the output doesn't scale.

There is still a requirement to map some of the environment complexity into the neural network, since it's important to determine what coordinates blasters are to be fired at. I quickly changed my approach to the original problem. I analysed the gene starting values for runs, since I wanted to find how 'good' the starting value was. Note: these include local and global optima, there isn't one optimal value.

The results are displayed in table 1 for the initial gene value across 10 runs.

Gene Value	Mean	Std. Deviation
0	0.45	0.120
0.001	0.47	0.127
0.2	0.54	0.116
0.6	0.49	0.124
1.2	0.52	0.114
1.6	0.55	0.096
2	0.49	0.112
5	0.51	0.132
10	0.56	0.128
20	0.50	0.097

The results seem to show that there tends towards higher error at the larger values of ± Gene Values. I would tend to assume this is due to the fact that there are larger initial action spaces for the higher Gene Values, and that causes additional variation in possible initial values. In response, based on my knowledge of Neural Networks and Evolutionary Algorithms, I wanted to create an Evolutionary Algorithm that would be ideal for fitting to binary outputs from a Neural Network. Note that the problems with smaller values also tended to converge a lot quicker than the larger values (but would have a lot less initial population diversity).

Note: I only took this approach due to the nature of the problem. If it had been a classifier, a regression problem or an Reinforcement Learning problem where the agent had to choose between (a robot that can't move a leg to go left and right at the same time) this approach would have been entirely inappropriate, but since all blasters could be fired at the same time, I felt it had a chance of working. It's also interesting to note here that the initial starting values for ±0 were either 0.37 or 0.63, while there should have been no variation due to the nature of the zeroed values.

2.4 New Approach Experimentation

From table 1, it shows that 0.001 and 0 performed the best initially. I wondered if there was a way of mutating heavily towards a global optima from 0 or 0.001 as opposed to genetically improving from a diverse population.

I set my ± Max/Min gene to 0. This would give me a starting population of zeroed agents. I raised my mutation rate to 0.7, to see how the heavy mutation worked.

Initial results were promising, so I decided to continue testing the idea. One of the initial issues experienced however, was that the mutation would increase and then decrease again, since it was a 50/50 chance of increasing and decreasing, so chromosomes would still tend towards 0. I then raised my mutation rate from 0 (positive chromosomes encouraged to increase, negative chromosomes encouraged to decrease) to encourage mutation from 0 (initially 60%). This helped mutate values away from 0, but it did so in regular increments. I needed something to help break the regularity. I implemented Algorithmic Crossover, which helped break local optima and added a bit of non-regularity. I found it wasn't effective enough though, so I added a multiplier of 0.8 to already negative decreasing values, and positive increasing values.

At this point, I was regularly recording between 0.08 and 0.12. I decided to modify my original one-point crossover to a two-point crossover to try and further increase the exploitation and converge my fitness, and swap further genetic material.

I found I could combat the linearity in increase and decrease by

multiplying the mutated figure by a random double between 0 and 1. This meant I increased the mutation value to 0.6 (quickly decreasing to 0.5), to deal with the multiplication of an average figure of 0.5.
I found that I was losing too much genetic material in the pool, since I was selecting two members of the population, from a relatively large set of options (20% of the pool to 30%). To combat this, I started an 'elite' and 'normal' selection, to try and select from different portions of the population. The elite selection was from roughly 30% of the population, while the normal selection was from roughly 5% of the population. I played around with mixing only elite and only non-elite on the arithmetic and two-point crossover, elite and non-elite on both.
I ended up settling on picking one elite candidate and two non-elite candidates. I use the elite candidate for the two-point and arithmetic crossover and use one non-elite candidate for each. This helps vary the operations performed.

I spent a lot of time messing around with high mutation values for a lot of variance, and then trying to converge the model with high selection pressure from the tournament selection.

2.5 Neural Network Parameters Implemented

Since the problem isn't extremely complex, I tried to limit the total number of nodes. I found that 2-3 led to high variance and 9-12 led to high bias. I thought 6 neurons was optimal, as it felt like a comfortable number for 3 actions. I found after the additional testing that 8 neurons provided a better average and more consistent results, so I ended up settling on 8.
As mentioned in the Original Approach section, I implemented a ReLU very early on, due to the performance benefits over the tanh function, and for the fact it seemed appropriate for a binary problem.

3 ANALYSIS

3.1 Final Parameters

My final parameter settings were:
Number of hidden nodes: 6
Min Gene: 0
Max Gene: 0
Pop Size: 200
Max Evaluations: 20,000
Elite Selection Size: 50
Selection Size: 8
Mutation Rate: 1
Mutate Change: 0.5 (*Random double from 0-1 for each)

The following tables explore the mean and standard deviation for 6-neuron and 8-neuron networks (10 runs each).

Mean	**6 - Fitness**	**8 - Fitness**	**6 - Lives**	**8 - Lives**
Training	0.043	0.028		
Test	0.100	0.101	0.588	0.613
Random	0.124	0.102	0.538	0.663
Go Nuts	0.175	0.173	0.424	0.483

Std Dev.	**6 - Fitness**	**8 - Fitness**	**6 - Lives**	**8 - Lives**
Training	0.036	0.023		
Test	0.105	0.071	0.415	0.347
Random	0.109	0.080	0.426	0.367
Go Nuts	0.120	0.110	0.363	0.285

The 6 hidden-node network struggled sometimes with convergence, but when it converged it gave some very solid results. I have included the standard deviation table to show the issues with convergence.
My top 3 (of 10) 6 hidden-node networks gave an average survival rate of 0.94 and fitness of 0.03 across the training, test and random sets.
The 8 hidden-node network also struggled slightly with convergence, but less so than the 6 hidden-node network. It took a lot longer to train however, for slightly better results. It tended to outperform the 6 hidden-node network.
My top 3 (of 10) 8 hidden-node networks gave an average survival rate of 0.92 and fitness of 0.03 across the training, test and random sets.

This means that the top three 6-node networks actually outperformed the 8-node network.

4 CONCLUSIONS

My approach was less successful than if I had approached it as a normal evolutionary algorithm. I feel that given more time, I could have crafted better parameters and operators for my network.
However, I'm far happier with the implementation that I did, than if I had implemented an original Evolutionary Algorithm. I feel that my approach in questioning the fundamentals of EAs, testing parameters, coming up with solutions to population issues and thinking in-depth about a completely different approach was highly successful. I have far more understanding of Evolutionary Algorithms as a result of this coursework.

5 SUMMARY

Neural Networks have become extremely commonplace across the field of Artificial Intelligence, due to the recent success of Deep Learning [6], [7]. Evolutionary Algorithms are an extremely plausible way to train them however, as opposed to Stochastic Gradient Descent, or value-based or policy-based methods [8].

In this coursework we can see the power of Evolutionary Algorithms for training neural networks.

I've examined a novel approach to Evolutionary Algorithms in my report, which I have tried to optimise based on my understanding of EAs and the underlying problem.

6 FUTURE WORK

I'd love to further explore the ideas that I've presented here. One of the ideas that could inspire future work is directed Evolutionary Algorithms, as opposed to random exploration [9]. Seeking for solutions based on novelty seems an exciting approach, and could have a huge impact on convergence.

Getting the algorithm to consistently converge has been one of the biggest challenges. I'd like to implement functions that would assist the algorithm with general robustness to finding solutions.

I'd also like to implement something similar to population based training (PBT) [10]. It would be extremely helpful for tuning hyperparameters, since it's extremely time consuming, and I can safely guarantee my hyperparameters are likely to be sub-optimal for the problem.

7 REFERENCES

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*. The MIT Press, 2018.

[2] A. Burkov, *The hundred-page machine learning book*. Andriy Burkov, 2019.

[3] G. Eiben and J. E. . Smith, *Introduction to Evolutionary Computing*. Springer International Publishing, 2nd ed., 2015.

[4] Rosenblatt, "Rosenblatt.pdf," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *NIPS*, pp. 1–1432, 2012.

[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[7] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, and M. S. Lew, "Deep learning for visual understanding: A review," *Neurocomputing*, vol. 187, pp. 27–48, 2016.

[8] F. P. Such, V. Madhavan, E. Conti, K. O. Stanley, J. Lehman, and J. Clune, "Deep Neuroevolution : Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning," 2018.

[9] E. Conti, V. Madhavan, F. P. Such, J. Lehman, K. O. Stanley, and J. Clune, "Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents," *Advances in Neural Information Processing Systems*, vol. December, no. Nips, pp. 5027–5038, 2018.

[10] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu, "Population Based Training of Neural Networks," 2017.