EVOLUTIONARY ANN AND ENVIRONMENT INTERACTION

MODELING NEURAL NETWORKS BY ENVIRONMENT INTERACTION AND EVOLUTIONARY ALGORITHMS.

BY

HÅVARD GODAL
OVE JØRGENSEN
JOHANNA KINSTAD
VEGARD RONGVE
MARIUS SØRENSEN
BJØRN CHRISTIAN WEINBACH

University of Stavanger
Stavanger

DECEMBER 6, 2020

Abstract

Evolutionary algorithms is a subfield of evolutionary computation for global optimisation inspired by biological evolution. Evolutionary algorithms is in some cases an alternative approach for modeling neural networks compared to training neural networks with backpropagation or similar algorithms. Our results show that introducing a wide range of diverse agents will yield some agents that perform extraordinarily well but the high mutation rate also makes it difficult ensuring that the population explores the space of possible weights and biases towards a minimum state. To combat this, dynamic mutation rate and selective mutation have been implemented and have resulted in populations of agents able to handle OpenAI gym's cartpole environment with scores up to 50000.

Contents

Contributions

Throughout the period of this project the group worked closely together on most aspects regrading the project. We started out in the first iteration, setting up starting points for all the different areas of the project in order to get the program to run somewhat satisfactory. This is where most of the independent work and contributions were made. From there we focused on improving the results we had already achieved. For the second and last iteration we spent a lot of time discussing the best ways to improve the code.

The code on github[1] is documented well and every function has an associated author with it. This code's is also submitted in the zip file uploaded on Canvas. The main script is contributed by all the members. If one want a more detailed description of contributions, see the repository's insights tab or use vscode's Gitlens extension to see line-by-line contributions.

- Håvard Godal - Crossover methods
- Ove Jørgensen - Mutation methods, average network
- Johanna Kinstad - Mutation methods, visualisation
- Vegard Rognve - Single point crossover, swap mutation, performance plot
- Marius Sørensen - Dynamic mutation, combat learning stagnation
- Bjørn Christian Weinbach - Creation of agents and population, dynamic mutation rate, partial fit

[1] https://github.com/bcwein/DAT540-EvolutionaryANN

Introduction

Developing agents that can accomplish challenging tasks in complex, uncertain environments are a key goal of artificial intelligence. Mathematically one can describe an intelligent agent as function [1]

$$f : P* \rightarrow A \tag{1}$$

Where $P*$ is a set of perceptions the agent receives and A is the set of actions after observing these perceptions. A key challenge in the field of artificial intelligence is finding the function f that solves a given task sufficiently. One way to approximate this function is to use evolutionary algorithms and neural networks and letting agents learn through interaction with its environment.

Evolutionary Algorithms

An evolutionary algorithms is a subset of evolutionary computation. These algorithms are used for global optimisation. In evolutionary computation, an initial set of candidate solutions is generated and iteratively updated. Each new generation is produced by stochastically removing less desired solutions, and introducing small random changes.

In our case, we will use evolutionary algorithms to approximate f. Our agents will get a score based on how well they perform in the environment and the agents that perform well are able to reproduce and we get a new generation of, hopefully, better agents. The agents also need to have a probability of mutating. This helps to ensure that the agents don't converge on a local minima. By randomly mutating agents, they are forced to explore the function space of f.

The environment

We will use the OpenAI gym's CartPole environment. This environment comes with action space, perception space and a fitness function already implemented and we can focus on the implementation of our genetic algorithms.

Structure of the project report

This report will present our code in iterations. Where we have the following workflow between iterations.

1. Implement algorithms
2. Visualise performance
3. Discuss results
4. Plan new implementations based on results

The aim of our project is to show what results we find, comparing different approaches to one another and iteratively improve the evolution of agents. This will hopefully provide some insight in how one best approaches this particular problem.

Code and installation

For any questions regarding setup and installation of the software we will reference to this projects github repository [2] and the README in particular. Still, some explanation for our way of organising our code will be given in this section.

Anaconda Environments

In the repository's README, installation of dependencies are handled by Anaconda. [3] Anaconda provides us with some great features, among which are:

- You can entirely define an environment, including the version of python, using yml files.
- It has a much more powerful dependency solver than pip, making it less likely you'll end up with an inconsistent environment
- It works across platforms.

Anaconda is therefore "necessary" to have installed on your computer for running our environment. It is also possible to manually install dependencies via pip or by other means. We have defined an anaconda environment for later use, making it easier to install previous versions of python and libraries that the code is made for.

Style guidelines, formatting and docstrings

The code in this project follow PEP 8's guidelines [4] and all code is linted using pycodestyle and pydocstyle, which enforces style and function documentation. This formatting have also been enforced by using githubs continuous integration features. When code is commited or merged with the main branch, it is tested for docstrings and code style and either passes or fails.

Every functions docstring states the functions author, if one want to address the project group about one of the functions in the source code, address the author named in the docstring.

Workflow

The project group has made extensive use of github to manage it's workflow. Tasks have been managed by the use of github issues. This has allowed us to share the load of the work among all members. We have used pull-request as a way of quality control as well as a way to engage other members of the project group to code that they themselves have worked on.

If one is interested to see how the tasks have been shared among the members or investigate details of a certain piece of code, there will be issues documented in the repository.

[2] Github Repository
[3] Anaconda
[4] pep8

First Iteration

For the first iteration of the code, functionality was implemented according to suggestions provided in the project description. Our first implementation was based on the following algorithm:

1. Generate the initial population of individuals randomly. (First generation)
2. Repeat the following re-generational steps until termination
   (a) Evaluate the fitness of each individual in the population (time limit, sufficient fitness achieved, etc.)
   (b) Select the fittest individuals for reproduction. (Parents)
   (c) Breed new individuals through **crossover** and **mutation** operations to give birth to offspring.
   (d) Replace the least-fit individuals of the population with new individuals.

Initial population generation

In this iteration, the initial population is generated by generating several neural networks that are partially fitted on a sample of observations O and a sample of actions A. Where these are sampled according to a uniform distribution

$$A \sim \mathcal{U}(\min(a), \max(a)) \tag{2}$$

and

$$O \sim \mathcal{U}(\min(o), \max(o)) \tag{3}$$

And by partially fitting the agent to these random observations and actions. We have initialised a population of agents. Then we let them start interacting with their environment.

Network evaluation

In order to determine when we should consider a network a success we made the decision to attempt to reach the translated scores presented on the CartPole-v0 leaderboard. CartPole-v1 states that in order to consider a network a success one should reach a reward score of 95% out of the max score. With this in mind, we decided to set the max possible value to 500 in order to limit the search. It is also defined that if the average score of the last hundred agents is 475, then the generation is considered a success and the simulation is terminated.

Crossover

Crossover is a genetic operator used to combine parts of two parents to create new offspring.

The first iteration of the program used a two-point crossover method. In a two-point crossover method, two random indexes i, j are chosen from the parent chromosomes. The genes that lies inside between the chosen indexes are copied over to the offspring from one parent, while the genes that lies outside the indexes are copied over from the other parent. This method produces one offspring per method call, similar to the crossover method from the lectures.

The random indexes i and j are chosen according to a uniform distribution.

$$i \sim \mathcal{U}(0, \text{end index}) \tag{4}$$

and

$$j \sim \mathcal{U}(0, \text{end index}) \tag{5}$$

Mutation

Mutation is used to maintain genetic diversity from generation to generation and in this iteration, a simple swap-mutation is implemented. Two rows on the chromosome are randomly selected and then swapped, resulting in a mutated chromosome. Indexes are as above, uniformly distributed.

Visualisation

To test our results from the first iteration we visualised 3 separate networks post training. Each network was run with the same number of generations and a different mutation rate as this was the most interesting factor at this point.

To perform the visualisation we utilised a tool called Weights & Biases [5]. This is a tool that allows you to track code runs and log specific data and plot the results while the simulation is running. This made it easy for us to view results from different simulation as they are all displayed in the same plot, which again gives us further insight into the difference between a good and an average network.

This tool was used throughout the entire project in order to track changes in performance as we made changes to the algorithm. The code for visualising using Weights & Biases is only present on a specific branch and not merged with the main as it requires a personal account in order to run.

To still be able to present a visualisation of the result after a run, a simple visualisation function was also implemented. This function returns a single plot of how the network performed throughout the simulation.

[5] Weights & Biases

Result

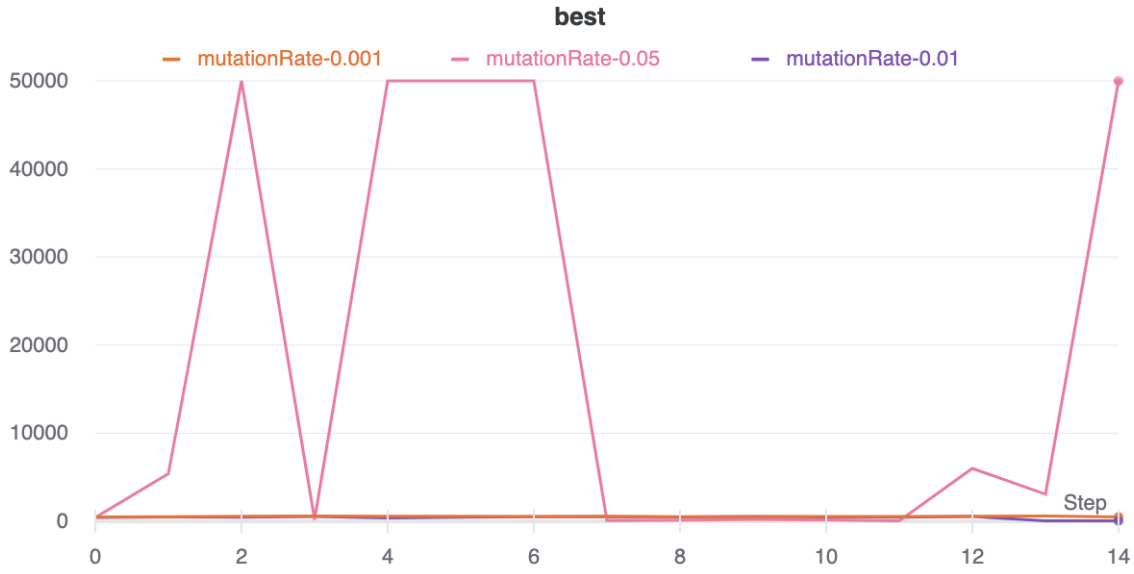

Figure 1: Results of various mutation rate

Our results are visualised in the figure above. We observe that we struggle to get good scores when the mutation rate is < 0.05 and we sometimes get incredibly high scores when having a mutation rate of $>= 0.05$. This suggests that our agents are not quite diverse enough and they get stuck in a local minimum. With a high mutation rate, we can sometimes jump to lower minima points but convergence is difficult to achieve.

To combat this, several approaches are considered:

- Introduce diversity in weights and biases (random weighs between -2 and 2)
- Introduce partial fit steps
- Crossover should be extended to have two offspring instead of one as of now.
- Investigate different mutation algorithms.

Second Iteration

Crossover

In the second iteration of the program, the crossover function was altered to return two offspring instead of only one. This was done by first creating one offspring the same way as in the first iteration, and then using the unused genes to create a second offspring. This means that the second offspring is in a way the "reverse" of the first offspring. This corresponds better to the fact that a k-point crossover method is capable of returning $k * 2$ unique offspring. This also reduces the number of calls of the crossover method.

Mutation

In the first iteration we attempted to use swapping as a mutation operation. For the second iteration we wanted to investigate alternative methods, mainly the following:

- Scramble Mutation
- Inversion Mutation
- Uniform Mutation
- Gaussian Mutation

Mutation Rate

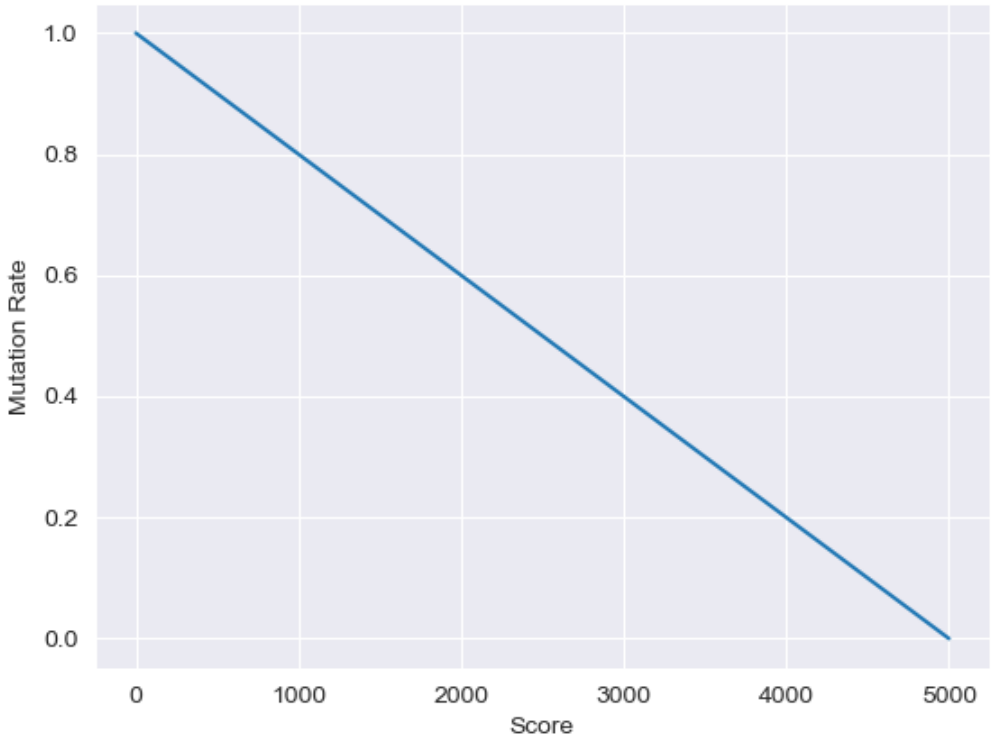

Figure 2: Linear mutation rate with goal set at 5000.

In addition to introducing new mutation methods, the way the mutation rate is utilised was also altered. Instead of passing the mutation rate as a static value throughout the entire run, we change the rate based on how the score is evolving. The mutation rate δ in this iteration follows the function

$$\delta_{S,G} = 1 - \frac{S}{G} \tag{6}$$

Where S is the current score of the single best network and G is the goal we try to achieve. This means that δ decreases linearly as we close in on the maximum score, ensuring that our agents don't mutate outside of the minimum.

Scramble Mutation

The scramble mutation implements a method where a subset of genes is selected from the chromosome and then scrambled.

The scramble mutation method is given by the following code:

$$random.shuffle(el[random1 : random2]) \tag{7}$$

where $random.shuffle$ is the Python $shuffle()$ function which randomly rearranges the given elements and returns them as a list. The elements given to the function is specified by a start and end index, which are randomly selected. The start and end indexes selected define a subset of genes to be shuffled from the agent in question.

Using scramble mutation seems to be the closest contender to the swap mutation implemented in the first iteration. It seems to either reach the desired value, 475, in the earlier generations or stagnate around a value pretty close to that value. In figure 3 below we see a run where the first result occurred and the desired value was reached in the third generation.

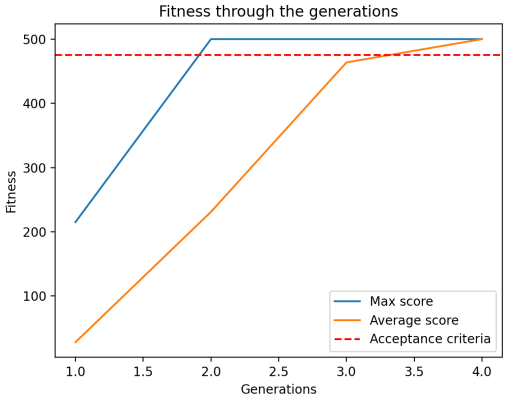

Figure 3: Result of one run - Scramble mutation

Inversion Mutation

The inversion mutation, similar to the scramble mutation, utilizes a method where a subsection of the entire chromosome is selected. All the elements in the selected section are then inverted, as portrayed by the below image.

With this method of mutation, the result varies quite a bit from run to run. In many situations we find that the networks average score stagnates around one area and therefore never reaches the desired value(475). An example of a run where this has occurred is displayed in Figure 4 below.

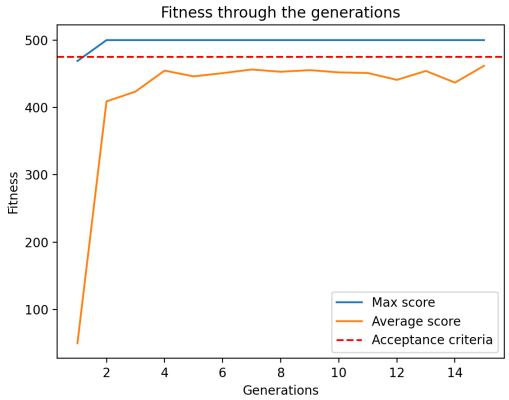

Figure 4: Result of one run - Inverse mutation

Uniform and Gaussian Mutation

By using the uniform mutation, the value of a random gene is replaced with a uniform random value.

The Gaussian method is similar to the Uniform method, although in this method a specific gene is selected and a random Gaussian distributed value is added or subtracted.

For both of these methods the results varies quite a lot, more so when using the uniform mutation. Below we see two separate runs, one utilising uniform mutation and one utilising Gaussian mutation.

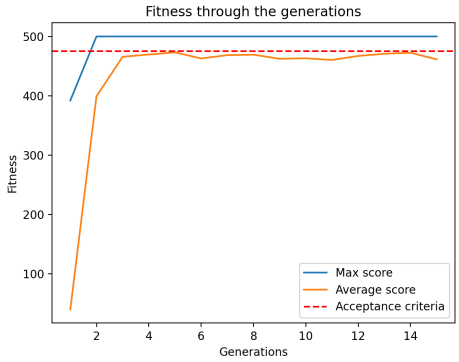

Figure 5: Result of one run - Uniform mutation

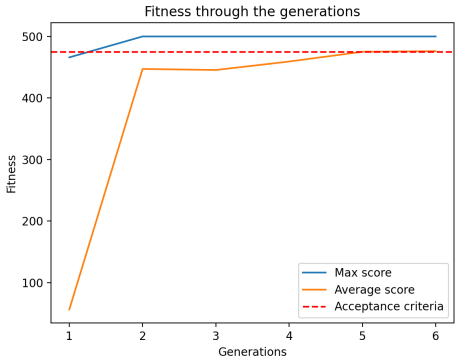

Figure 6: Result of one run - Gaussian mutation

Figure 5 show a typical run when using uniform mutation. Also here the networks average score stagnates and therefore never reaches the desired value. In what area the average stagnates varies, but in this specific run the networks average value is located right below the desired value for all generations. Gaussian mutation on the other hand seems to perform better as it more often reaches 475, which is the desired value. A run where this occurred is visualised in figure 6.

Mutation Results

To discover which mutation method that overall yields the best results, we decided to create a simple simulation. We ran our program a multitude of times with each of the mutation methods, creating a plot displaying the overall average for each generation for each method.

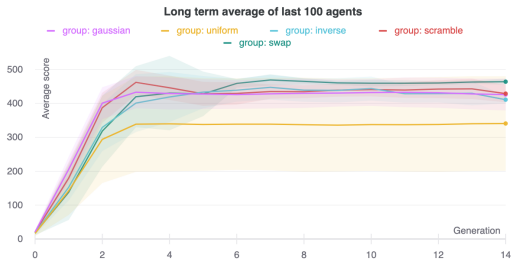

Figure 7: Average reward through generations

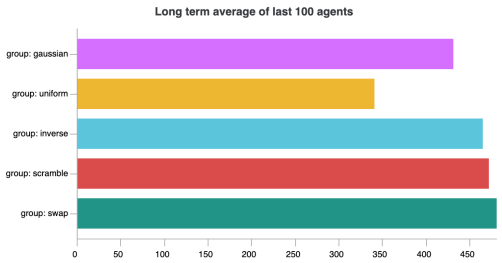

Figure 8: Total average reward

By inspecting the resulting plot, we can quite clearly see that there is a significant variance present. This provides an outcome of averages that are lower than desired. Of all the methods, we see that the swap method has a line that converges towards a value that is just below our lower score limit of 475. Based on this and the fact that the swap method is the only method that resulted in an overall average above 475(figure 8), we decided to venture onward with the swap method as it seemingly gave the most reliable outcomes.

Results

We see from the plots that the simulation is able to find agents that solve the task quite well and we reach an acceptance of 95% before the termination after 15 generations in section . The key takeaway from this iteration is that to find good agents, a high diversity in the initial population is key. Having the mutation rate high in the beginning and decreasing as you is closing in on the global minima ensures (almost certainly) termination. There are still some problems though:

- Termination is not guaranteed

  This seems to be due to the mutation rate function yielding a mutation rate of $\delta = 0$ as soon as the best agent achieves the max score, while the other agents struggle to achieve 95% acceptance. Different δ should be explored by introducing non-linear functions or mutations rates that can increase conditionally on the agents struggling to terminate.

- Successful termination has been achieved for scores as high as 10000. This can be increased further.

  Further investigation of mutation rates should be considered. Other mutation and crossover algorithms or maybe even other models for the agents should also be investigated.

Third Iteration

In the second code iteration we encountered a problem where our network would sometimes stagnate before reaching the requirement. In this case the requirement for success was having the average of the last 100 agents higher than 0.95 * max episode steps (500). In the third iteration we tried several different approaches to combat this.

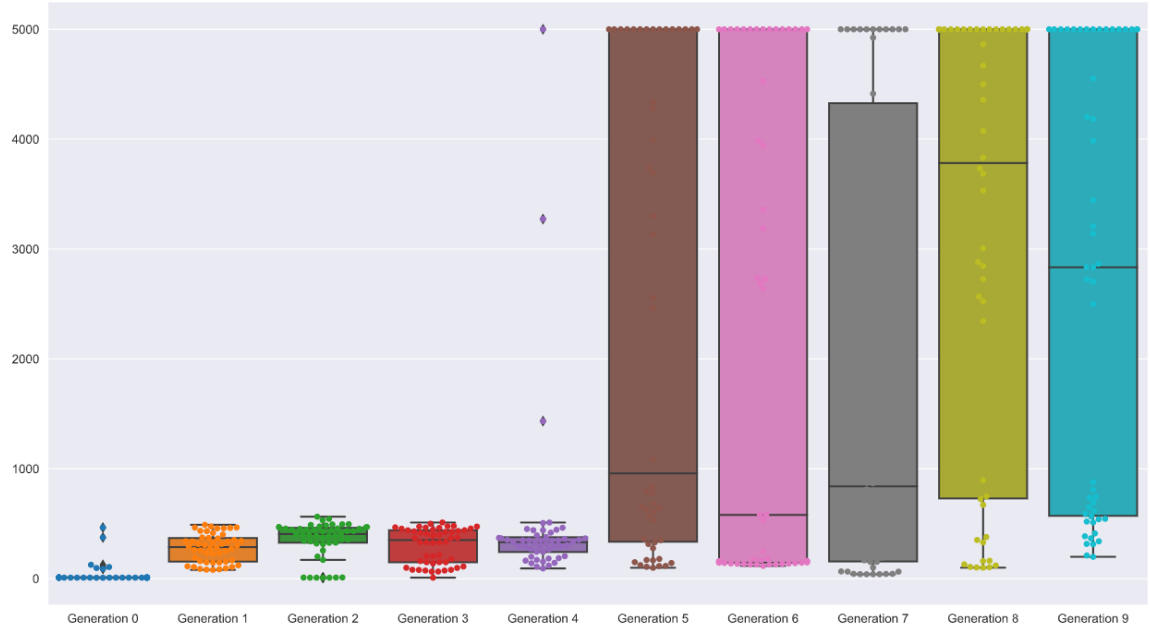

Figure 9: Boxplot of a failed learning sequence

This figure shows a run where the learning stagnated and was cancelled at generation 10. A dot indicates a single agent in the population of the generation. The boxes indicate the variance of the population. We can see that the early generations were unlucky with their mutations and we received no good networks until the fourth generation.

When we have a good network there is a great increase in score once from generation 4 to 5 while the crossover does its job and creates several good networks. However, we can still see that a lot of the agents are being negatively mutated and staying at the bottom of the score scale. The problem which we will solve in this iteration becomes clear in the seventh generation. Notice how there are fewer agents at the very top in the seventh generation than in the sixth. Because we are still mutating at a high rate several of our "perfect" networks have been mutated for the worse, and now perform badly. At the end of this chapter we will discuss a similar plot which shows a successful evolution of our networks.

Crossover

To find a new crossover method to test and implement, an overview of the project was required.

After each generation, the agents undergo a process of selection, reproduction, recombination and mutation

to create the next generation of agents. These steps correspond closely to evolutionary algorithms, a subset of evolutionary computation.

Differential evolution is a comparatively simple variant of an evolutionary algorithm, and therefore also one of the more popular algorithms. The goal of the algorithm and our program is to find a solution m for which $f(m) \leq f(p)$ in the search space, and thus finding m as the global minima.

The differential evolution algorithm first defines the crossover probability CR and the differential weight F. These parameters are set to their standard values 0.9 and 0.8, respectively. For each agent in the population, the algorithm chooses three other agents at random. For each gene in the selected agent's chromosomes, the algorithm either passes the gene to the offspring or uses a combination of the other agents to create a new gene. This combination follows the function $y_i = a_i + F * (b_i - c_i)$, where the y_i is the new gene created, and a_i, b_i, c_i corresponds to the genes of the other agents. The new agent, created from the genes of the differential evolution algorithm, replaces the chosen agent if it performs better. [2]

This method did not provide us with the desired results. Instead, most of the generations after the initial generation performed worse. As the method under-performed compared to the two point crossover, we decided on finding a new crossover method.

A simplified differential evolution algorithm was found in a comment to a post with the same issue as us. [3] This simplified differential evolution algorithm works on a chromosome level instead of a gene level. The chromosomes of the offspring equals the chromosomes of the fittest parent, added with the difference in the chromosomes of the two parents multiplied with a list of uniformly distributed random values between 0 and 1. In other words: $y = a + \mathcal{U}(0,1) * (b - a)$, where a is the chromosomes of the fittest parent and b is the chromosomes of the second fittest parent.

This yielded much better crossover breeding compared to the classical differential evolution algorithm, and a somewhat better breeding compared to the two point crossover method from iteration one.

Mutation

A problem in the second iteration was encountered when using the scaling mutation rate mentioned in the previous chapter.

$$\delta_{S,G} = 1 - \frac{S}{G} \tag{8}$$

This mutation rate helped us reach the goal in fewer generations than before, but might have introduced the stagnation problem. We quickly realised that once a single network´s best agent has a score equal to the maximum value the mutation rate will fall to 0. This results in learning problems for the rest of the networks, as these are no longer able to mutate.

This problem should be solvable by simply setting a minimum value for the mutation rate. Even though we have a network which contains agents which has achieved the maximum score, the networks should be able to continue mutating and improving. However, when testing, the stagnation is still in place. We realised that this approach introduced a new problem. When most of our networks are above the threshold (0.95 * max episode steps) some of these are still being mutated. As the mutation is now still running, some of our good networks might be getting mutated for the worse.

There is no reason to mutate the networks that are good enough. Therefore, we decided to implement a new method of applying mutation. Instead of mutating every single network, we now only mutate the networks that are below a new threshold. Since we are striving for the average between the last 100 agents to be above 0.95 * max episode steps, we decided that an acceptance threshold of 0.975 * max episode steps would be

suitable. We still run crossover over all the networks, but for mutation we find all the networks which do not satisfy our new threshold and mutate them. This provided more stability. Now our networks will be randomly mutated until they are "good enough" and from there on they will only be affected by crossover. At the same time, the networks that are "not good enough" are being mutated so they can acquire the correct genes for a successful simulation.

Ultimately, we were able to greatly reduce the chance of stagnation by combining our previous efforts. When our average gets closer to the goal we are sill mutating a large part of the agents under our threshold. For our final improvement we reemployed the scaling mutation rate from the above equation. This time we used the average score of a population to calculate the mutation rate, rather than the highest score like we did before. This lets us aggressively mutate agents until our average converges toward the goal score. Finally, we let a small amount of mutations and the crossover breeding bring the last networks over the success threshold. After this implementation less than 4% of our tests end up stagnating.

Results

Recall the similar plot we discussed at the start of this chapter, and compare it with the following plot.

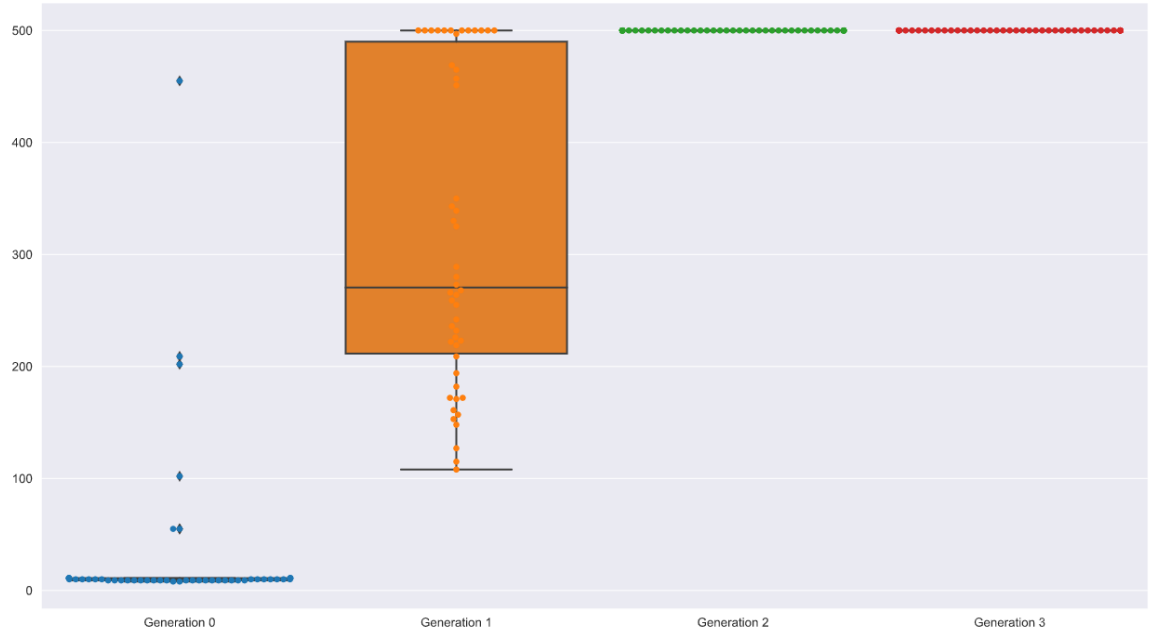

Figure 10: Boxplot of a successful learning sequence

After implementing better crossover and mutation we also see better results from the box plots. From the plot above we can see that even though we have 0 max score agents in the first generation we are able to create several "perfect" agents already in the second generation. Due to the new selective mutation none of the agents that are accepted are being mutated for the worse and we are able to achieve 100% of the agents at max score already in the third generation. Now we can wait for the average of the last 100 agents to catch up and finish the task in only four generations.

Additional materials

Some of our endeavours did not result in the outcome that was envisioned by the project group. These materials are described in this section.

Flattening Before Mutating

The mutation methods we are using in the final product work by altering randomly selected indices of the rows of chromosomes. These rows originate from the weight and bias matrices, and we wanted to see if we could improve our results. We wanted to do this by instead of mutating the rows, mutate the elements inside randomly selected rows. We did this by using the *numpy.ravel* function to flatten the matrices, allowing us to perform regular array operations while mutating. We then reshaped these arrays to their original shapes before returning the outcome of the mutation. What we found by doing this what that we either got terrible outcomes as a result of doing "over-altering" of the chromosomes, or outcomes that were not much different than those without any mutations at all. Although this did not give us any viable outcomes, it proved to be an excellent learning experience, and gave us a deeper understanding of the mutation process.

Trying Different Mutation Methods

We ended up trying a series of different mutation methods, i.e. the $swap$, $inverse$, $scramble$, $uniform$ and *gaussian* methods. The first method we tried to use was the swap method, which from the start gave decent results. However we wanted to see if another method could provide better results and lower variances. We modified our mutation function to accept mutation-method as an input argument, and spent a while implementing the different methods. As outlined in section Mutation Results, this was yet another venture that gave less than desirable results. The swap method came out on top, ensuring that we were right to use it to begin with. Even though we did not end up using any of the methods we implemented, this was a good learning experience, and it gave us a better grasp of the importance of selecting a good mutation algorithm.

Single Point Crossover

The simplest and most used reproduction method in genetic algorithms is the single point crossover method shown in figure 11. In this method a single crossover point on the parent gene is selected. All data beyond that point on the gene is swapped between the two parent genes. The single point crossover function created in this project breeds new offsprings from the two fittest agents from the selection method.

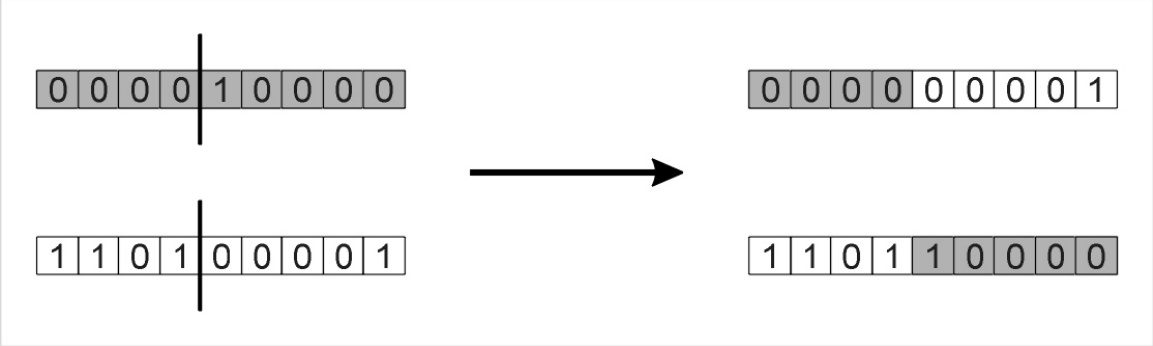

Figure 11: Single point crossover

Conclusion

Based on the project description all functionality has been implemented and works as expected. By implementing additional properties such a dynamic mutation rate, we managed to generate results which we see as satisfactory.

Network performance

At the beginning of the project we observed a high variance in the network performance. We introduced different static mutation rates(0.001, 0.05, 0.1),and found that this did not improve our results in form of stability or accuracy. Throughout the project we introduced multiple different mutation methods as well as a dynamic mutation rate. We found it somewhat difficult to get clear results from comparing the methods as a high level of variance was still an issue. Going in to the last iteration we made additional improvements relevant to the mutation rate, and experience a large improvements in the network performance.

As of now we experience a much more stable performance across simulations than in previous iterations. The figure below visualises the performance of 50 consecutive network simulations. As we can see, most of the networks reach the desired mean reward and succeeds within 15 generations.

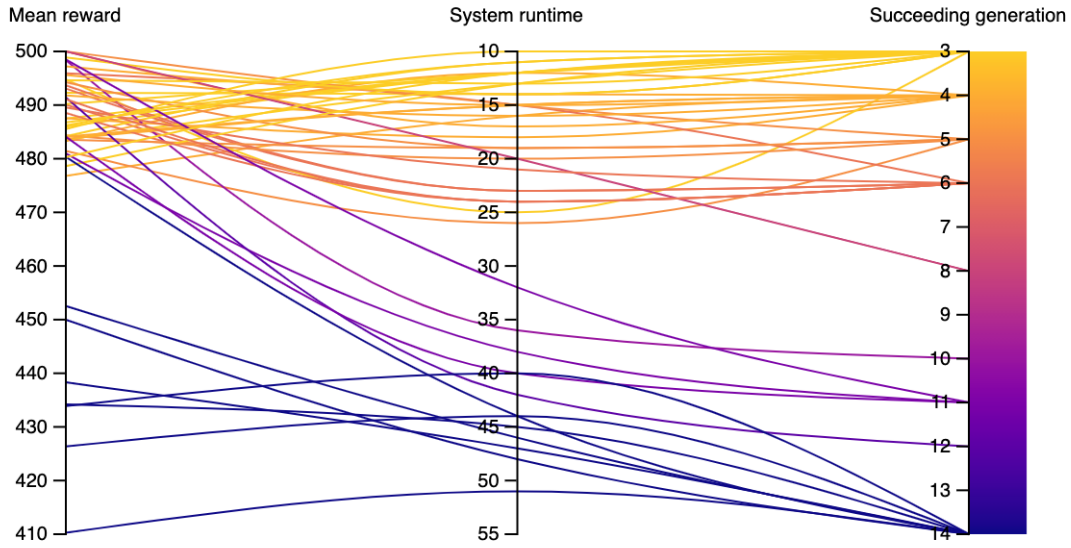

Figure 12: Visualisation of several simulation runs with $N = 50$ simulations. The simulation almost certainly succeeds within the maximum number of generations being 15. With an acceptance ratio of 95%, eight simulation runs did not achieve a successful mean reward.

Without having any real expectations to compare our results with, it was difficult to say how well we performed. Luckily, OpenAI has an open leader board [6] for the Cartpole-v0 environment, which is similar to the one we used. When testing with the environment and the restrictions specified for the leaderboard we consistently achieved results which placed us in the top 10 contributions. With our limited experience in this field, this is a result we can feel proud of.

[6] Cartpole-v0 leaderboard

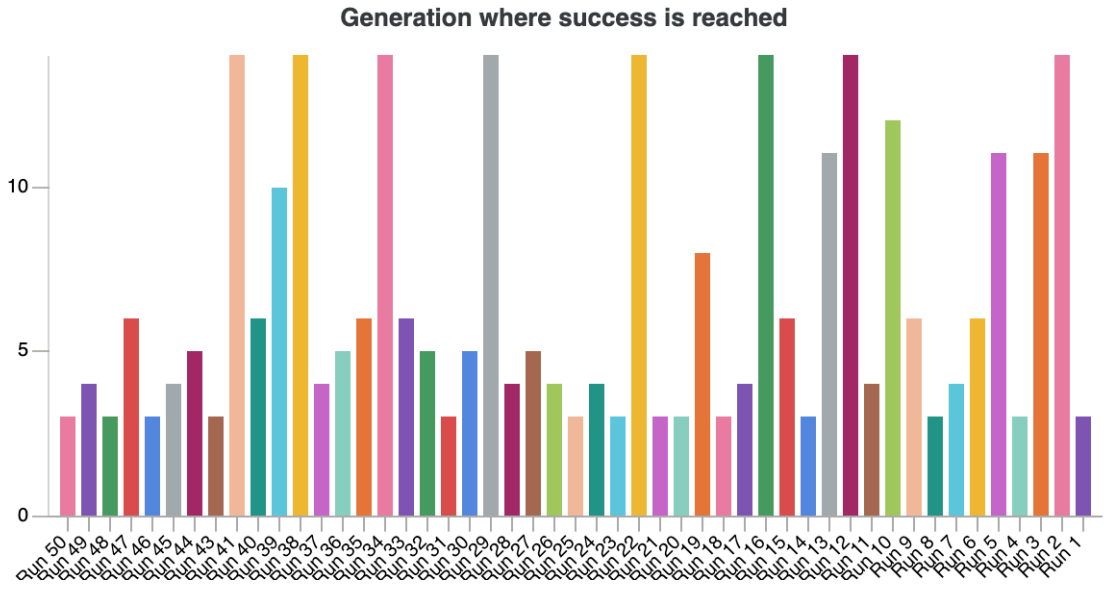

Figure 13: Visualisation of generation of success for several simulation runs with $N = 50$ simulations. The simulation almost certainly succeeds within the maximum number of generations being 15. In average the network takes 6.29 generations in order to reach success.

Finding good agents

One of our main findings have been that by introducing very diverse agents, very well performing agents will occur and will be selected for breeding. This was done by introducing a high mutation rate. Intuitively the well performing agents will then be selected for breeding and their genetic information passed on to the next generation resulting in a well performing population. While this is partially the case, the high mutation rate caused the population to never converge on the global minima.

By introducing a mutation rate function $\delta_{S,G}$ instead of a constant mutation rate, high diversity in the beginning will occur and when optimal agents are discovered the mutation rate decreases, ensuring that the population is able to search for the global minimum.

Selective mutation

We encountered problems with not reaching successful termination due to the well performing agents mutating out of the global minima or no mutation rate at all that sometimes ended with well performing agents not occuring. Selective mutation is a successful way to ensure that the population reaches high scores, though it violates some of the principles of biological evolution that our algorithms are based on,

Teamwork

When working with projects such as this, it is always a challenge to fuse the widely varying schedules of six different individuals into an organised and productive group. As mentioned in the introduction, we used GitHubs Issues and Pull Requests workflow to support our teamwork efforts for this project. In combination

with a workspace in Slack, we have not only been able to work together, physically, at school, but also to be productive remotely. This solution and combination of great teammates let us finish the project well within the time limits and provided us a great learning experience on the subject of teamwork.

References

[1] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[2] David Mayor. Differential evolution – an easy and efficient evolutionary algorithm for model optimisation. https://www.sciencedirect.com/science/article/pii/S0308521X04000745, 2005.

[3] Glen Rodriguez. What is the most suitable crossover method to train ann using ga? https://www.researchgate.net/post/What_is_the_most_suitable_crossover_method_to_Train_ANN_using_GA, 2012.