Investigating the effects of evolutionary parameters in the NeuroEvolution of Augmenting Topologies Algorithm

Research Question: To what extent does the compatibility threshold and species stagnation in the NEAT Genetic Algorithm influence its performance in terms of convergence speed in balancing a Cart-Pole Control System?

Word Count: 3942

Student code: [redacted]

Contents

1 Introduction

The rate of advancements in computing have only skyrocketed in the past 50 years ("50 Years of Computer Science"). More recently, an emerging technology has taken center stage in the rapid development of mankind's capabilities throughout the last few decades. Machine learning, a subfield of Artificial Intelligence involves teaching computers to learn rather than follow explicit instructions. Computers, which were once limited by the ingenuity of programmers, only capable of executing step-by-step meticulously engineered algorithms, are now capable of adapting and self improving, far surpassing their creators in ways never thought possible. One powerful approach to machine learning is by taking inspiration from biology and devising evolutionary algorithms. These algorithms simulate genetic evolution to evolve neural networks over time, allowing for a population of agents to rapidly and efficiently learn complex tasks.

The specific algorithm that I'll explore is NEAT (Neuroevolution of augmenting topologies) due to its widespread popularity, which will in turn increase the utility of this research. Given that genetic algorithms essentially replicate aspects of biological evolution, there are several parameters that dictate how the evolution is computed.

This extended essay seeks to evaluate how changes in two evolutionary parameters in a specific model affect the performance of this algorithm, which will be measured in the amount of generations required to meet a success criterion. This investigation will simulate a physics-based scenario commonly used to assess the performance of a machine learning system, more specifically, a cart-pole system, which will be balanced by the trained models. Therefore, this research question proposed is ***"To what extent does the compatibility threshold and species stagnation in the NEAT Genetic Algorithm influence its performance in terms of convergence speed in balancing a Cart-Pole Control System?"*** Following this exploration, the results (amount of generational iterations required for each variation of the two parameters) will be analyzed and evaluated to propose optimal values of the studied parameters.

2 Background Information

2.1 Neural Networks

Neural networks are a key concept in the field of AI, they represent the pillars on which all machine learning models are built off of. Inspired by the architecture of the human brain, neural networks consist of interconnected nodes, which act as artificial neurons that collectively work to identify patterns within data. These neurons are organized into layers, an input layer, one or more hidden layers, and an output layer, which presents the outputs from the network when the given inputs are processed. The connections between these neurons are assigned a weight that modulates the signal traveling through it. These networks learn by training on data that contains inputs and their expected outputs when run through the network. Training primarily operates through two key processes, forward propagation, and backpropagation (Machine Learning in Plain English).

Structure of a feedforward neural network

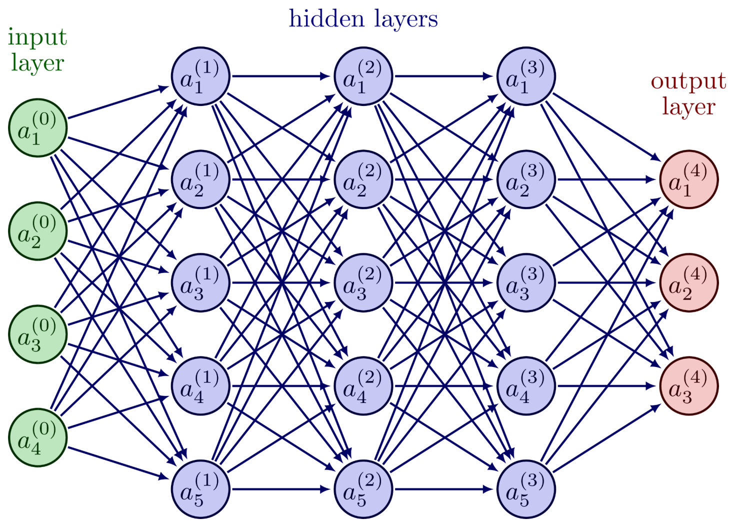


Fig 1. (Neutelings)

In forward propagation, (essentially running data through the network), data moves through the network from the input layer to the output layer, producing a prediction (ajitjaokar).

First, the neurons in the hidden layers receive the weighted sum of all the results of the connected neurons in the previous layer, plus a bias term.

For example, with an input vector $x = [x_1, x_2, . . . , x_n]$, where each element represents a feature of the input data, (for example the velocity of a car in a driving dataset), the output of a hidden neuron h_j can be calculated as:

$$h_j = a(\sum_{i=1}^{n} \omega_{jk} h_j + b_j)$$

Where ω_{jk} represents the weight between the input x_i and the hidden neuron h_j. b_j is the bias for the hidden neuron h_j, and $a(\bullet)$ is the activation function, which essentially removes the completely linearity that arises from this multiplication and addition of weights and biases. In this study, the only activation function used is the hyperbolic tangent function:

$$a(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

Fundamentally, the "goal" of the machine learning algorithm is to minimize the error between the result vector produced when inputs are forward-propagated through the function, and the expected results (Pargaonkar). This is usually achieved through back propagation, a logical algorithm to comprehend, but achieving a mathematically efficient solution is rigorous and for brevity's sake, will be omitted from this work. However for clarity, the process consists of applying the chain rule of differentiation to compute how each weight affects the overall error, and then a partial derivative of the loss with respect to each weight is derived (Jafari)(Saravanan). Then, using gradient descent (Donges), (a simple algorithm to approach the minimum of a function by subtracting a starting input by the derivative of that inputted value), the weight and biases are subtracted by their partial derivatives:

$$\omega_{ij} := \omega_{ij} - \eta \frac{\partial L}{\partial \omega_{ij}}$$

$$b_j := b_j - \eta \frac{\partial L}{\partial b_j}$$

Where η is the learning rate, a hyperparameter that controls the step size of the weight and bias adjustments.

Theoretically, a large enough neural network of this type could learn anything (Felbert), however, due to practical limitations in computing power, raw feed-forward neural networks are rarely used for complex tasks, and instead more ingenious solutions are devised that shortcut the processing power required to train a functional neural network.

2.2 Genetic Algorithms

One such solution is through genetic algorithms, which offer a beautiful alternative to the traditional, mathematical training methods used conventionally with neural networks. Instead, genetic algorithms learn by taking inspiration from natural selection. They operate by evolving a population of candidate solutions, (many neural networks with different random weight configurations), over many generations (Warangal) ("Genetic Algorithm in Machine Learning")**.** The key difference being that instead of directly tweaking the weights and biases of a network to minimize error, genetic algorithms evolve solutions by selecting and combining / reproducing the best-performing individuals, using biological processes like crossover and mutations.

2.2.1 Fitness Function

Likely the most important component for genetic algorithms is the Fitness Function. This is an abstract function that varies for every application, but essentially measures the performance of the agents being trained and scores them accordingly, ultimately leading to the highest scoring individuals to reproduce and advance to the next generation.

2.3 Overview of NEAT

The genetic algorithm explored in this essay is NEAT. Devised in 2002, "NeuroEvolution of Augmenting Topologies" is an advanced approach in evolutionary algorithm that not only evolves the weights and biases of its agent's neural networks, as with a typical genetic algorithm, but also allows for simulated evolution to alter and mutate the structure / shape of the networks (Stanley and Miikkulainen). Allowing this simultaneous optimization of both the weights and the network topology itself overcomes a common issue with standard machine learning. In a standard machine learning model, the desired amount of hidden layers and the amount of neurons per layer has to be chosen by random estimation or through trial and error attempts, making it time consuming to optimize the shape of a neural network to the bare minimum complexity it can be while still solving the problem. Furthermore, it's very easy to "overfit" or "underfit" the network, which signifies that the network either doesn't have enough neurons to meet the required solution, or the network is unnecessarily complex, which can cause it to try to perfectly fit the training data instead of generalizing and discovering patterns, leading to a noisy solution (Nautiyal).

Furthermore, NEAT is inherently stochastic because it relies on random processes at several stages of its genetic algorithm cycle. From the initial configuration of the population of neural networks, to the probabilistic mutations and crossovers that drive the evolution of topologies and weights. This randomness ensures that many diverse solutions are explored, helping the algorithm avoid getting stuck in local minimum.

Example of Getting Stuck at a Local Minimum in a Two-Parameter Loss Surface

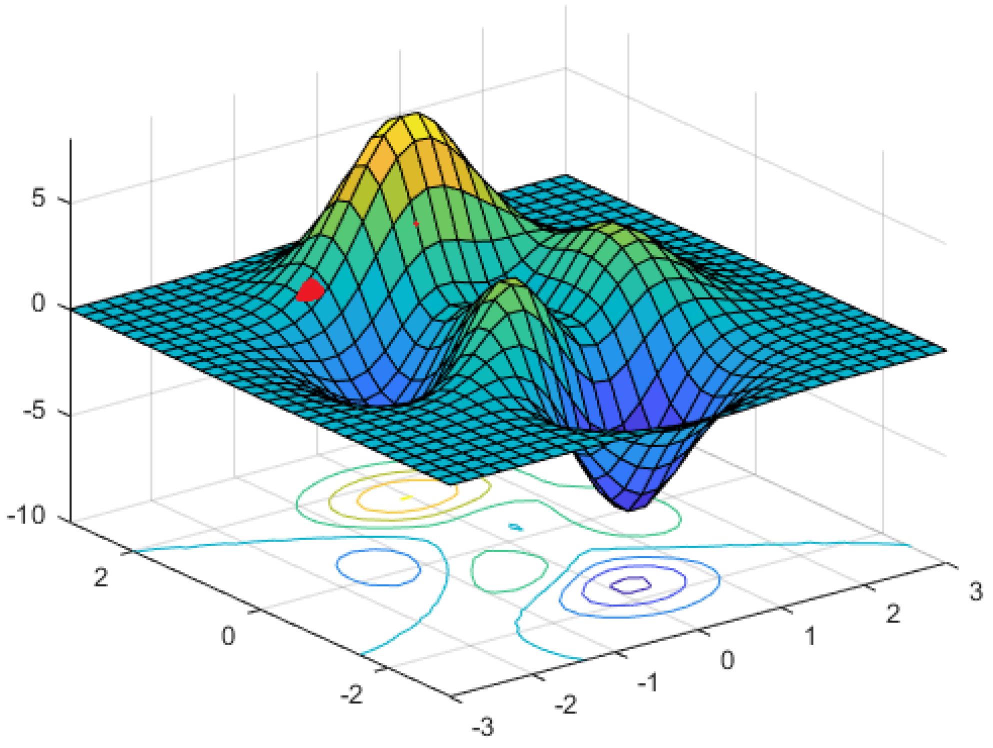


Fig 2. (Kelsey)

2.4 Studied NEAT Parameters

The overarching focus of this study is themed around the evolutionary parameters of these genetic algorithms, as they stand to be the most fascinating and insightful variables. In their very essence, these are the aspects which we cannot simulate, the logic that is too complex to mirror reality's counterpart, and hence genetic algorithms must resort to techniques that try to replicate real results.

2.4.1 Compatibility Threshold in NEAT

One such parameter is the Compatibility Threshold, which is a parameter which is critical to classifying networks into species based on their similarity. This value is used to determine whether two neural networks belong to the same species, with their similarity being calculated using a "compatibility distance" metric (Stanley and Miikkulainen). This distance d is defined as

$$d = c_1 \frac{E}{N} + c_2 \frac{D}{N} + c_3 \overline{W}$$

Where E represents the number of excess genes between two networks, D represents the number of disjoint genes (amount of genes mismatched between the two networks). $\overline{W}$ is the average weight difference of matching genes, N is the normalizing multiplier for the total number of genes (used to scale the distance for larger networks). c_1, c_2, c_3 are preset coefficients that respectively weight the importance of excess genes, disjoint genes, and weight differences (Stanley and Miikkulainen).

The threshold controls what the compatibility distance must be greater than, to consider two networks of different species, allowing NEAT to speciate. This is crucial for maintaining a balance between diversity and convergence within the population. By grouping the population's networks into species, NEAT ensures that new structures, brought upon by recent mutations or crossovers, are protected from being outcompeted by well established structures (Stanley and Miikkulainen). Thus this protection allows new innovations to mature and improve, increasing their likelihood of succeeding. Overall, a good compatibility threshold maintains a balance between diversity and convergence within the population.

2.4.2 Species Stagnation in NEAT

The second parameter analyzed in this essay is the species stagnation. The species stagnation is a value that determines how long a stagnant species can survive. A species is considered stagnant when its maximum fitness does not improve after the amount of generations depicted with the species stagnation parameter (Stanley and Miikkulainen). In essence, it dictates how lenient the algorithm is willing to be with non-improving species.

When a species doesn't improve after that amount of generations it becomes considered a stagnant species, meaning that members of that species can no longer reproduce (Stanley and Miikkulainen).

2.5 Cart-Pole System as a Benchmark Problem

The cart-pole system is a classical control problem wherein a cart moves along a horizontal axis, with a pendulum / pole attached. The aim of the problem is to maneuver the cart in a manner that keeps the pole balanced above it. This scenario is frequently used to evaluate the performance of machine learning algorithms (Weber). This is due to many reasons, but some include:

- **Continuous feedback:** The cart-pole system provides real time feedback (such as the pole's angular velocity, and the cart's position, etc…). This mirrors many real-world applications of machine learning, such as self-driving cars, wherein the system is required to make decisions in a dynamic environment.
- **Exploration vs Exploitation:** More specific to this essay, the cart-pole system poses as a great medium to evaluate how the system is performing biologically, that being how varied the species are, and how effective the algorithm is at fully utilizing its resources, (ie: not wasting it's population on stagnant strategies). This directly ties to the aim of this study in exploring the effects of the biological parameters on NEAT's performance.

3 Methodology

3.1 Experimental Setup

Firstly, this essay relies purely on primary experimental data due to the limited amount of data in other studies investigating the effects of NEAT parameters. The only external resources used in the data collection process are the libraries discussed in **3.1.2**.

3.1.1 System information

The entire codebase was written in Python version 3.12.2, this is both because of its excellent NEAT implementation present in the library "*neat-python*", and my familiarity with the language and the libraries used.

The hardware configuration used to run the data collection scripts is a Windows 11 computer with an RTX 2060 Super graphics card, Ryzen 3700x CPU with 16GB of RAM.

3.1.2 Library Dependencies

Library Used	Rationale
neat-python (external)	NEAT python implementation; code for the genetic algorithm used, handles population instantiation, gene crossover, mutations, etc…

pygame (external)	Used for visualizing evolution in development
pyperclip (external)	Used to run automated data collection script
math (built-in)	Used for cart-pole physics calculations
os (built-in)	Used to access and edit "*config.txt*" file wherein the investigated parameters are provided to NEAT.

3.2 Simulation

3.2.1 Cart-Pole Simulation Environment

The cart-pole scenario was implemented by first analyzing the forces acting on a standard pendulum, then by applying Newton's laws of torque and inertia, a function of the angular acceleration of the pendulum given a the acceleration of the cart can be found:

("A Pendulum with the Moving Pivot")(Weber)(Green)(Florian)(Matthew Kelly)

Tangential force due to gravity:

$$F_{tangential} = m \cdot g \cdot sin(\theta)$$

Angular velocity due to gravity:

$$\alpha = \frac{-g}{L} \cdot sin(\theta)$$

Additional angular velocity due to cart motion:

$$\alpha_{cart} = \frac{k}{L} \cdot cos(\theta)$$

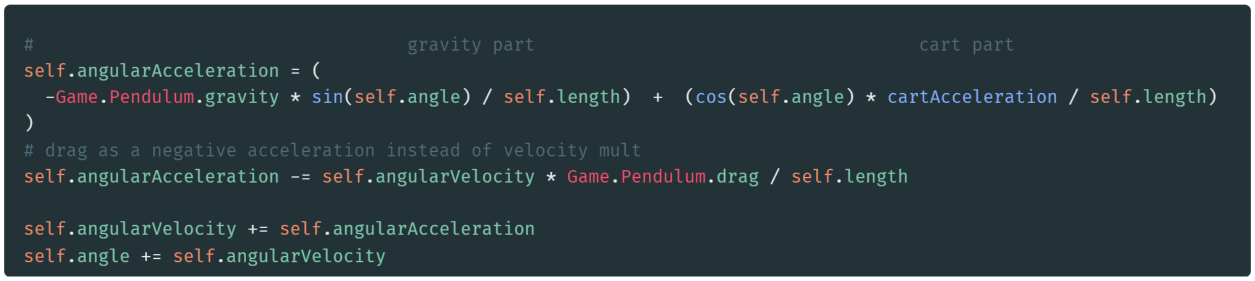
Total angular acceleration:

$$\alpha_{total} = \frac{-g}{L} \cdot sin(\theta) + \frac{k}{L} \cdot cos(\theta)$$

Where:

- **g:** Acceleration value due to gravity.
- **m:** Mass of the pendulum bob, cancels out in the final equation, thus mass doesn't affect the pendulum movement.
- **L:** Length of pendulum, (distance from cart to the pendulum bob).
- **θ:** Angle of the pendulum from the vertical.
- **α:** Angular acceleration of the pendulum.
- **$F_{tangential}$:** Tangential force acting on the pendulum.
- **k:** Left-right acceleration of the cart.

These equations can be applied programmably to create this:



```
#                                   gravity part                                   cart part
self.angularAcceleration = (
  -Game.Pendulum.gravity * sin(self.angle) / self.length)  +  (cos(self.angle) * cartAcceleration / self.length)
)
# drag as a negative acceleration instead of velocity mult
self.angularAcceleration -= self.angularVelocity * Game.Pendulum.drag / self.length

self.angularVelocity += self.angularAcceleration
self.angle += self.angularVelocity
```

Fig. 3: Implementation of cart pole mechanics

3.3 NEAT Algorithm Implementation Details

As for the evolutionary algorithm, this essay only modified the studied parameters, (compatibility threshold and species stagnation). The majority of parameters were left untouched in their default state to keep these results as consistent with other studies as possible. However, the key components of NEAT were set as such:

3.3.1 Population size

A population size of 100 agents was chosen somewhat arbitrarily, an increase in agents leads to an increase in diversity, which allows for a greater exploration of possible solutions (Stanley and Miikkulainen), but leads to decreased performance. However, the diversity effects are somewhat linear and thus any value still produces practical results as the changes of population size will be proportional. Thus I settled on as high a population as possible without severely impacting the performance of the data-collection.

3.3.2 Inputs and Outputs

Naturally with control system problems, the means of control and sensing are extremely consequential. In this study, I opted for a left-right binary action space, meaning that the agents have two possible inputs, to accelerate the cart to the left, or to the right. This decreases the amount of control they have over the cart's movement, as they aren't able to accelerate at a wanted velocity, but in genetic algorithms, allowing for the least control required to still complete the task is key in maintaining performance, as the cost of running these simulations can have a significant toll in processing power.

As for inputs, a similarly sparing approach is a good measure when managing performance, but regardless, should the agents not have enough information, they won't be capable of achieving a solution. Thus the agent's inputs are:

- Pendulum angle
- Pendulum angular velocity
- Cart x position

These inputs, while limiting, still allow for a successful solution to arise ("Cart Pole").

3.3.3 Common issues

A frequent problem with artificial intelligence in control systems like these is that the agents tend to exploit certain conditions of their environment and become overly specific to starting conditions. For example, to prevent the agents from not moving at all, the pole cannot start at a perfectly vertical orientation, as this wouldn't need movement to remain balanced. Instead, when initially testing the cart-pole environment, I set the pole's starting angle to be 90.001 degrees, this led to agents relying on the fact that the pole always falls to the right. To prevent this, the pole will alternate between a right and left leaning tilt between generations.

3.4 Data Collection

To quantify the performance of the algorithm, I will measure the amount of generations required to achieve a perfect agent, with a range of compatibility thresholds and species stagnation values. An agent is classified as perfect when it is capable of maintaining the pole in balance (above the cart) for two generations in a row (to ensure success with a right and left side falling pole. See 3.3.3). This metric intentionally allows some flexibility by defining perfection leniently, as measuring the learning speed to complete the task is typically more insightful than slowly reaching a great result. The data collection process involves a multisample approach, wherein for each value of species stagnation or compatibility threshold tested, there will be 10 samples collected. A sample is the amount of generations needed to create a perfect agent in a simulation of 50 generations of the population. If a sample yields no perfect agent after 50 generations, it is considered failed. Should more than 5 out of 10 samples fail, the data collection for that tested value should be viewed tentatively. These values were all chosen as compromises between ensuring reliable data quality and maintaining feasibility within the constraints of time and computational resources.

To facilitate the data collection, I've written a script that automates the data collection process by running the evolution process, gathering results, and modifying the studied parameters, then this is repeated until the necessary data is collected. This can be found in Appendix D.

3.4.1 Compatibility Threshold Configuration

For compatibility threshold, the range of values explored will be from 0.5 to 7.5 at a step value of 0.2. As discussed earlier, a lower threshold means that two networks have to be really similar to be classified as the same species. Therefore a low threshold leads to more species, which when below ~0.5 classifies around 100 species, which becomes extremely intensive to run.

3.4.2 Species Stagnation Configuration

The species stagnation metric doesn't necessarily impact the program's performance so the range can be greater. A range from 3 to 30 will be explored.

3.4.3 Cost of Compromises

It is important to note that these decisions made to alleviate the processing power and time needed in the data collection process should be recognized and acknowledged. All together, these choices contribute to significant impacts on the collected data which must be addressed via deep analysis.

4 Experimental Results

Due to the extensive amount of raw data from the sample size and tested variable ranges, the collected data is presented in tables in Appendix B. As discussed previously, simulations that don't For the sake of brevity, I'll only present certain pieces of the collected data throughout the analysis, as needed.

4.1 Results for Different Compatibility Thresholds

This graph shows the mean generations needed of the 10 samples calculated for each of the 36 compatibility threshold values (0.5-7.3). Figure 2 further includes the raw samples.

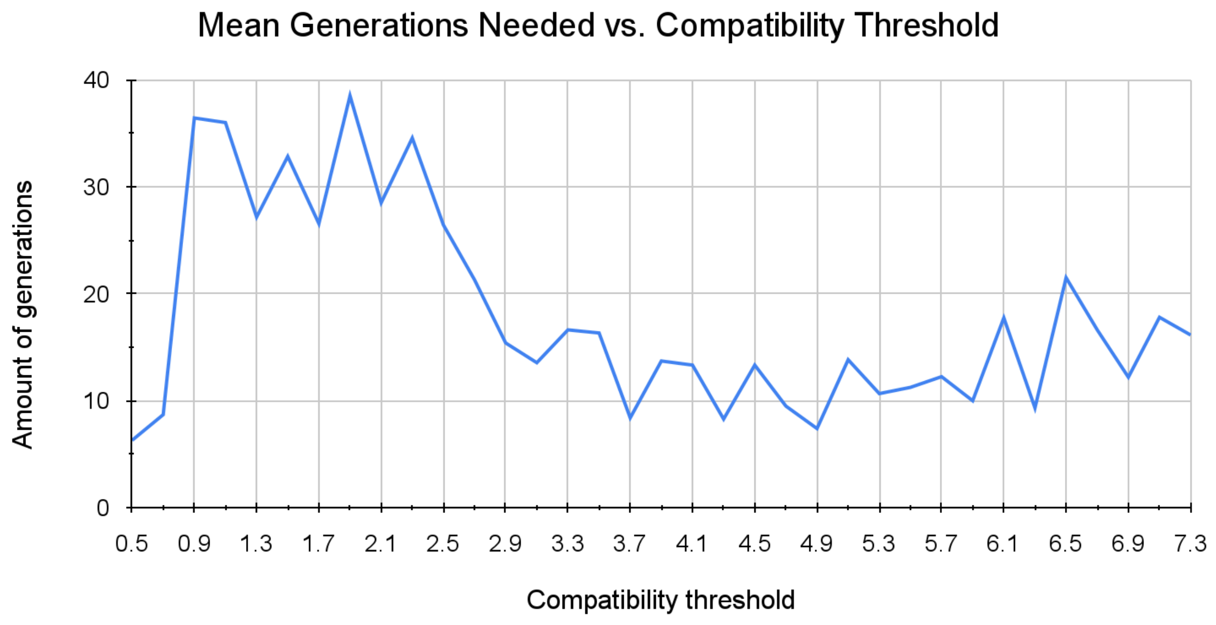


Figure 4

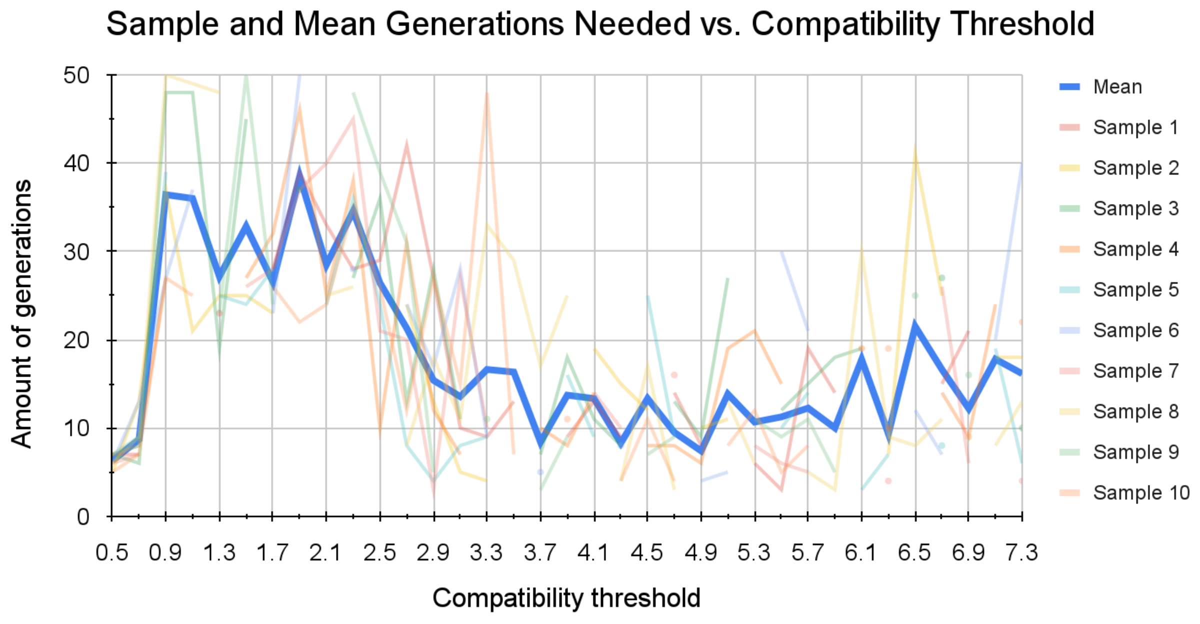


Figure 5

4.2 Results for Different Species Stagnation Thresholds

Similarly, these graphs show mean and samples of how many generations were needed for a perfect agent to evolve with species stagnation values from 3 to 49.

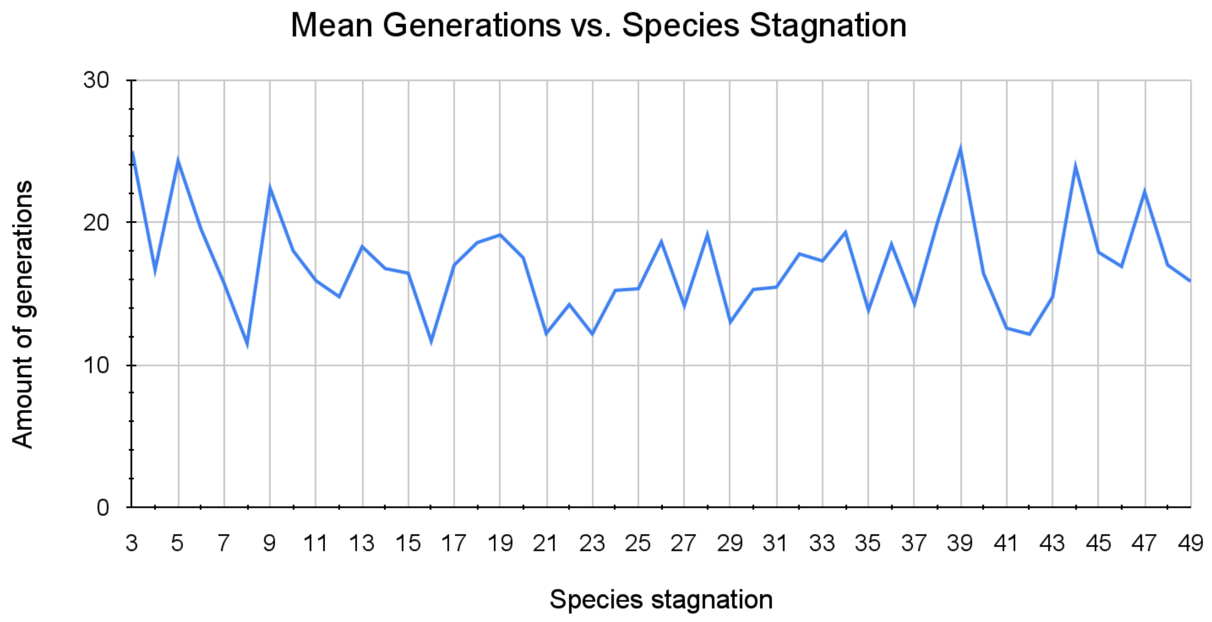


Figure 6

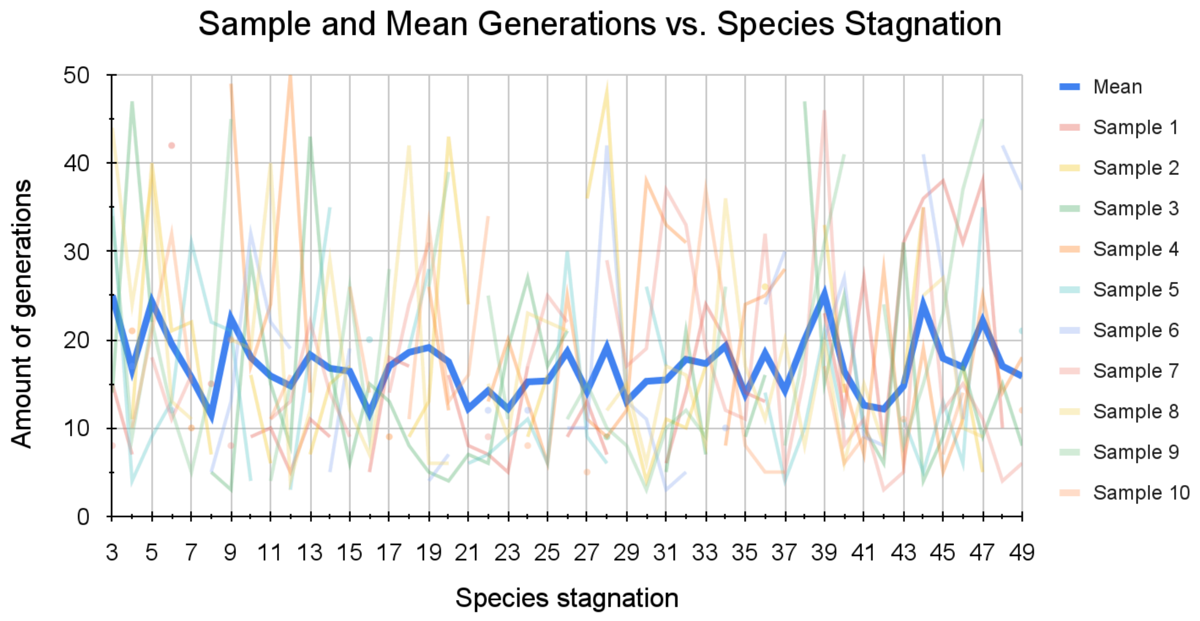


Figure 7

5 Data Analysis

5.1 Influence of Compatibility Threshold

5.1.1 Accounting for inconsistencies

It is evident through *figure 4* and *figure 5* that there is a substantial correlation between compatibility threshold and the performance of the NEAT algorithm, albeit non-linear. *Figure 5* displays that the individual samples are highly inconsistent, which arises from the previously discussed stochastic nature of the NEAT algorithm. This is clarified through the standard deviation of these samples:

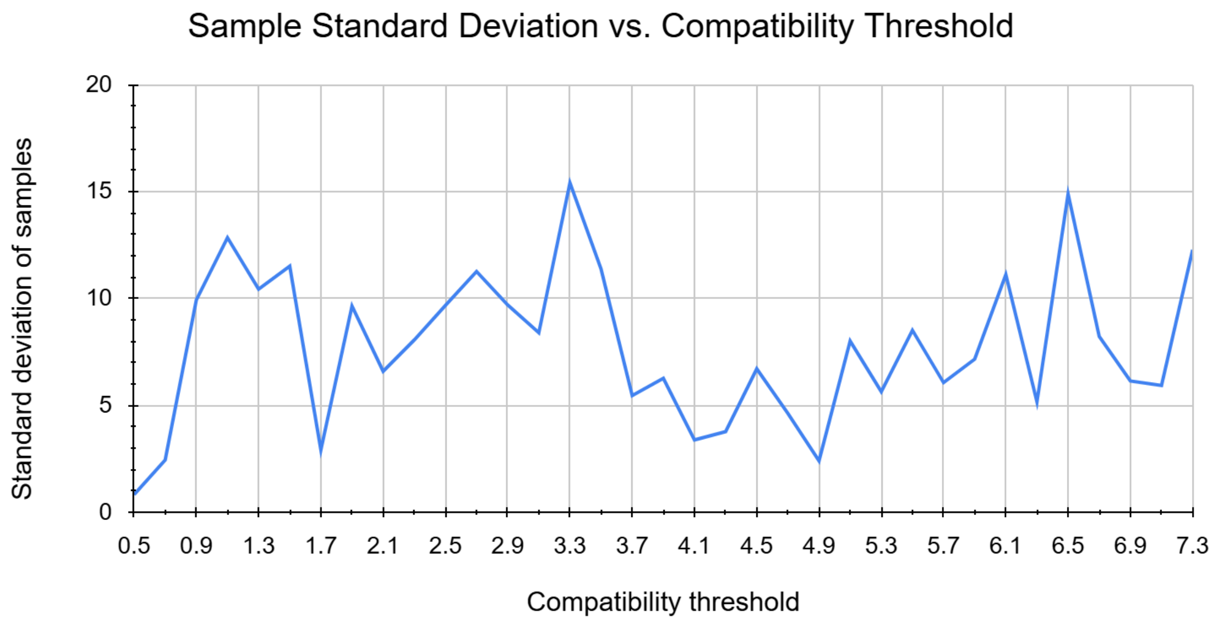


Figure 8: *Standard deviation of all 10 samples per threshold*

As seen in *figure 8*, the disparities between samples are significant with standard deviation climbing past 15 generations at a compatibility threshold of 3.3. While this does suggest that the large drop in "generations needed" seen in *figure 1* (from thresholds

2.3 to 3.1) may have been exaggerated due to randomness, the standard deviation remains reasonably constant nevertheless. Furthermore, the standard deviation of the sample standard deviations per threshold is ~3.57, suggesting that only a uniform randomness is present in the data, reinforcing the claim that this inconsistency is simply a consequence of NEAT's use of randomness. This can be better visualized with an upper and lower bound on the mean:

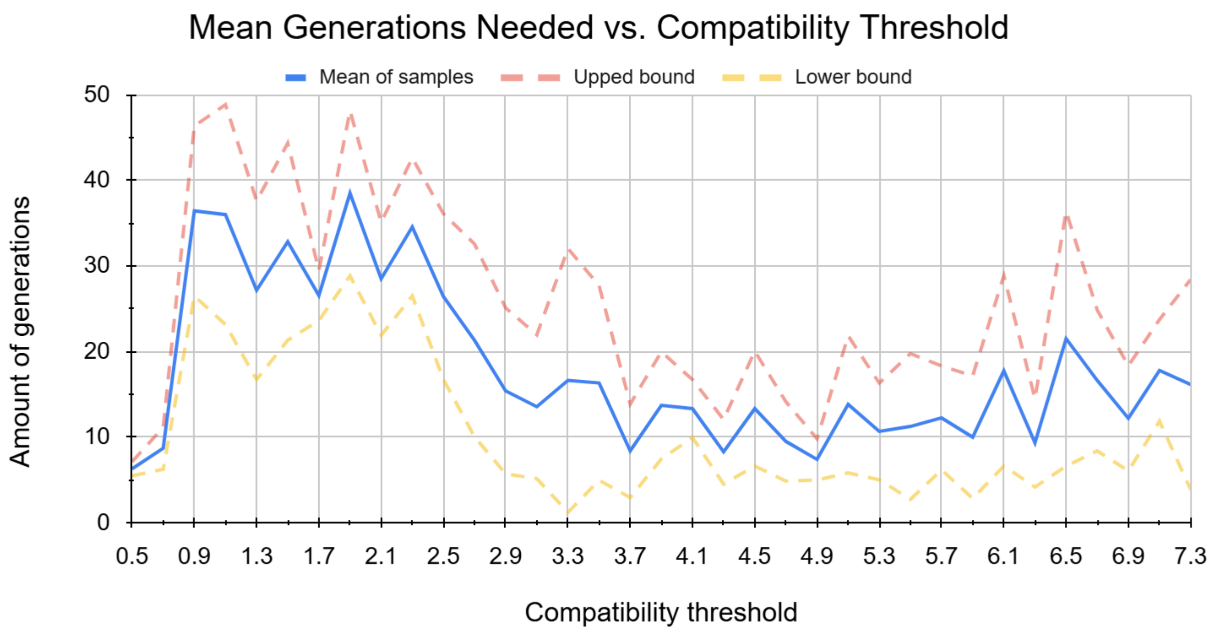


Figure 9: *Mean of samples and bounds of mean ± standard deviation for bounds*

With this graph, it's clear that the trend seen in the mean lies beyond the bounds of its expected deviation, proving that observations made with *figure 1* are valid. Given this, *figure 1* shows a positive correlation between the compatibility threshold and performance up to a threshold of ~2.3. Threshold values greater than this seem to decrease the amount of generations needed to around 14, after which their effect is negligible.

5.1.1 Survivorship Bias

An extremely important factor to take into account is the failed samples discussed previously. As mentioned in the methodology section, if no perfect agents are found within 50 generations, that sample is considered failed and is excluded when calculating the mean and standard deviation, which was a necessary compromise when performance was taken into account. This is problematic because this effectively excludes cases where the process takes more than 50 generations, meaning that the set of valid samples doesn't accurately reflect all data collected. While this can impact the previous observations, through deeper analysis, its influence on the data's validity can be mitigated.

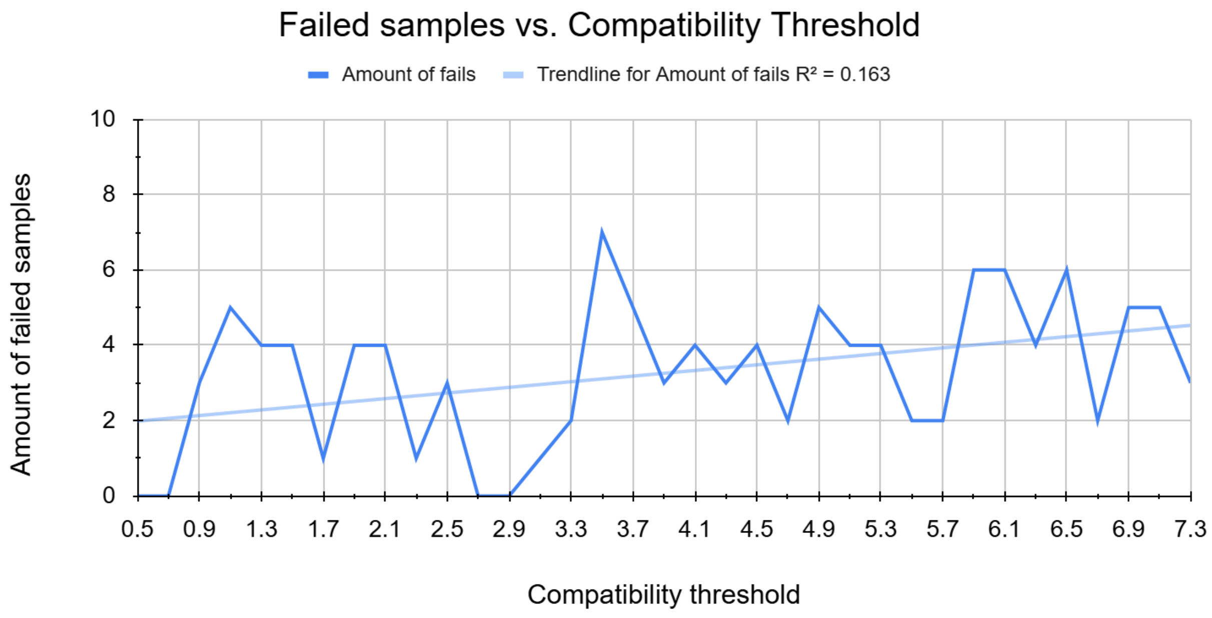


Figure 10: *Amount of samples (out of 10 total) that failed per threshold value*

On average there are ~3.3 / 10 samples per threshold that failed. The graph above shows a possible relationship between compatibility and the amount of failed samples. This would indicate that higher compatibility threshold values lead to more samples that require over 50 generations, and thus likely have a higher mean. Contrary to this, a drop in the mean of samples is seen at a compatibility threshold of ~2.3. This phenomenon can be attributed to the survivorship bias, wherein an incorrect inverse trend is formed due to the elimination of certain data. We can prove this by comparing the mean with the amount of failed samples:

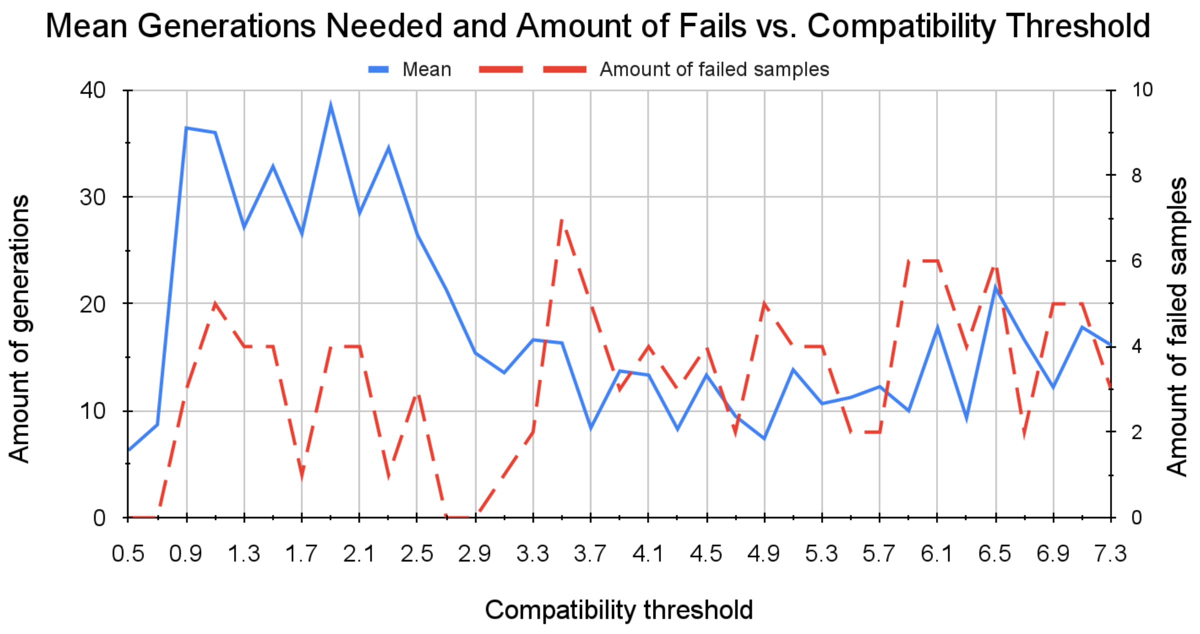


Figure 11: *Comparison of failed sample count and mean*

Initially, this visualization appears representative of a survivorship bias being the culprit of the decrease in the mean, as the increase in failed samples (samples that needed more than 50 generations) almost perfectly correlates with the lower mean. This suggests that the amount of generations needed grows increasingly inconsistent as higher compatibility thresholds are used.

5.2 Influence of Species Stagnation WIP

Unlike compatibility threshold, species stagnation yielded results that do not suggest a straightforward relationship with the convergence performance of the NEAT algorithm. Figures 3 and 4 illustrate that varying the species stagnation parameter between values of 3 to 49 did not lead to a clear trend or consistent improvement in the convergence speed. Instead, the samples demonstrate high variability and irregular fluctuations, reinforcing the stochastic nature inherent to genetic algorithms like NEAT.

Recalling that species stagnation dictates how lenient the algorithm is with non-improving species, once a species fails to increase its maximum fitness for a specified number of generations, it becomes ineligible for reproduction (Stanley and Miikkulainen), it might be expected that very low or very high stagnation thresholds would significantly alter how quickly good solutions are found. Yet, the data tells a different story. Regardless of whether the algorithm prunes stagnated species early or allows them to persist longer, there is no evident, consistent impact on the average number of generations required to achieve a perfect agent.

One hypothesis for this outcome is that the cart-pole balancing problem is relatively simple. Most species either discover a workable solution or become obsolete within just a few generations. Consequently, adjusting the tolerance for stagnation does not drastically reshape the population dynamics—stagnant species are often phased out quickly, and thriving species rapidly improve before hitting any stagnation threshold. This could mean that the algorithm is driven more by the immediate success of newly-mutated individuals than by the survival of suboptimal lineages.

Another related possibility is that small network topologies in early generations quickly converge on viable balancing strategies, reducing the need to rely on prolonged species preservation. With a higher stagnation parameter, the algorithm might allow a potentially useful species to stay around a bit longer, but if the cart-pole environment is already conducive to rapid improvement, the extra chance afforded to these species does not necessarily produce better or faster results. In other words, the evolutionary pressure to find a successful balancing agent is so strong that once a decent strategy emerges, it quickly proliferates, regardless of how leniently the algorithm treats stagnating species.

Finally, while the species stagnation parameter might not appear beneficial in this specific scenario, it can still serve an important role in more complex domains. In problems where it takes longer to discover partial solutions, or where multiple stepping-stone innovations must be nurtured, a more fine-tuned stagnation threshold may help to maintain diversity and keep the algorithm from prematurely converging on suboptimal solutions. Hence, its lack of clear effect here should not be interpreted as lack of utility in general; rather, it highlights the importance of context when tuning evolutionary parameters.

6 Conclusion

6.1 Summary of Findings

The results of this study provide valuable insights into how varying compatibility threshold and species stagnation parameters in NEAT can influence the number of generations required to evolve a successful cart-pole balancing agent. Even where the data show no strong correlation, the observations still clarify potential parameter ranges that might optimize performance. In particular, higher compatibility thresholds (above about 2.9) consistently reduced the required generations, suggesting that allowing greater structural diversity among species encourages more robust exploration. However, it is important to consider that survivorship bias and smaller sample sizes could mean the actual mean across many runs might be slightly higher than the illustrative plots indicate. Species stagnation, on the other hand, did not exhibit a starkly uniform trend, though very low or very high stagnation values appeared detrimental. Moderately flexible stagnation settings thus seem most conducive to striking a balance between preserving innovative structures and preventing unproductive species from lingering indefinitely.

6.2 Answer to the Research Question

In response to the central question,"To what extent does the compatibility threshold and species stagnation in the NEAT Genetic Algorithm influence its performance in terms of convergence speed in balancing a Cart-Pole Control System?", the findings suggest that higher compatibility thresholds (around 2.9 or more) notably improve convergence speed by fostering sufficient population diversity. Meanwhile, fine-tuning species stagnation is critical to prevent stagnation parameters from prematurely eliminating promising networks or perpetually retaining unproductive species. Together, these parameters exert a meaningful influence on how quickly NEAT can discover a robust solution to the cart-pole balancing task.

6.3 Future Works and Recommendations

Although this study centers on how many generations are required for the algorithm to meet a success criterion, measuring the actual time to evolve a perfect or near-perfect agent could offer more practical insights, especially for real-world implementations where computational cost and speed are paramount. A time-based examination could also illuminate subtler performance trade-offs that generational data alone might obscure, thereby showing how parameter adjustments can be tailored for more efficient runs.

To build on these findings, future investigations may explore how other parameters, (such as mutation rates, crossover probabilities, or the fitness function design), interact with compatibility threshold and species stagnation. Further applying these refined parameter insights to a range of benchmark problems, including robotics controls and game-playing agents, would help confirm whether the recommended ranges generalize across diverse tasks. By integrating these broader perspectives, subsequent research can offer more nuanced guidelines for practitioners seeking to balance complexity, efficiency, and reliability in NEAT-based systems.

7 References

7.1 Works Cited

"50 Years of Computer Science." *Departement Computerwetenschappen*, 22 Feb. 2024, wms.cs.kuleuven.be/cs/english/50-years-of-computer-science-1/50-years-of-computer-science. Accessed 13 Dec. 2024.

"A Pendulum with the Moving Pivot." *Physics Stack Exchange*, 22 Mar. 2022, physics.stackexchange.com/questions/700185/a-pendulum-with-the-moving-pivot. Accessed 23 Sept. 2024.

ajitjaokar. "The Mathematics of Forward and Back Propagation - DataScienceCentral.com." *Data Science Central*, 30 Apr. 2019, www.datasciencecentral.com/the-mathematics-of-forward-and-back-propagation/. Accessed 7 Oct. 2024.

"Cart Pole." *Www.gymlibrary.dev*, www.gymlibrary.dev/environments/classic_control/cart_pole/. Accessed 8 Oct. 2024.

"Configuration File Description." *Neat-Python.readthedocs.io*, neat-python.readthedocs.io/en/latest/config_file.html. Accessed 5 Oct. 2024.

Donges, Niklas. "Gradient Descent: An Introduction to One of Machine Learning's Most Popular Algorithms." *Built In*, 23 July 2021, builtin.com/data-science/gradient-descent. Accessed 7 Oct. 2024.

Felbert, Alexander. "The Universal Approximation Theorem – Deep Mind." *Deep Mind*, 26 Mar. 2023, www.deep-mind.org/2023/03/26/the-universal-approximation-theorem/. Accessed 7 Oct. 2024.

Florian, Razvan V. *Correct Equations for the Dynamics of the Cart-Pole System*. July 2005, pp. 1–6, coneural.org/florian/papers/05_cart_pole.pdf. Accessed 8 Oct. 2024.

"Genetic Algorithm in Machine Learning - Javatpoint." *Www.javatpoint.com*, www.javatpoint.com/genetic-algorithm-in-machine-learning#:~:text=A%20genetic%20algorithm%20is%20an,a%20long%20time%20to%20solve. Accessed 8 Oct. 2024.

Green, Colin D. "Equations of Motion for the Cart and Pole Control Task." *Sharpneat.sourceforge.io*, 4 Jan. 2020, sharpneat.sourceforge.io/research/cart-pole/cart-pole-equations.html. Accessed 23 Sept. 2024.

IBM. "What Is Machine Learning?" *IBM*, 2024, www.ibm.com/topics/machine-learning. Accessed 11 Dec. 2024.

Jafari, Kiana. "Understanding the Math behind Deep Neural Networks - Kiana Jafari - Medium." *Medium*, 6 May 2024, medium.com/@Kiana-Jafari/understanding-the-math-behind-deep-neural-networks-602b7b6a150a. Accessed 7 Oct. 2024.

Kelsey, Lewis. *Example of Getting Stuck at a Local Minimum*. i.sstatic.net/recxI.png. Accessed 7 Feb. 2025.

Machine Learning in Plain English. "Deep Learning Course — Lesson 5: Forward and Backward Propagation." *Medium*, Medium, 26 May 2023, medium.com/@nerdjock/deep-learning-course-lesson-5-forward-and-backward-propagation-ec8e4e6a8b92.

Matthew Kelly. "Cart-Pole Dynamics -- Part 1 of 2." *YouTube*, 14 June 2015, www.youtube.com/watch?v=Ongkt1_XjGM. Accessed 16 Sept. 2024.

Nautiyal, Dewang. “Underfitting and Overfitting in Machine Learning.” *GeeksforGeeks*, 23 Nov. 2017, www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/. Accessed 8 Oct. 2024.

Neutelings, Izaak. “Visual Illlustration of a Neural Network.” *Https://Tikz.net/Neural_networks/*, Online Image, tikz.net/wp-content/uploads/2021/12/neural_networks-001.png. Accessed 12 Dec. 2024.

Pargaonkar, Sandesh. “How Neural Networks Can Learn (Almost) Anything?” *The Road of Data*, 3 June 2023, sandesh21.hashnode.dev/how-neural-networks-can-learn-almost-anything. Accessed 7 Oct. 2024.

Saravanan, Dasaradh K. “A Gentle Introduction to Math behind Neural Networks.” *Medium*, 30 Oct. 2020, towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba. Accessed 7 Oct. 2024.

Stanley, Kenneth O., and Risto Miikkulainen. “Evolving Neural Networks through Augmenting Topologies.” *Evolutionary Computation*, vol. 10, no. 2, June 2002, pp. 99–127, https://doi.org/10.1162/106365602320169811. Accessed 8 Oct. 2024.

Warangal, Big Data, Analytics & Consulting Cell NIT. “GENETIC ALGORITHMS in MACHINE LEARNING.” *Medium*, 11 Feb. 2024, medium.com/@bdacc_club/genetic-algorithms-in-machine-learning-f73e18ab0bf9. Accessed 8 Oct. 2024.

Weber, Ethan. “Cart Pole.” *Ethanweber.me*, 2017, ethanweber.me/cartpole.html. Accessed 8 Oct. 2024.

“Welcome to NEAT-Python’s Documentation!” *Readthedocs.io*, neat-python.readthedocs.io/en/latest/. Accessed 23 Sept. 2024.

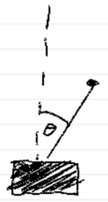
7.2 PyPI links for External Libraries Used

1. https://pypi.org/project/neat-python
2. https://pypi.org/project/pygame
3. https://pypi.org/project/pyperclip

8 Appendices

8.1 Appendix A: Cart-Pole Physics Working

m = point mass
a = angular vel
L = pendulum length



$\vec{a}_g = \sin(\theta)\cdot g$

$\vec{F}_g = m\cdot\sin(\theta)\cdot g$

t = torque

$\underline{t = F\cdot L}$

$t_g = gm\sin(\theta)\cdot L$

$\underline{\tau = I\cdot a}$

Mom of inertia for simple point mass:

$I = mL^2$

$\downarrow$

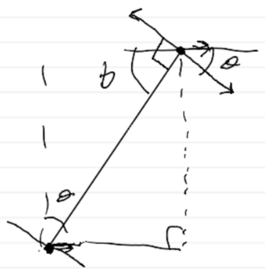
$FL = I\cdot a$

$gm\sin(\theta)\cdot L = mL^2 a$

$\downarrow$

$a = \frac{\cancel{gm}\sin(\theta)\cdot\cancel{L}}{\cancel{mL^2}}$

$a = \frac{g\sin(\theta)}{L}$



$a_c = \cos(\theta)\cdot c$

$F_L = \cos(\theta)\cdot cm$

$\underline{t = F\cdot L}$

$t_c = \cos\theta\cdot cmL$

$\underline{\tau = I\cdot a}$

Mom of inertia for simple point mass:

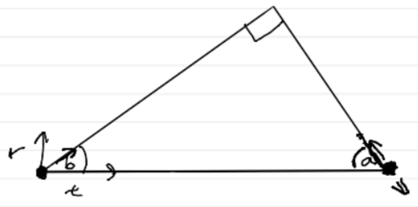
$I = mL^2$

$\downarrow$

$\cos\theta\cdot cmL = mL^2\cdot a$

$a = \frac{\cos\theta\cdot c\cancel{mL}}{\cancel{mL^2}}$

$a = \frac{\cos\theta\cdot c}{L}$



$b = 180 - 90 - a \rightarrow$

component forces:

$r = \sin(b)\, g$

$t = \cos(b)\, g$

8.2 Appendix B: Raw data

8.2.1 Raw data collected for varying compatibility thresholds

Number of generations needed to achieve a perfect agent with varying compatibility threshold values:

Compatibility threshold	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7	Sample 8	Sample 9	Sample 10
0.5	7	5	7	7	7	6	6	6	7	5
0.7	7	13	6	8	8	13	7	9	9	7
0.9	FAILED	37	48	27	39	27	FAILED	50	FAILED	27
1.1	FAILED	21	48	FAILED	FAILED	37	FAILED	49	FAILED	25
1.3	23	25	19	FAILED	25	FAILED	FAILED	48	23	FAILED
1.5	FAILED	25	45	27	24	FAILED	26	FAILED	50	FAILED
1.7	28	23	FAILED	32	28	23	28	27	24	26
1.9	39	FAILED	37	46	FAILED	50	37	FAILED	FAILED	22
2.1	33	FAILED	FAILED	25	24	FAILED	40	25	FAILED	24
2.3	28	FAILED	27	38	36	28	45	26	48	35
2.5	29	FAILED	36	10	24	FAILED	21	FAILED	39	26
2.7	42	24	13	31	8	24	20	8	31	12
2.9	27	13	28	12	4	17	3	18	5	27
3.1	10	5	11	7	8	28	27	11	FAILED	15
3.3	9	4	FAILED	FAILED	9	9	10	33	11	48
3.5	13	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	29	FAILED	7
3.7	FAILED	FAILED	7	10	FAILED	5	FAILED	17	3	FAILED
3.9	9	FAILED	18	8	16	FAILED	FAILED	25	9	11
4.1	13	19	11	14	9	FAILED	14	FAILED	FAILED	FAILED
4.3	9	15	8	FAILED	FAILED	8	10	4	FAILED	4
4.5	FAILED	12	FAILED	8	25	FAILED	FAILED	17	7	11
4.7	14	FAILED	13	8	9	FAILED	16	3	9	4
4.9	8	10	9	6	FAILED	4	FAILED	FAILED	FAILED	FAILED
5.1	FAILED	11	27	19	FAILED	5	FAILED	13	FAILED	8

5.3	6	FAILED	FAILED	21	FAILED	FAILED	8	6	11	12
5.5	3	FAILED	12	15	10	30	6	FAILED	9	5
5.7	19	FAILED	15	FAILED	14	21	5	5	11	8
5.9	14	FAILED	18	FAILED	FAILED	FAILED	FAILED	3	5	FAILED
6.1	FAILED	FAILED	19	19	3	FAILED	FAILED	30	FAILED	FAILED
6.3	10	7	FAILED	FAILED	7	FAILED	4	9	FAILED	19
6.5	FAILED	41	FAILED	FAILED	FAILED	12	FAILED	8	25	FAILED
6.7	15	25	27	14	8	7	26	11	FAILED	FAILED
6.9	21	FAILED	FAILED	9	FAILED	FAILED	6	FAILED	16	9
7.1	FAILED	18	FAILED	24	19	20	FAILED	8	FAILED	FAILED
7.3	FAILED	18	10	FAILED	6	40	4	13	FAILED	22

8.2.2 Raw data collected for varying species stagnation values

Number of generations needed to achieve a perfect agent with varying species stagnation values:

Species stagnation	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7	Sample 8	Sample 9	Sample 10
3	15	FAILED	14	FAILED	35	FAILED	8	44	34	FAILED
4	7	11	47	21	4	17	FAILED	24	9	10
5	FAILED	40	22	FAILED	9	FAILED	18	38	22	21
6	42	21	FAILED	FAILED	13	12	11	13	12	32
7	FAILED	22	FAILED	10	31	FAILED	16	11	5	15
8	15	7	5	FAILED	22	5	FAILED	FAILED	15	FAILED
9	FAILED	FAILED	3	49	21	13	8	20	45	20
10	9	16	29	17	4	32	FAILED	19	FAILED	FAILED
11	10	6	15	24	FAILED	22	11	40	4	11
12	5	FAILED	8	50	3	19	13	4	15	16
13	11	7	43	14	19	FAILED	22	12	FAILED	FAILED
14	9	15	17	FAILED	35	5	14	29	10	FAILED
15	FAILED	17	6	FAILED	FAILED	19	9	12	26	26
16	5	FAILED	15	FAILED	20	FAILED	FAILED	7	9	14

17	18	FAILED	13	9	FAILED	FAILED	14	20	28	FAILED
18	17	9	8	FAILED	19	FAILED	24	42	FAILED	11
19	FAILED	13	5	26	28	4	31	6	26	33
20	16	43	4	12	FAILED	7	FAILED	6	39	13
21	8	24	7	FAILED	6	FAILED	FAILED	FAILED	FAILED	16
22	7	FAILED	6	13	7	12	9	15	25	34
23	5	FAILED	19	20	9	FAILED	FAILED	9	11	FAILED
24	17	9	27	13	11	12	17	23	FAILED	8
25	FAILED	16	17	6	6	FAILED	25	22	FAILED	FAILED
26	9	FAILED	21	25	30	10	22	21	11	FAILED
27	13	36	FAILED	11	9	10	FAILED	FAILED	15	5
28	7	48	9	9	6	42	29	12	10	FAILED
29	FAILED	13	FAILED	12	FAILED	13	17	15	8	FAILED
30	FAILED	4	FAILED	38	26	11	19	6	3	FAILED
31	6	11	5	33	17	3	37	17	10	FAILED
32	14	10	21	31	FAILED	5	33	16	12	18
33	24	17	7	FAILED	FAILED	FAILED	19	8	9	37
34	20	FAILED	FAILED	8	FAILED	10	12	36	26	23
35	14	FAILED	9	24	FAILED	FAILED	11	17	FAILED	8
36	13	26	16	25	14	24	32	11	FAILED	5
37	FAILED	FAILED	FAILED	28	4	30	5	20	8	5
38	FAILED	FAILED	47	FAILED	10	FAILED	16	8	19	FAILED
39	23	33	15	FAILED	21	20	46	20	31	17
40	12	14	25	15	10	27	8	6	41	6
41	27	FAILED	10	7	FAILED	9	11	15	FAILED	9
42	8	FAILED	6	28	FAILED	8	3	8	24	FAILED
43	31	15	31	6	11	FAILED	5	FAILED	8	11
44	36	35	4	19	5	41	35	25	15	FAILED
45	38	FAILED	9	5	13	27	12	27	23	7
46	31	17	17	11	6	11	15	10	37	14
47	38	5	9	25	35	FAILED	11	9	45	FAILED
48	10	FAILED	15	14	FAILED	42	4	FAILED	FAILED	FAILED
49	FAILED	FAILED	8	18	21	37	6	9	FAILED	12
50	26	17	FAILED	29	12	FAILED	22	12	15	FAILED

8.3 Appendix D: Data Collection Scripts

8.3.1 Compatibility threshold collection script

collectCt.py

```
import pyautogui as pa
import pyperclip
from time import sleep

def updateCompatibilityThreshold(newValue):
    with open('config.txt', 'r') as file:
        lines = file.readlines()

    with open('config.txt', 'w') as file:
        for line in lines:
            if 'compatibility_threshold' in line:
                file.write(f'compatibility_threshold = {newValue}\n')
            else:
                file.write(line)


def getResult():
    pyperclip.copy('NA')
    pa.write('python main.py')
    pa.press('enter')
    while pyperclip.paste() == 'NA':
        sleep(1)

    try:
        r = int(pyperclip.paste())
        pyperclip.copy('NA')
        return r
    except:
        pyperclip.copy('NA')
        return None

def main():
    sampleAmount = 10

    sleep(2)

    ct = 0.5

    while ct < 7.4:
        updateCompatibilityThreshold(ct)
        count = 0
        roundData = []
        n = 0
        total = 0
        for _ in range(sampleAmount):
            result = getResult()
            roundData.append(result)
            if result:
                n += 1
                total += result
            else:
                count += 1

        print(f'\nFor CT {ct}:\n{roundData}')
        if count > sampleAmount / 2:
            print('FAILED')
        else:
            print(f'Average of {total/n}')

        ct += 0.2


if __name__ == '__main__':
    main()
```

8.3.2 Species Stagnation Data Collection Script

collectSs.py (collection script for species stagnation)

```python
import pyautogui as pa
import pyperclip
from time import sleep

def updateMaxStagnation(newValue):
    '''
    This function updates the max_stagnation in the config.txt file.
    '''
    with open('config.txt', 'r') as file:
        lines = file.readlines()

    with open('config.txt', 'w') as file:
        for line in lines:
            if 'max_stagnation' in line:
                file.write(f'max_stagnation = {newValue}\n')
            else:
                file.write(line)


def getResult():
    pyperclip.copy('NA')
    pa.write('python main.py')
    pa.press('enter')
    while pyperclip.paste() == 'NA':
        sleep(1)

    try:
        r = int(pyperclip.paste())
        pyperclip.copy('NA')
        return r
    except:
        pyperclip.copy('NA')
        return None

def main():
    sampleAmount = 10

    sleep(2)

    Ss = 3

    while Ss <= 50:
        updateMaxStagnation(Ss)
        count = 0
        roundData = []
        n = 0
        total = 0
        for _ in range(sampleAmount):
            result = getResult()
            roundData.append(result)
            if result:
                n += 1
                total += result
            else:
                count += 1

        print(f'\nFor Ss {Ss}:\n{roundData}')
        if count > sampleAmount / 2:
            print('FAILED')
        else:
            print(f'Average of {total/n}')

        Ss += 1


if __name__ == '__main__':
    main()
```

8.4 Appendix E: Main File

main.py (main script that handles the evolution process)

```
from time import time
from os import environ, path
from vector import Vector2d
from math import sin, cos, pi
from random import random
import neat

environ['PYGAME_HIDE_SUPPORT_PROMPT'] = 'hide'
import pygame

from window import Window


class Game(Window):
    class Cart:
        friction = 0.15

        def __init__(self, winSize) -> None:
            self.pos = Vector2d(winSize[0] / 2, winSize[1] / 2)
            self.vel = 0
            self.acceleration = 0
            self.winSize = winSize

            self.inputs = [False, False]

        def update(self, timeCoefficient):
            if (self.inputs[0] and self.inputs[1]) or (not self.inputs[0] and not
self.inputs[1]):
                self.acceleration = 0
            elif self.inputs[0]:
                self.acceleration = -2
            else:
                self.acceleration = 2

            self.acceleration -= self.vel*Game.Cart.friction

            self.vel += self.acceleration * timeCoefficient

            self.pos.x += self.vel * timeCoefficient
            if self.pos.x < 25:
                self.pos.x = 26
                self.vel = 0
                self.acceleration = 0

            elif self.pos.x > self.winSize[0] - 25:
                self.pos.x = self.winSize[0] - 26
                self.vel = 0
                self.acceleration = 0

    class Pendulum:
        drag = 2.5
        gravity = 0.05

        def __init__(self, winSize: tuple[int, int], length: float, angle: float)
-> None:
            self.pos = Vector2d(winSize[0] / 2, winSize[1] / 2)
            self.mass = 0.1
            self.length = length

            self.angle = angle
            self.angularVelocity = 0
            self.angularAcceleration = 0

        def update(self, cartAcceleration, timeCoefficient):
            # computes angular accel from grav and cart movement
            #                                    gravity part
cart part
            self.angularAcceleration = (-Game.Pendulum.gravity * sin(self.angle) /
self.length)  +  (cos(self.angle) * cartAcceleration / self.length)

            self.angularAcceleration -=
self.angularVelocity*Game.Pendulum.drag/self.length    # drag as a negative
acceleration instead of velocity mult

            # standard
            self.angularVelocity += self.angularAcceleration * timeCoefficient
            self.angle += self.angularVelocity * timeCoefficient

            self.angle = self.angle % (2 * pi)

    class Agent:
        def __init__(self, winSize, angle: float) -> None:
            self.pendulum = Game.Pendulum(winSize, 100, angle)
            self.cart = Game.Cart(winSize)

            self.streak = 1
            self.inStreak = False
    def __init__(self, winSize: tuple[int, int] | str = (1000, 700), render: bool =
True, title: str = 'Window', backgroundColor: tuple[int, int, int] = (20, 20, 20))
-> None:
        super().__init__(winSize, title, backgroundColor)

        self.fps = 60
        self.fpsReference = 60  # IE: targeted fps

        self.render = render
        self.dir = 0.05
        self.passedLast = False

        self.agents = []

        if self.render:
            self.initPygame()

    def create(self, genomes, nets, ge, config):
        self.genomes = genomes
        self.nets = nets
        self.config = config
        self.ge = ge

        self.agents.clear()

        for _, g in self.genomes:
            g.fitness = 0
            net = neat.nn.FeedForwardNetwork.create(g, self.config)
            self.nets.append(net)

            self.agents.append(Game.Agent(self.winSize, pi + self.dir))

            self.ge.append(g)

        self.dir *= -1

    def update(self, dt: float):
        timeCoefficient = dt * self.fpsReference  # like dt but weighted/normalized
with fps reference so its 1 when running at targeted fps

        for id, agent in enumerate(self.agents):
            output = self.nets[id].activate([agent.pendulum.angle,
agent.pendulum.angularVelocity, agent.cart.pos.x])

            if output[0] > 0.75:
                agent.cart.inputs[0] = True
            else:
                agent.cart.inputs[0] = False

            if output[1] > 0.75:
                agent.cart.inputs[1] = True
            else:
                agent.cart.inputs[1] = False

            agent.cart.update(timeCoefficient)
            agent.pendulum.update(agent.cart.acceleration, timeCoefficient)


            if agent.pendulum.angle >= pi/2 and agent.pendulum.angle <= pi + pi/2:
                agent.inStreak = True
                agent.streak *= 1.014
            else:
                if agent.inStreak:
                    self.ge[id].fitness += agent.streak
                    agent.streak = 1
                    agent.inStreak = False


    def run(self, o):
        '''Main loop'''
        dt = 1/self.fps
        self.running = True
        frame = 0
        while self.running:
            if self.render:
                self.handleEvents()

                self.update(dt)

                self.screen.fill(self.backgroundColor)  # Clear screen with black

                self.draw()

                pygame.display.flip()  # Update the display

                dt = self.clock.tick(self.fps) / 1000.0  # Delta time in seconds
(60 fps)

            else:
                self.update(dt)
            frame += 1
            if frame > 500:
                self.running = False
        highestFitness = -1

        for id, agent in enumerate(self.agents):
            if agent.inStreak:
                self.ge[id].fitness += agent.streak
                agent.streak = 1
                agent.inStreak = False

            if self.ge[id].fitness > highestFitness:
                highestFitness = self.ge[id].fitness
                self.bestGenome = self.ge[id]

        if self.passedLast:
            if highestFitness >= 800:
                print(f'{o}   {round(highestFitness)}   {self.passedLast}')
                raise StopIteration()
            else:
                self.passedLast = False
        else:
            if highestFitness >= 800:
                self.passedLast = True
        print(f'{o}   {round(highestFitness)}   {self.passedLast}')


    def draw(self):
        pygame.draw.line(self.screen, (255, 255, 255), (0, self.winSize[1] / 2),
(self.winSize[0], self.winSize[1] / 2))

        for i, agent in enumerate(self.agents):
            if i == 0:
                pygame.draw.rect(self.screen, (0, 255, 0),
pygame.rect.Rect(agent.cart.pos.x-25, agent.cart.pos.y-12.5, 50, 25))
            else:
                pygame.draw.rect(self.screen, (200, 0, 0),
pygame.rect.Rect(agent.cart.pos.x-25, agent.cart.pos.y-12.5, 50, 25))

            pygame.draw.line(self.screen, (255, 255, 255), (agent.cart.pos.x,
agent.cart.pos.y), (agent.cart.pos.x + cos(agent.pendulum.angle +
pi/2)*agent.pendulum.length, agent.cart.pos.y + sin(agent.pendulum.angle +
pi/2)*agent.pendulum.length), 3)


def eval_genomes(genomes, config):
    global o, a
    nets = []
    ge = []
    a.create(genomes, nets, ge, config)
    a.run(o)
    o += 1

def run_neat(config_path):
    import pickle

    # p = neat.Checkpointer.restore_checkpoint('neat-checkpoint-85')
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                        neat.DefaultSpeciesSet, neat.DefaultStagnation,
                        config_path)

    global a
    a = Game(render = False)
    p = neat.Population(config)
    # p.add_reporter(neat.StdOutReporter(True))

    global o
    o = 1

    try:
        p.run(eval_genomes, 50)
        print('Never found after 50 generations\n\n')
        return 'None'
    except StopIteration:
        print(f'Reached goal after {o} generations.\n\n')
        return o

if __name__ == '__main__':
    import pyperclip
    local_dir = path.dirname(__file__)
    config_path = path.join(local_dir, 'config.txt')

    pyperclip.copy(run_neat(config_path))
```