Evolutionary strategy for a virtual robotic control

BSc Computer Science Final Project

Kenjiro Ono

March 10, 2023

1. Introduction

Kark Sims first proposed the genetic algorithm with evolving virtual creatures [1]. Since then, genetic algorithms itself have been evolving and emerged into various domains. In this project, neuroevolution, which uses evolutionary computation to evolve neural networks. The neural networks are then used to successfully control a virtual robot in a simulation environment. Specifically, the NEAT (NeuroEvolution of Augmenting Topologies) [2] introduced by Kenneth O. Stanley and Risto Miikkulainen are employed in this project. The NEAT initialises with the minimal network topology, then the topology and network weights are co-evolved incrementally making the algorithm biassed towards a search for the minimal network topology for a given task.

Neural network is widely adapted for robotic control tasks as it can address complex, noisy and non-linear problems which are the type of problems robots would encounter in the real-world application.

The simulation environment can virtually replicate the robotic system interacting with the real-world and evaluate control policies at much higher rate than testing such policies in the real-world environment one procedure at a time, accelerating the system development cycle. Also, the simulation can be multi-threaded to further accelerate and search through greater search spaces. For the simulation environment, OpenAI gym with Mujoco simulation environment is used.

This project is based on the template of Automated design using evolutionary computation from CM3020 Artificial Intelligence.

1.1 Evolutionary Computation vs Supervised Learning

Although conventional machine learning techniques such as deep learning performs well on well-behaved tasks such as image classification using convolutional neural networks, it has not performed well for not-well-behaved tasks or reinforcement learning problems such as gameplay or robotic control. One of the reasons is that the availability of training data is limited to effectively train the network using statistical methods such as gradient descent. On the other hand, evolutionary computation or reinforcement learning does not require such a training dataset, therefore, can be applied to the aforementioned tasks.

The drawback of evolutionary computation is that it is computationally expensive. Unlike the machine learning algorithms training a single model with gradient descent or backpropagation, the evolutionary algorithm evaluates a large number of models for each evaluation cycle.

1.2 Evolutionary Computation vs Reinforcement Learning

Both evolutionary computation and reinforcement learning evaluates the fitness or reward of the agent accumulated interacting with a given environment, however, they are different in their approach. In evolutionary computation, it employs the theory of evolution, where the evaluation cycle starts with many agents and the best performing agent is selected to reproduce and mutate for the next generation or evaluation cycle. It only considers the optimal solution and the suboptimal solutions are discarded. In reinforcement learning, a sole agent interacts with the environment in different ways and learns both positive and negative outcomes of certain actions taken. By the difference in their approach, reinforcement learning could be more susceptible to local optimization as only one agent is interacting with the environment, whereas evolutionary computation can reach larger search space as many agents are randomly initialised at the expense of discarding the negative and suboptimal information to learn.

1.3 Conventional Neuroevolution vs TWEANNs (Topology and Weight Evolving Artificial Network algorithms)

There have been many proposed neuroevolution algorithms thus far. Two primary distinctions are algorithms that only evolve network weights of pre-defined topology called conventional neuroevolution, and algorithms that evolve both the weights and topology called TWEANNs (Topology and Weight Evolving Neural Network) which the NEAT algorithm belongs to. The conventional neuroevolution can utilise the performance proven network architectures to start the search with only to concern optimising the network weights, however, a dimensionality of the problem space is fixed relative to the network architectures. On the other hand, the TWEANN algorithms initialise the search from minimal network topology and it could arrive at the solution with a smaller network topology than the fixed-topology, which reduces the dimensionality of a problem space making it an efficient solution.

There are several advantages of minimising the network topology. It can minimise the computational resource required, making the process faster and more memory efficient. This is beneficial for mobile systems such as robotics, where the real-time inference is critical, computational resources and energy supply might be constrained. Also, by minimising the network size, it makes the search for optimal network weights more efficient.

2. Literature Review

2.1 GNARL (GeNeralized Acquisition of Recurrent Links)

GNARL (GeNeralized Acquisition of Recurrent Links) [3] is an evolutionary algorithm which evolves both topology and weight of neural networks. In this paper, limitations of conventional genetic algorithms using the crossover to evolve neural networks are discussed and GNARL is proposed to address the limitations.

The first limitation discussed is “competing conventions problem” [4]. Suppose that there are networks that share both a common topology and common weights, as interpretation functions may be many-to-one, they do not have to have identical bit string representation. Crossover of these networks may create a network containing repeated connection and components and discard the computational ability of parent networks whose performance tends to be worse. As the network size increases, the competing convention problem would increase exponentially.

Secondly, a neural network can generalise and provide multiple solutions to a problem. This means that the problem representations are distributed across hidden units, connections and weights. A removal of a small number of hidden nodes has minor altercations in the performance as network connections and weights values prominently determine the network performance, thus, arbitrary crossover of networks with identical topologies but different weights are unlikely to produce a viable network.

Finally, as the number of hidden units and the network connectivity increases, so do the network’s representations of a problem, therefore, topologically distinct parent networks are unlikely to produce good offspring.

GNARL addresses the aforementioned limitations by, given a parent, η, introducing the temperature, $T(\eta)$, which determines the severity of mutation,

$$T(\eta) = 1 - \frac{f(\eta)}{f_{max}} \tag{1}$$

where, $f(\eta)$ is the fitness of the parent and f_{max} is the maximum fitness for a given task.
Therefore, the higher the temperature, mutation is more severe, on the contrary, lower temperature means it is closer to the solution, thus, mutating only slightly. This allows the search of topology and wights to be broad initially, then the search space is narrowed as a network approaches a solution.

Additionally, to preserve the fitness of the parent network, when new links are added, they are initialised with zero weight, making the behaviour of the modified network unchanged. Similarly, when new hidden nodes are added, they are initialised without any incident connections.

Although, GNARL algorithm addresses the aforementioned limitations of the genetic algorithm to some extent, it does not completely illuminate the "competing convention problem" as the algorithm can still perform crossover on networks with a common topology and common weights. Also, the fitness of the parent network is preserved by initialising the added links and hidden nodes, they may never be fully integrated to the network itself, adding complexity to the parametric search. Thus, the question of evolving topologies along with weights providing an advantage over evolving weights of a fixed-topology is inconclusive.

2.2 Neuroevolution of augmenting topologies

NEAT (Neuroevolution of augmenting topologies) [1] is a TWEANN algorithm in which network topology and weights are evolved together where topology is biassed toward minimality.

In the NEAT, the question asked thus far about the TWEANN approach is answered to some extent. It suggests that if implemented correctly, it can produce minimised topology and result in a significant gain in performance and learning efficiency.

In NEAT, a concept of history making is used to address the competing conventions problem. The history marking uses the innovation number that is a global identifier number assigned to each gene. Whenever a new gene appears through mutation, the innovation number is incremented and assigned to the gene thus providing a way to identify functionality equivalence of different network topologies (e.g. hidden layer of [A,B,C] is functionally equivalent to [C,B,A]).

Adding nodes and connections typically initially decreases a performance of the network as the problem representation in the network is partially lost when mutated. This suggests that the mutated networks are more likely to be discarded for each generation cycle making the topological innovation to stagnate. NEAT uses speciation to protect such innovation.

The population is speciated into their own niches to compete, therefore, the new topologies have time to optimise themselves. The speciation is determined by measuring the compatibility distance using the innovation number to match up each component of the topologies. Higher numbers of disjoint and excess components suggest that compared topologies are likely to belong to different species.

Typically TWEANN algorithms start with an initial population of random topologies to diversify the search surface. In contrast, NEAT starts out the search with a uniform population of minimal topologies with no hidden nodes. New components are added incrementally and only the viable new topologies are chosen for the evolution cycle. This ensures the search is always biassed towards minimal-dimensional space, which gives a performance advantage compared to the other approaches such as evolving a fixed-topology.

In the paper, the performance of NEAT is evaluated using the Double Pole Balancing benchmark and compared with the other Neuroevolution algorithms. NEAT shows fewer evaluations and topological components needed to archive the sufficient score in the benchmark than the other algorithms (CE [5], ESP [6]). Furthermore, the effectiveness of key components of NEAT that are historical marking, speciation and incremental topological growth are evaluated individually and concluded that all these components contribute to the performance and fewer evaluations needed for the algorithm to arrive at the solution.

2.3 Compositional Pattern Producing Networks

CPPN (Compositional Pattern Producing Networks) [7] uses an encoding scheme analogous to nature. In nature, simple genotypes evolve to be significantly complex phenotypes by exploiting the symmetry, reuse, reuse with variation, preservation of regularities, and elaboration of existing regularities. In CPPN, such properties are represented by mathematical functions such as sine to produce segmented patterns with repetitions, Gaussian gradient to produce symmetry and linear functions to produce linear or fractal-like patterns to construct complex systems.

The evolution in nature starts from a minimal system and gradually becomes more complex. To evolve CPPN in a similar fashion, NEAT is used which starts off the evolution with a uniform minimal system then the complexity is added gradually evaluating the fitness at each evolutionary step.

2.4 Hypercube-based NEAT

HyperNEAT (Hypercube-based NEAT) [8] evolves neural networks using properties (symmetry, repetition, and repetition with variation) from CPPN. Utilising such properties, HyperNEAT can efficiently evolve larger neural networks that look more like neural connectivity patterns in the brain. Unlike other neural learning algorithms, HyperNEAT sees the geometry of the problem domain which significantly enhances the learning process.

3. Design

3.1 Overview

The main objective of this project is to develop artificial neural networks using an evolutionary algorithm to solve reinforcement learning tasks. Specifically, the neuroevolution algorithm is used to evolve a neural network to successfully control a virtual robot in a simulation environment.

To accurately simulate the dynamics of the robot, a physics simulator is used. The robot description is parsed and deployed in the simulation. For the simulator, Mujoco physics simulator is used and a robot described in a MJCF file, similar to an xml file is deployed in the simulator for the reinforcement learning task.

The environment is set as a OpenAI gym environment, which provides a convenient way to interface with the simulator in the code. Within the gym environment, various functions are defined for observation, action and reward calculations which are then used in the NEAT algorithm to evolve neural networks.

3.2 Template for this project

The template for this project is Automated design using evolutionary computation from CM3020 Artificial Intelligence.

3.3 Domain and users

The primary domain of this project is to archive robotic control using the neuroevolution algorithm.

The primary users for this project are the robot developer and researchers. The robotic developer can deploy the designed robot in a simulation environment and examine how the robot interacts with the environment without explicitly requiring to program the control of the robot, therefore, the robotic developer can conduct prototyping rapidly.

This project can also facilitate an accurate physics simulation environment for researchers to test out various reinforcement learning algorithms. The environment can also be modified to fit the requirements of each user.

3.4 Overall Structure

For the primary development programming language, python is chosen as the software dependencies for this project support python which are mentioned below. Also python provides a convenient way to manipulate and visualise data using numpy, pandas and matplotlib, which are used to evaluate the project.

The python implementation of the NEAT algorithm which only depends on the standard library is used in the project. The hyper-parameters of the algorithm can be edited on a configuration text file. It implements speciation by measuring the compatibility of the genomes and keeps track of the improvement to identify the stagnation of each species. It provides a graphical reporting tool for fitness value and speciation of the neural networks, also, it provides graphical representation of the network topology.

For the simulation environment, Mujoco physics simulator is used, which is a general purpose physics engine used for research and development in robotics, biomechanics and other domains requiring fast and accurate simulation. Mujoco is chosen as it offers python bindings and supports MJCF file format which is akin to URDF file format to describe a robot in the simulator.

OpenAI gym is an environment for developing and learning agents. It is particularly suited for reinforcement learning agents. The gym can be installed with the predefined reinforcement learning environments and accessed from the python code. The classical task or environment such as cart-pole balancing often used as a benchmarking task is included within the gym installation. The environment provides a convenient interface to access observation and reward of the agent, which are then used by the NEAT algorithm to generate action and to evolve the neural networks.

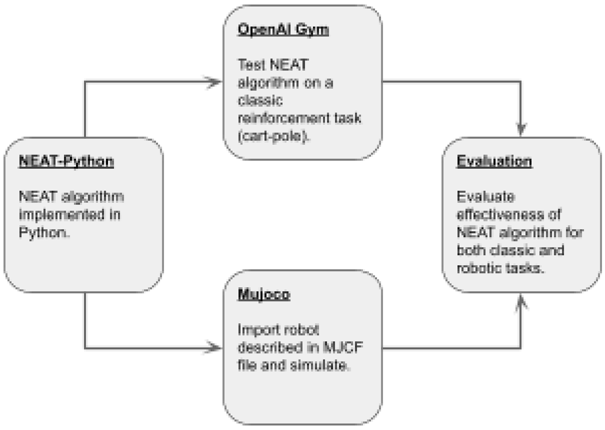

OpenAI Gym
Test NEAT algorithm on a classic reinforcement task (cart-pole).
NEAT-Python
NEAT algorithm implemented in Python.
Evaluation
Evaluate effectiveness of NEAT algorithm for both classic and robotic tasks.
Mujoco
Import robot described in MJCF file and simulate.

Figure 1. Overview of interaction between NEAT-Python, OpenAI Gym environment, Mujoco physics simulator and evaluation process.

3.6 Evaluation

The implementation of the evolutionary algorithm is evaluated using the common reinforcement learning benchmark tasks such as classic cart-pole-balancing. The implementation of the evolutionary algorithm for the robotic control is evaluated in the simulation environment. The fitness is calculated by the distance travelled or efficient use of actuators to achieve certain behaviours. For instance, the efficiency can be calculated by aggregating the total amount of actuated lengths of the actuators.

To evaluate the characteristics of the algorithm under different conditions, various hyper-parameter settings and fitness functions are used.

3.6 Project plan

The gantt chart for the work plan is shown in **Appendix A**. The project plan is planned on weekly bases across the span of the project.

4. Implementation

4.1 Overview

The NEAT algorithm and its evolved neural networks are evaluated on reinforcement tasks with different difficulties to find the solution.

Firstly, the algorithm is evaluated on the cart-pole balancing task. The task is set up with a cart on a frictionless track and a pole is attached by an un-actuated joint to the cart. The goal of this task is to balance the pole upright by applying forces in the lateral direction on the cart. This task is commonly used as a benchmarking task to evaluate reinforcement learning algorithms. The task is simple to solve as there are only two possible actions to consider given a single output value, therefore, it is suitable to compare the performance of different algorithms.

Secondly, the algorithm is evaluated on a robot control task in a physics simulation environment. Using the physics simulator, a robot control strategy can be accurately simulated and evaluated. Compared to the cart-pole task, a robotic control task can be much more complex as robots typically have multiple joints and actuators. Because of the vast amount of possibilities for the robots to both observe and take action, it makes a challenging reinforcement learning task.

By default, the NEAT is configured that the population consists of 150 randomly initialised genomes and each genome would not have any hidden nodes or connections to evolve the network from the minimal network topology. This is to evaluate one of the advantages of the NEAT algorithm of which it tends to search for the minimal network topology.

4.2 NEAT with OpenAI Cart Pole

The cart-pole balancing is a classic benchmarking reinforcement learning task and it is predefined in the gym environment. The environment is imported in code and predefined action space, observation space and reward functions shown in **Appendix B** are used in the NEAT algorithm. The environment only takes action value either 0 to push the cart to left with a constant force or 1 to push the cart to right with a constant force.

The NEAT-Python [9] is imported in the code and configured refering the configuration file. For each genome in the population, a neural network is constructed and activated by passing the observation of the environment as input values and an output value is generated. For the activation function, tanh is used which returns an output value ranged between -1 and 1. Since the action can only be either 0 or 1, output value is converted to 1 if it is greater than 0, and to 0 otherwise. Based on the output value, the agent takes an action in the environment which generates new observations and a reward based on the outcome of the action.

For this environment, two different fitness functions are used. The first fitness function accumulates reward for each step the agent keeps the pole blanched on the cart. The second fitness function accounts the cart position deviated from the initial position and reward is discounted by the absolute value of the deviation,

$$reward = reward - \alpha\,|cartPos| \quad (2)$$

Where, α is used to alter the proportion of the discount by the change in the cart position. This will force the search towards the neural network which holds the pole upright at the initialised position which adds a slight complexity to the task.

The accumulated rewards are used to evaluate the fitness of the genome. Once every genome in the population is evaluated, the population for the next generation is generated by reproducing among the best performing genomes. Also, their network topologies are evolved based on the various hyper-parameters.

4.3 NEAT with Mujoco

To evaluate the NEAT algorithm on robotic control tasks, a custom gym environment is constructed using Mujoco physics simulator. In the environment, a xml file describing a robot (Unitree A1 quadruped robot) is imported and observation, action, reward function and other simulation environment settings are defined.

The A1 xml file is acquired from Mujoco Menagerie which is licensed under a BSD 3-Clause. The model consists of four legs and a trunk. Each leg has 3 positional actuators (servo motors) to control abduction, hip and knee joints, thus, action space takes twelve input values. The observation space takes 37 values which are 19 positional and 18 velocity values of the parts of the robot.

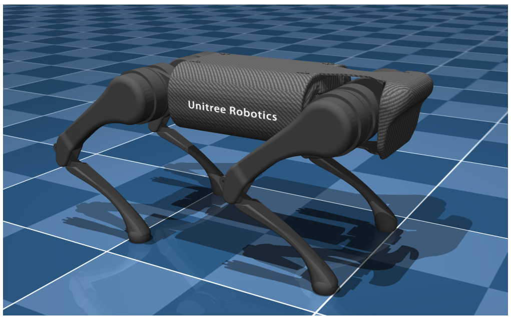


Figure 2. Unitree A1 model in Mujoco physics simulator.

For each genome in the population, a neural network is constructed and activated using the observation of the robot in the environment as input values. The activation function uses the tanh function, therefore, output values ranging between -1 and 1 are returned. However, each joint type has a different range of motion, therefore, the output values are interpolated to the corresponding joint range value. The robot receives the interpolated action values and actuated. The environment generates a reward value from the outcome of the action and accumulated as the fitness of the genome.

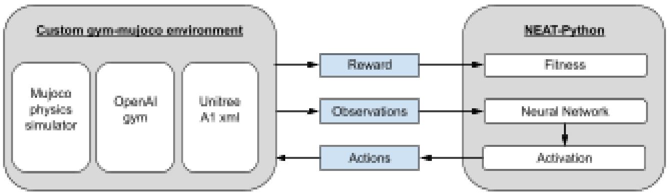


Figure 3. Overview of interaction between NEAT-Python and custom gym environment with Mujoco physics simulator.

The reward is calculated by the distance the robot travelled in forward direction. Additionally, a control cost is derived from the difference in between previous and subsequent actions and penalised from the final reward,

$$forward\,reward = \alpha\frac{x-x'}{dt} \quad (3)$$

$$control\,cost = \omega(\,action_{prev} - action_{new})^2 \quad (4)$$

$$reward = forward\,reward - control\,cost \quad (5)$$

Where, α is the forward reward weight, x is the current position, x' is the previous position, dt is the time unit and ω is the control cost weight. This rewards the control strategy which efficiently actuates the motors to gain the distance travelled. A mobile system such as a robot might have limited battery capacity, therefore, minimising the actuation to conserve energy is deseable for the robot to operate at maximum capacity. Also, this constrains the robot to exploit the limitation of the simulator to simulate the physics in the real-world.

5. Evaluation

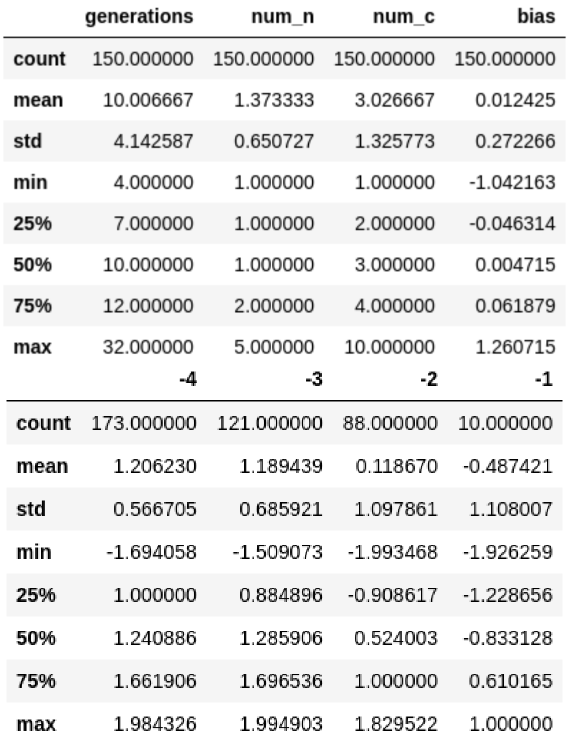
5.1 OpenAI Cart Pole environment

To evaluate the performance of the NEAT in a cart-pole environment, two different fitness functions are used. The first fitness function accumulates reward for each step the agent keeps the pole blanched on the cart. The second fitness function accounts the cart position deviated from the initial position and reward is discounted by the absolute value of the deviation.

For each genome in the population, the environment was run for 500 steps and fitness threshold was set at 499.5. Once a genome archives a fitness value over the fitness threshold, the evaluation is terminated. The evaluation was run 150 times for each fitness function to acquire average data points such as number of generations required to archive the fitness threshold, number of hidden nodes or connections, bias and weight value.

The NEAT is configured to be conservative to evolve by setting the various parameters low to generate a minimal network topology as it is a simple reinforcement learning task. The population is set as 150 to evaluate both fitness functions. The configuration for the NEAT algorithm is shown in **Appendix C**.

5.2 150 evaluations without cart position discount



	generations	num_n	num_c	bias
count	150.000000	150.000000	150.000000	150.000000
mean	10.006667	1.373333	3.026667	0.012425
std	4.142587	0.650727	1.325773	0.272266
min	4.000000	1.000000	1.000000	-1.042163
25%	7.000000	1.000000	2.000000	-0.046314
50%	10.000000	1.000000	3.000000	0.004715
75%	12.000000	2.000000	4.000000	0.061879
max	32.000000	5.000000	10.000000	1.260715

	-4	-3	-2	-1
count	173.000000	121.000000	88.000000	10.000000
mean	1.206230	1.189439	0.118670	-0.487421
std	0.566705	0.685921	1.097861	1.108007
min	-1.694058	-1.509073	-1.993468	-1.926259
25%	1.000000	0.884896	-0.908617	-1.228656
50%	1.240886	1.285906	0.524003	-0.833128
75%	1.661906	1.696536	1.000000	0.610165
max	1.984326	1.994903	1.829522	1.000000

Figure 4. Distribution of data points for 150 evaluations. '-4' depicts pole angular velocity, '-3' depicts pole angle, '-2' depicts cart velocity and '-1' depicts cart position.

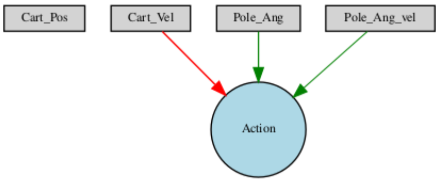
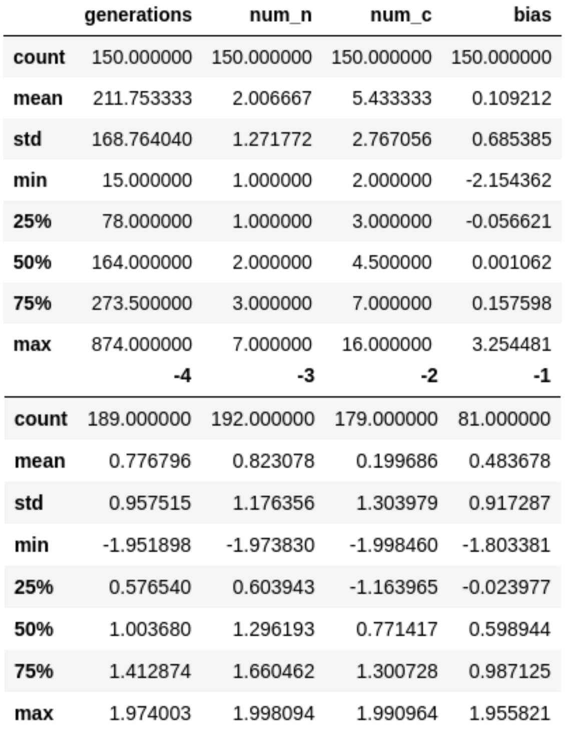


Figure 5. Network topology of the best performing genome.

From the evaluation, it is observed that the pole-balancing task can be solved by a minimal network topology often without any hidden nodes and the fitness threshold is achieved on average, after 10 generations. On average, the input node that takes 'pole angular velocity' has the highest weight, followed by the input node for 'pole angle', 'cart velocity' then 'cart position'. This explains that the 'pole angular velocity' is the most significant input data and the 'cart position' is the least significant input data to determine the action. This is understandable due to the fact that the 'cart position' itself cannot determine whether the pole is balanced or not. On the contrary, the 'pole angular velocity' and the 'pole angle' can directly provide the state of the pole which essentially determines the success of the task. Figure 5 depicts the sample of best performing network topology of the task, where 'cart position' is not considered at all. The distributions of data points are shown in **Appendix D**.

5.3 150 evaluations with cart position discount



	generations	num_n	num_c	bias
count	150.000000	150.000000	150.000000	150.000000
mean	211.753333	2.006667	5.433333	0.109212
std	168.764040	1.271772	2.767056	0.685385
min	15.000000	1.000000	2.000000	-2.154362
25%	78.000000	1.000000	3.000000	-0.056621
50%	164.000000	2.000000	4.500000	0.001062
75%	273.500000	3.000000	7.000000	0.157598
max	874.000000	7.000000	16.000000	3.254481

	-4	-3	-2	-1
count	189.000000	192.000000	179.000000	81.000000
mean	0.776796	0.823078	0.199686	0.483678
std	0.957515	1.176356	1.303979	0.917287
min	-1.951898	-1.973830	-1.998460	-1.803381
25%	0.576540	0.603943	-1.163965	-0.023977
50%	1.003680	1.296193	0.771417	0.598944
75%	1.412874	1.660462	1.300728	0.987125
max	1.974003	1.998094	1.990964	1.955821

Figure 6. Distribution of data points for 150 evaluations. '-4' depicts pole angular velocity, '-3' depicts pole angle, '-2' depicts cart velocity and '-1' depicts cart position.

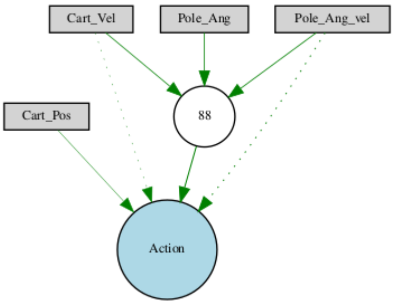


Figure 7. Network topology of the best performing genome.

For the evaluations with cart position discount, the fitness threshold is achieved on average, after 212 generations. On average, the network consists of a hidden node and 5 connections. Compared to the evaluations without cart position discount, the evaluations with cart position discount requires significantly more generations to achieve the same fitness threshold.

Unlike previous evaluations without cart position discount, the input node for 'cart position' has high weights or significance. This is comprehensible as the deviation of the cart position from the initialised position is discounted from the reward. The network is required to minimise the cart position deviation while keeping the pole balanced to achieve the fitness threshold. However, both 'pole angular velocity' and 'pole angle' still have the higher significance than the 'cart position' as the reward for keeping the pole balanced is higher for this task. The distributions of data points are shown in **Appendix E**.

5.4 NEAT evaluation on Robotic control task

For the evaluation of the NEAT algorithm on a robotic control task, various hyper parameter settings are experimented such as probability of adding connection or node, bias mutation rate and weight mutation rate. The population is set as 256 and each neural network is initialised as a fully connected network without any hidden nodes as the task is a complex reinforcement learning problem. It is expected to require some hidden nodes to solve the task. The initial NEAT configuration is shown in **Appendix F**.

At first, the evaluation was conducted with conservative parameter settings, that are low mutation rates and small probabilities to add nodes or connections. As a result, it did not yield successful results. With the slow evolving rate, the average fitness value showed marginal improvement over many generations. Without improving the fitness value, the species reached maximum stagnation value and became extinct.

Secondly, the evaluation was conducted with radical parameter settings that are relatively high mutation rate and probabilities to add new nodes or connections while low probabilities of deleting existing nodes or connections. This resulted in increased improvement of fitness value and it was less susceptible for species to reach maximum stagnation value to become extinct. Although this showed improved results, it had taken many generations to arrive at a sufficient solution. Typically, the best performing networks had more than 30 hidden nodes and 200 connections. Increased network size required more computation and memory resources, which contributed to the extended duration of the evaluation.

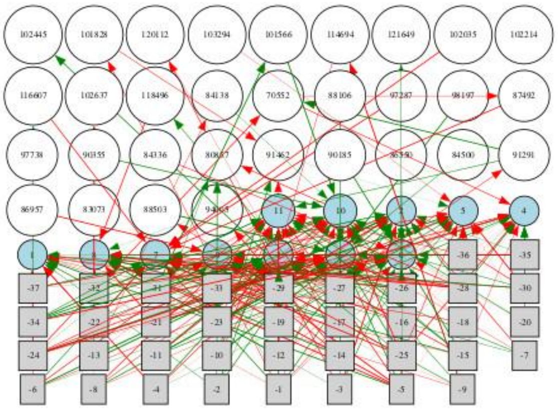


Figure 8. Best neural network topology with 30 hidden nodes and 254 connections. Square boxes depict input nodes, white circle depicts the hidden nodes, blue circle depicts the output nodes and arrows depict the connections.

Though it did not arrive at optimal solution within the reasonable amount of time, the best performing neural networks showed efficient actuation of the robot. It had learned to differentiate the timing of actuating each leg to travel forward effectively while minimising the actuation of the motors. The robot is actuated in a way that the applied forces to the legs were not excessive to produce vertical or jumping motion, instead, only adequate forces were applied to propel the robot in horizontal direction without falling. The resulting movement was akin to how four legged animals such as dogs would walk or gallop in forward direction.

6. Conclusions

Initially, I was planning to implement the NEAT algorithm from scratch, however, I was underestimating the effort required to implement both the algorithm and the custom simulation environment. Therefore, I had chosen to use a publicly available NEAT algorithm implemented in Python and focused my effort on building the custom gym environment using a mujoco physics simulator.

With the best performing result, the robot was actuated to create movements akin to four legged animals walking in forward direction, however, it did not yield optimal results. This project demonstrates some limitations using an evolutionary computation to solve a complex reinforcement learning task. For such tasks, evolutionary computation often requires large computational resources. In this project, only one local computer was used, and as the result, it took a significant amount of time to evaluate. The result may differ if more computational resources are available for this project. The separate limitation of this project is that the algorithm tends to be stacked at the local minima. The evolution of species was susceptible to stagnation and often became extinct.

To improve the result of the NEAT algorithm for complex reinforcement tasks, larger computational resources can be employed. For example, the algorithm runtime can be sped up by utilising the accelerated computing with modern GPUs (Graphics Processing Units). Also, the algorithm can be implemented in compiled programming languages such as C or C++ for faster computation rather than implementing in Python programming language which is considerably slower. To mitigate the stagnation on the early evolution stage, the neural networks can be pre-seeded with sequences as example actions. By providing the example actions initially, the search could be steered towards a desirable direction, which could achieve a more efficient evolution process.

References

[1] Karl Sims. 1994. Evolving virtual creatures. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques, 15–22.

[2] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," in Evolutionary Computation, vol. 10, no. 2, pp. 99-127, June 2002, doi: 10.1162/106365602320169811.

[3] P. J. Angeline, G. M. Saunders and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," in IEEE Transactions on Neural Networks, vol. 5, no. 1, pp. 54-65, Jan. 1994, doi: 10.1109/72.265960.

[4] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In Proceedings of COGANN-92 International Workshop on Combinations of Genetic Algorithms and Neural Networks, 1992.

[5] Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J. R. et al., editors, Genetic Programming 1996: Proceedings of the First Annual Conference, pages 81–89, MIT Press, Cambridge, Massachusetts.

[6] Gomez, F. and Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In Dean, T., editor, Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, pages 1356–1361, Morgan Kaufmann, San Francisco, California.

[7] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. Genetic Programming and Evolvable Machines, 2007.

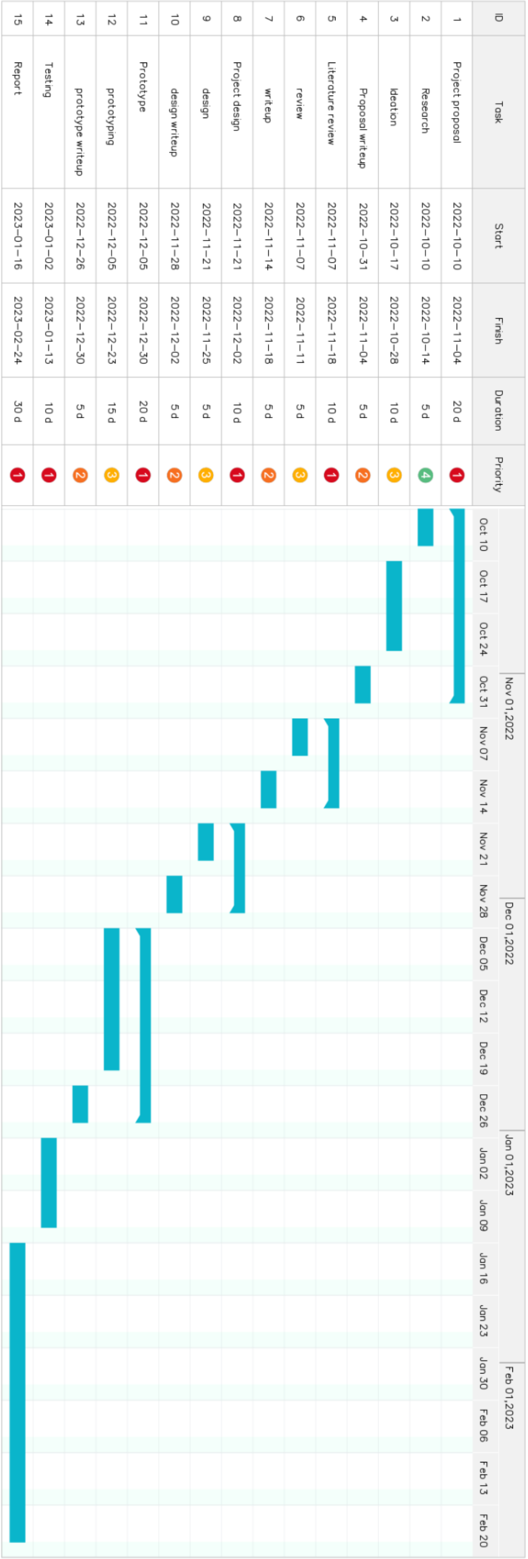
[8] Stanley, Kenneth O.; D'Ambrosio, David B.; and Gauci, Jason, "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks" (2009). Faculty Bibliography 2000s. 2178.

[9] McIntyre, A., Kallada, M., Miguel, C. G., Feher de Silva, C., & Netto, M. L. neat-python [Computer software]

Appendices

Appendix A

Work plan Gantt chart:



ID	Task	Start	Finish	Duration	Priority
1	Project proposal	2022–10–10	2022–11–04	20 d	1
2	Research	2022–10–10	2022–10–14	5 d	4
3	Ideation	2022–10–17	2022–10–28	10 d	3
4	Proposal writeup	2022–10–31	2022–11–04	5 d	2
5	Literature review	2022–11–07	2022–11–18	10 d	1
6	review	2022–11–07	2022–11–11	5 d	3
7	writeup	2022–11–14	2022–11–18	5 d	2
8	Project design	2022–11–21	2022–12–02	10 d	1
9	design	2022–11–21	2022–11–25	5 d	3
10	design writeup	2022–11–28	2022–12–02	5 d	2
11	Prototype	2022–12–05	2022–12–30	20 d	1
12	prototyping	2022–12–05	2022–12–23	15 d	3
13	prototype writeup	2022–12–26	2022–12–30	5 d	2
14	Testing	2023–01–02	2023–01–13	10 d	1
15	Report	2023–01–16	2023–02–24	30 d	1

Appendix B

OpenAI Cart Pole environment:

Action space	
0	Apply force to the left
1	Apply force to the right

Observation space			
Number	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-0.418 rad (-24°)	0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Termination conditions	
Cart position	The task is terminated if the cart position exceeds beyond the value range between -2.4 and 2.4.
Pole angle	The task is terminated if the pole angle exceeds the range between -0.2095 rad and 0.2095, or -12° and 12°.

Rewards
Rewards are accumulated for every step taken until the task is terminated either by reaching the end of the evaluation steps or one of the above termination conditions is met.

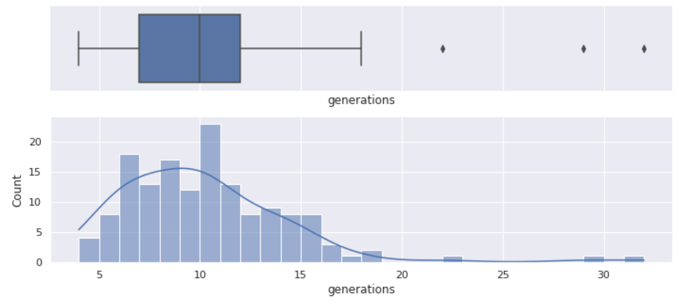
Appendix C

OpenAI Gym cart-pole - NEAT configuration parameters:

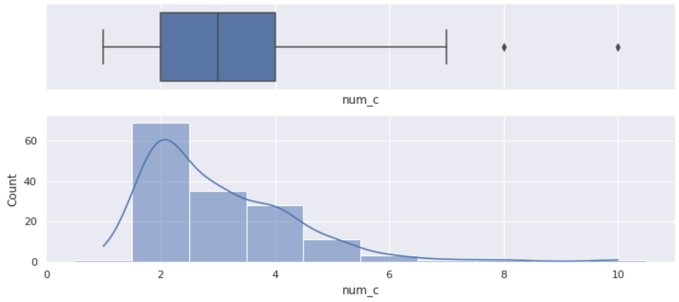
Parameter	Value
fitness_criterion	max
fitness_threshold	499.5
pop_size	150
reset_on_extinction	1
[DefaultGenome]	
num_inputs	4
num_hidden	0
num_outputs	1
initial_connection	Unconnected
feed_forward	True
compatibility_disjoint_coefficient	0.01
compatibility_weight_coefficient	0.6
conn_add_prob	0.05
conn_delete_prob	0.05
node_add_prob	0.05
node_delete_prob	0.05
activation_default	tanh
activation_options	tanh
activation_mutate_rate	0.0
aggregation_default	sum
aggregation_options	sum
aggregation_mutate_rate	0.0
bias_init_mean	0.0
bias_init_stdev	1.0
bias_replace_rate	0.01
bias_mutate_rate	0.01
bias_mutate_power	0.05
bias_max_value	30.0
bias_min_value	-30.0
response_init_mean	1.0
response_init_stdev	0.0
response_replace_rate	0.0
response_mutate_rate	0.0
response_mutate_power	0.0
response_max_value	30.0
response_min_value	-30.0
weight_init_type	Uniform
weight_max_value	10
weight_min_value	-10
weight_init_mean	0.0
weight_init_stdev	1.0
weight_mutate_rate	0.01
weight_replace_rate	0.01
weight_mutate_power	0.01
enabled_default	True
enabled_mutate_rate	0.01
[DefaultSpeciesSet]	
compatibility_threshold	2.0
[DefaultStagnation]	
species_fitness_func	Max
max_stagnation	30
[DefaultReproduction]	
elitism	5
survival_threshold	0.2

Appendix D

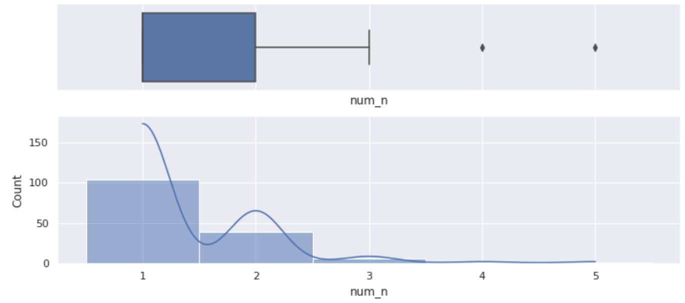
OpenAI Gym cart-pole task experiments run for 150 times:



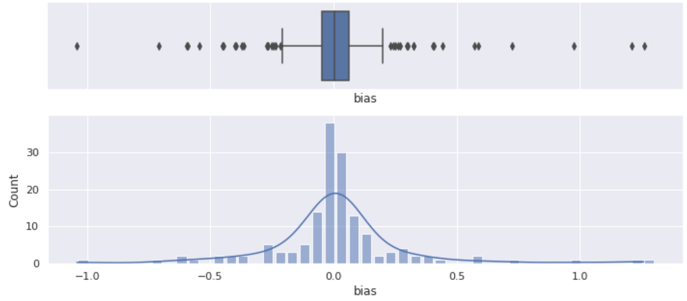
Distribution of the generations



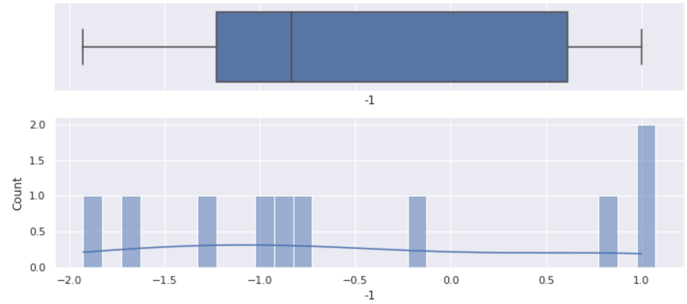
Distribution of number of connections



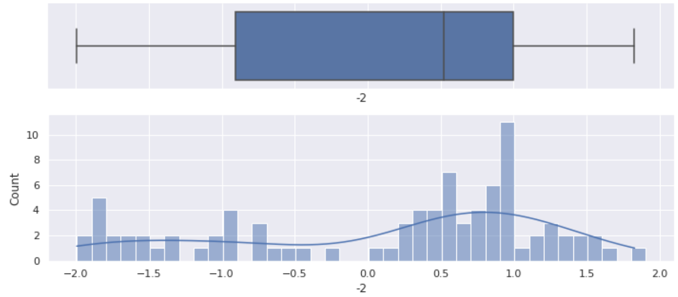
Distribution of number of nodes



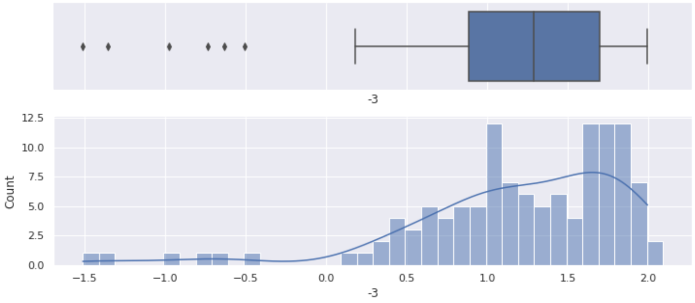
Distribution of the bias



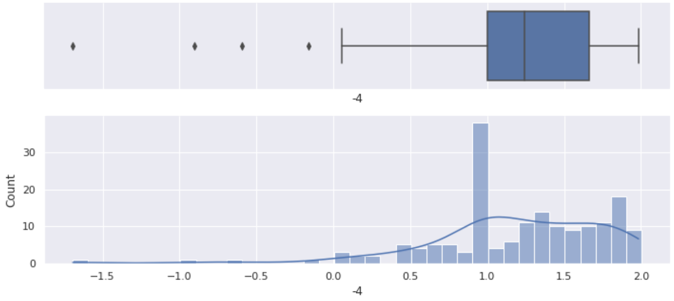
Distribution of the weight (-1)



Distribution of the weight (-2)



Distribution of the weight (-3)

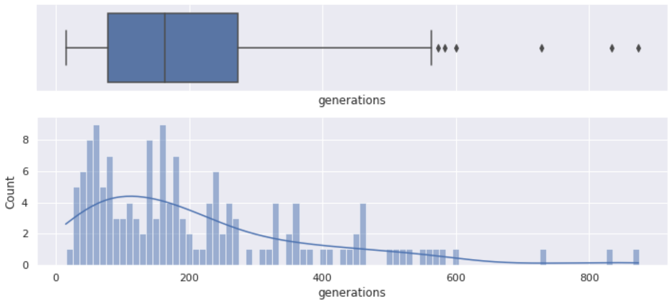


Distribution of the weight (-4)

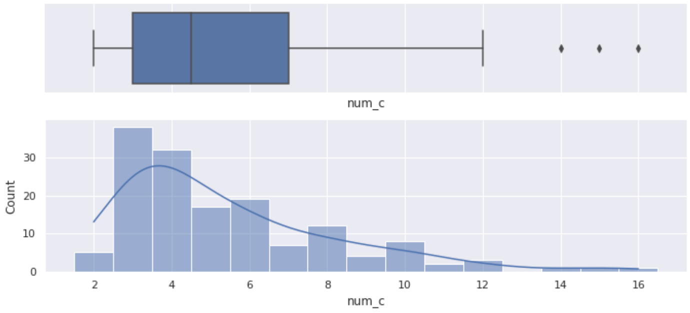
Appendix E

OpenAI Gym cart-pole task with cart position discount experiments run for 150 times:

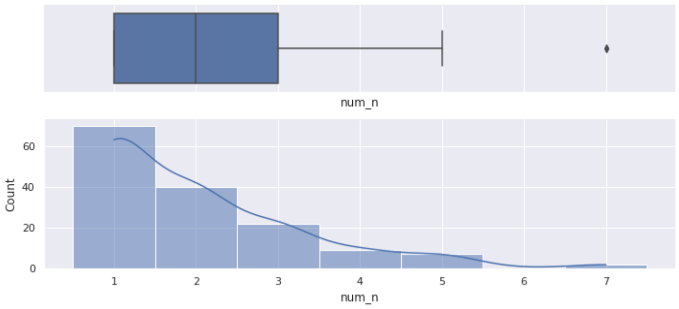
Distribution of the generations



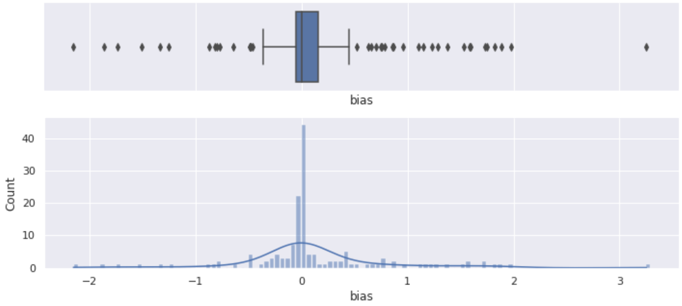
Distribution of number of connections



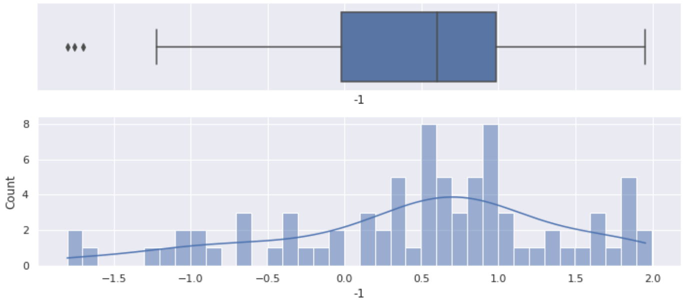
Distribution of number of nodes



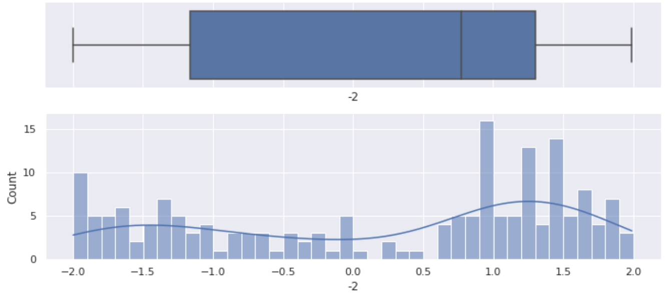
Distribution of the bias



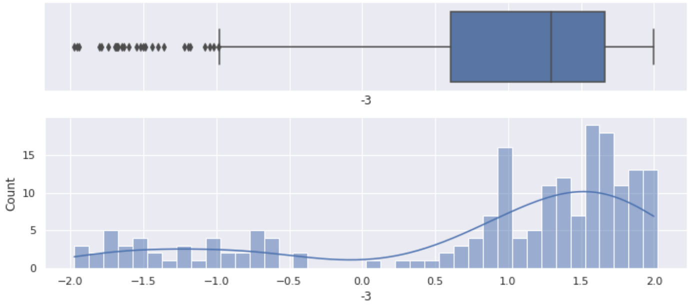
Distribution of the weight (-1)



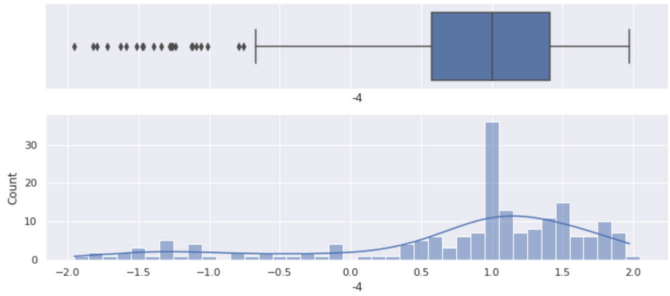
Distribution of the weight (-2)



Distribution of the weight (-3)



Distribution of the weight (-4)



Appendix F

Mujoco - NEAT configuration parameters:

Parameter	Value
fitness_criterion	max
no_fitness_termination	True
pop_size	256
reset_on_extinction	1
[DefaultGenome]	
num_inputs	37
num_hidden	0
num_outputs	12
initial_connection	full_direct
feed_forward	True
compatibility_disjoint_coefficient	0.01
compatibility_weight_coefficient	0.6
conn_add_prob	0.1
conn_delete_prob	0.1
node_add_prob	0.1
node_delete_prob	0.1
activation_default	tanh
activation_options	tanh
activation_mutate_rate	0.0
aggregation_default	sum
aggregation_options	sum
aggregation_mutate_rate	0.0
bias_init_mean	0.0
bias_init_stdev	1.0
bias_replace_rate	0.1
bias_mutate_rate	0.1
bias_mutate_power	0.1
bias_max_value	30.0
bias_min_value	-30.0
response_init_mean	1.0
response_init_stdev	0.0
response_replace_rate	0.0
response_mutate_rate	0.0
response_mutate_power	0.0
response_max_value	30.0
response_min_value	-30.0
weight_max_value	30
weight_min_value	-30
weight_init_mean	0.0
weight_init_stdev	1.0
weight_mutate_rate	0.1
weight_replace_rate	0.1
weight_mutate_power	0.1
enabled_default	True
enabled_mutate_rate	0.01
[DefaultSpeciesSet]	
compatibility_threshold	4
[DefaultStagnation]	
species_fitness_func	Max
max_stagnation	30
[DefaultReproduction]	
elitism	5
survival_threshold	0.2