Use of NeatAI in simple movement controllers

Implementation of a modifed NeatAI algorythm for the development of 2D bipedal movement controllers with minimal network dimension

AE4350: Bio-inspired Intelligence and learning for Aerospace Applications

Diogo Serra

Use of NeatAI in simple movement controllers

Implementation of a modifed NeatAI algorythm for the development of 2D bipedal movement controllers with minimal network dimension

by

Diogo Serra

Student Name	Student Number
Diogo Serra	6067786

Link to Github Repository

`https://github.com/GitManDS/Simple_NeatAI_for_Humanoid_Walking`

Link to side report with results in .gif format

`https://github.com/GitManDS/Simple_NeatAI_for_Humanoid_Walking/blob/main/report.md`

Instructor:	E Van Kampen, G.C.H.E. de Croon
Project Duration:	2 Months
Faculty:	Faculty of Aerospace Engineering, Delft

Cover:	Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TUDelft

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
TWEANN	Topology and Weight Evolving Artificial Neural Networks
NEAT	NeuralEvolution of Augmented Topologies
URDF	Universal Robotic Description Format
NN	Neural Network

Symbols

Symbol	Definition	Unit
E	excess genes	[-]
D	disjoint genes	[-]
$\bar{W}$	average weight difference	[-]
N	Number of connection genes	[-]
δ	Compatibility distance	[-]

1 Introduction

The development of a physics based movement controller is itself quite computationally intensive and, in general, can be hard to develop as a human body requires syntonic movement of all body parts in order to achieve balance, even in more unstable regimes such as walking and running. Moreover, due to this syntonic nature, defining clear feedback systems is highly complex and possibly too restricting. Instead, an approach using Neuralevolution of genetic algorithms is taken, through which a neural network(NN) can evolve through the evolution of one or more genetic algorithms. There are several implementations of Neuralevolution such as Symbiotic, Adaptive Neuro-Evolution(SANE) and Enforced Sub-Populations Method(ESP), a lot of which have shown a great promise for evolutionary robotics [1][2].

In traditional Neuralevolution of genetic algorithms, it is often the case that the dimension is arbitrarily and subjectively set by the developer *a priori*. This method of network encoding (direct encoding) works fairly well, however, some research has shown that by including the structure of the network into the evolutionary process using cellular encoding, it's possible to improve learning speeds significantly and improve learning efficiency[2][3][1]. Reducing the size of the networks is of interest for real world applications of this technology, for instance, a leg exoskeleton might become impractical in size if it needs to dedicate a significant volume portion to a dedicated electronics section just to compute through a high dimension network several times per second.

The NEAT algorithm, by K. O. Stanley and R. Miikkulainen[4][5] was developed around the idea that not only is it possible to improve learning speeds of networks with the use of Topology and Weight Evolving Artificial Neural Networks(TWEANN) but it's also possible to do so whilst ensuring minimal network dimension. The algorithm adopts solutions to common problems in the crossover and mutation processes between networks of different structures. This algorithm has since been found very reliable for autonomous robotics and agent controls in video games, such as the semi-famous *MarI/O* NEAT algorithm used to complete one of *Super Mario World* levels[6][7].

In this report, the methodology behind the development and solution to a 2D bipedal controller is discussed. The use of the NEAT algorithm, paired with some minor modifications, aims to achieve a controller solution that balances out the high computational cost of physics simulations by leveraging the high efficiency and learning speed of TWEANNs and of the methods employed by the NEAT algorithm for network evolution, which should lead to a minimal dimension controller. The final solutions are then analysed and the controllers are observed to verify if a walking pattern was achieved.

2 Model Implementation

The project is divided into algorithm and test environment. The test environment, a physics simulator, is a python built shell of the publicly available PyBullet package[1]. The shell is simply used as a way to easily interface the NeatAI algorithm with the simulation setup, computations and results. The networks control a human like robot which is a significantly modified version of a standard model from the pybullet package, modified using the Universal Robotic description format(URDF) syntax. The original model was modified to include 6 joints (waist-leg joint, knee joint and ankle joint (2x for each leg) and only allow movement in the 2D plane, as well as other cosmetic changes.

In order to properly understand and manipulate the variables involved in NEAT, the algorithm was coded from the ground up using python's standard libraries and common or personal interpretations of the original papers [4][5] on this subject, which explain the techniques used to tackle problems such as crossover and initialisation in detail. This chapter will briefly present all of these mechanisms and how they were implemented/interpreted by the author of this report.

2.0.1. Genetic encoding and initialization

Because the structure is always changing, the layers of neurons can no longer be treated as matrices and the structure will need to be enconded differently. The Neat algorithm uses direct encoding to keep track of the structure (nodes and connections) and their characteristics (weight, enabled/disabled[2], in and out indexes,[etc]). With this encoding, an entire set of neurons and connections, including the weights can be represented using the Genotype (Figure 2.1). The genotype has all the information needed to form the phenotype which completely describes the entire network.

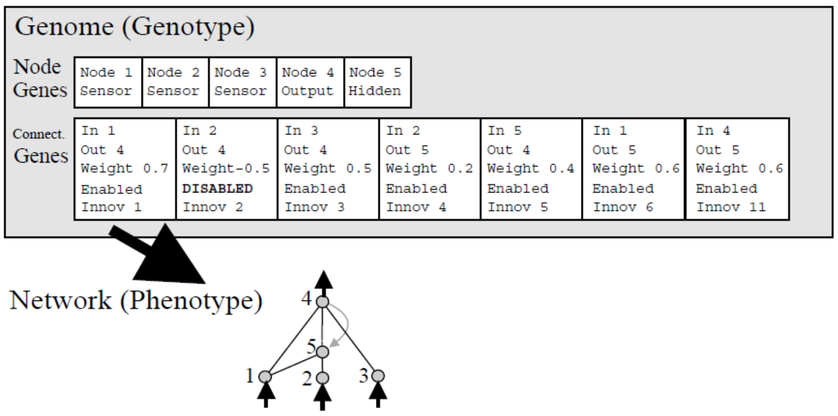
The connection genes in the genotype are placed in the order they appear in and have a extra parameter unique to the NEAT algorithm which is the innovation number. This number is best explained as a number that identifies structural changes in the topology. Each connection that is unique in the nodes it connects to has its own innovation number, 2 different networks that have the same connection between the same nodes will have the same innovation number. Although this innovation number can be any number that's unique, it was chosen to use a counter and increment that counter to get a new unique value whenever a new innovation is created. The use of this number will become clearer in subsection 2.0.3.

One important notice is that there's no storage for the bias of each node in this specific implementation of NEAT. This decision was made not only because the original authors of [4] don't mention any protocol for crossover of this kind of data but also because adding bias to nodes would increase the computational cost of the solution.

Each network is part of the population which consists of several networks, each one is unique inside the population with their own typologies and weight distribution. In order to preserve one of the main ideas

[1] `https://pybullet.org/wordpress/`

[2] If a connection is disabled, it will count as non existent when calculating the output but the connection remains present in the phenotype in case it later gets re-activated



Node 1 Sensor	Node 2 Sensor	Node 3 Sensor	Node 4 Output	Node 5 Hidden		
In 1	In 2	In 3	In 2	In 5	In 1	In 4
Out 4	Out 4	Out 4	Out 5	Out 4	Out 5	Out 5
Weight 0.7	Weight-0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6	Weight 0.6
Enabled	**DISABLED**	Enabled	Enabled	Enabled	Enabled	Enabled
Innov 1	Innov 2	Innov 3	Innov 4	Innov 5	Innov 6	Innov 11

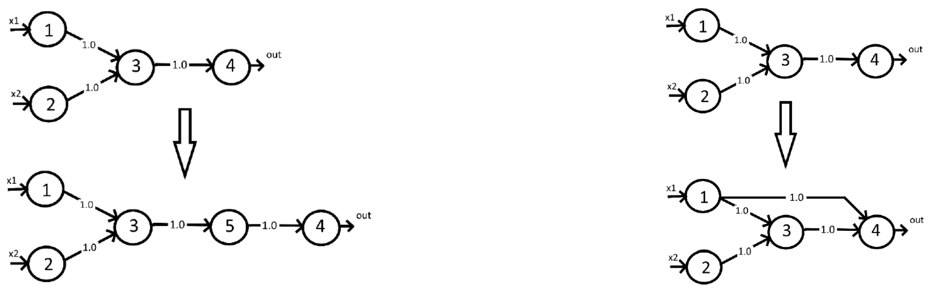
Figure 2.1: Illustration of a genotype and phenotype as shown in [4, Fig 2.]

of NEAT, which is to ensure minimal network dimension, all networks in the population are generated with no hidden nodes, with connections between every input and output node using random weights between -1 and 1 and with all the connections enabled.

2.0.2. Genotype mutations

One of the features of the NEAT algorithm is the possibility of a network to mutate in a random matter in order to introduce genetic diversity. For this implementation is was chosen to mutate each network a random number of times per generation, every generation, with a random mutation type. Of the mutation types, the following are implemented:

- **Update weight**: A connection gets its weight updated. This was implemented such that it could only update up to $\pm 50\%$ of the original value. This mutation is never a structural innovation.
- **Update status**: A connection's status (Enabled/Disabled) gets flipped. This mutation is never a structural innovation.



(a) Add node mutation

(b) Add connection mutation

Figure 2.2: Illustration of structural innovation mutations from [8, Fig 2, 3]

- **Add Node**: If there isn't a connection, a node gets added between the selected nodes and 2 connections are made to complete the connection(one into the new node and one out of the new node). If there is a connection already between the selected nodes, the connection's status is set to disabled, the new node gets created and 2 new connections get created in and out of the new node, however the connection going into the new node inherits the value of the weight of the old disabled connection. This mutation, if it hasn't happened before, will constitute a structural innovation.
- **Add connection**: If there isn't an existent connection between the selected nodes, a new one is created. This counts as a structural innovation except if the connection already existed and it was disabled. If this is the case, the status gets flipped and it does not count as a structural innovation.

The structural innovations of all networks are registered in a central structural innovations list to check and manage the innovation numbers.

One last mutation that isn't present in the original paper but has been used in previous papers[8] to reduce the training times for this algorithm is the remove node mutation.

- **remove node**: If the node exists, remove it and all the connections going in and out of the node. This does not count as a structural innovation.

Without this last mutation, a population in insufficient numbers might see one network successfully grow in size through normal successful and justified structural innovations. Because, through generation, crossover mechanisms tend to "lock" the structure of the most successful network inside a species, a high dimension network that grows in size and in score will influence the rest of the population to do the same, even in cases when another lower dimensional structure could achieve the same score increase, had the random process been slightly different. This "one-way" evolution might then result in increased structural dimensions.

The remove node mutation also does not try to compensate for the lost node in any form as it is not supposed to. The article on this mutation [8] noted correctly that the removal of a node often leads to the substantial loss of information, which most often leads to a significant drop in score. If this is the case then the network affected gets eliminated soon enough by the mechanism employed to keep the population under a certain number. If however removing a node has no significant implications to the score or increases the score then the network keeps competing, eventually being put on a different species and introducing more diversity

2.0.3. Crossover

In TWEANNs, crossover is not trivial due to the way different topologies can achieve the same result, that is, if 2 topologies are significantly different but give the same output per input, it's very likely that the combination of both topologies does not yield a similar result and much less so the same result. Additionally, the typical way to determine which networks crossover is by ordering networks by their score in the last simulation round. However one very important note made by [4] is that adding a node or a connection often leads to a significant performance drop. A population wide competition would negatively impact networks with recent structural innovations, leading to a evolution stagnation.

The NEAT algorithm employs several solutions to these problems which are implemented in this project. Firstly, the crossover uses a dominant-recessive mechanism, where the network with the higher score defines the structure of the offspring network. This means that any disjoint or excess genes in the recessive network do not transfer to the final offspring, as can be seen in Figure 2.3.

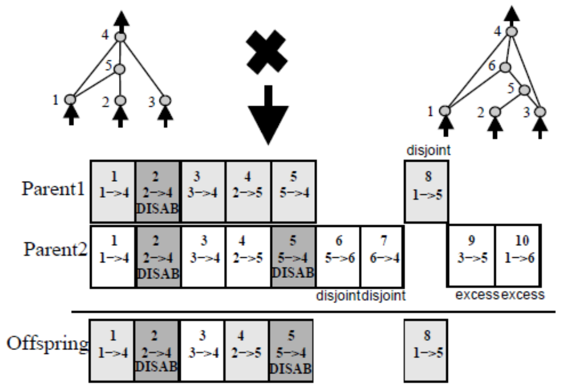


Figure 2.3: Crossover process between parent1 (dominant) and parent2 (recessive). Adapted illustration from [4, Fig. 4]

In turn, the innovation number is used to see if there are matching connection genes, if the innovation numbers of 2 connections match then the connection gene is present in both networks and the child inherits either from the recessive or dominant parent in a random manner.

In order to protect structural innovations, the idea of species is introduced. Networks are grouped together inside the population in their own species according to how structurally similar each network is to the other networks. By doing this, it is possible to restrict crossover to individual species and as such networks only need to compete inside their species for the dominant spot. This distance is called the "compatibility distance" δ and is calculated using the number of excess genes(E), number of disjoint genes(D), average weight different($\bar{W}$) and number of connection genes(N):

$$\delta = c_1 \frac{E}{N} + c_2 \frac{D}{N} + c_3 \bar{W} \tag{2.1}$$

After every mutation round, the mutations are reorganised and the δ is calculated between every network, if a network finds a species where δ is below a threshold then it is placed in that species, else a new species is created and the network is placed there. A fine tuning of c_1, c_2, c_3 and δ allows for different behaviours of this implementation.

This crossover process is done until the maximum number of offspring is hit, always with one dominant network per specie doing crossover with the several recessive networks. The remaining networks that don't crossover get eliminated. The number of offspring per species is determined according to the average network scores and linearly scaled to match the maximum offspring limit.

One last measure implemented by the NEAT Algorithm is the *explicit fitness sharing*[9] which aims to reduce the size of big populations by normalising the score of a network by the number of networks in the specie it belongs to. It is likely that this is intended for a population starting from a small number of networks or a single network, however in this implementation, *explicit fitness sharing* is not used as the author of this project found it to be highly aggressive on the highest score (and consequently) fastest growing species when starting with a large, diverse population, which stunts the fine tuning of new innovations. Instead, the population size is regulated through carefully controlled parameters such as max population number and max offspring number to ensure a species cannot get too big (see subsection 2.0.4 for a better explanation of these parameters).

2.0.4. Population management

In order to manage the evolution process and population several parameters and mechanisms are implemented. This algorithm implements a population size control by defining the max number of networks in the population, after which it will delete the worst performing species until the maximum limit is again met. Additionally it also implements a maximum offspring mechanism that will effectively limit the maximum size of a species by deleting all the networks in a network that won't participate in the crossover. This way it's possible to ensure a somewhat maximum fixed fraction of the total population each species can occupy. Additionally, there are parameters specific to the a network that restrict the number of mutations they can be subject to per round of mutations and control how the mutations are done.

2.0.5. TWEEAN output calculations

Due to the lack of a definitive structure to the networks, this algorithm cannot use the typical matrix calculations used in normal fixed topology network. The alternative to matrix calculations is going node by node and adding all the input contributions until eventually reaching the output nodes. Two predictable problems arise from this method, the first of which is the possibility of a connection loop. Because the values of the nodes are calculated sequentially, it is possible for a node (A) can connect to a node (B) which can then connect to a node (C) which ends connecting to node (A), which results in a never ending loop in the calculation process. To avoid this, whenever a new connection or node is added, a recursive search function is used to detect these loops and avoid them.

The other problem is in the order of nodes, despite commonly being treated as such, the hidden layer cannot be treated as a blackbox for calculation purposes and needs to be ordered into actual layers according to which nodes input and output to each other, else the calculation would run into hidden nodes that require the value of other hidden nodes that haven't been resolved yet. To solve this a mechanism is in order to layer the hidden nodes, as can be seen in Figure 2.4b.

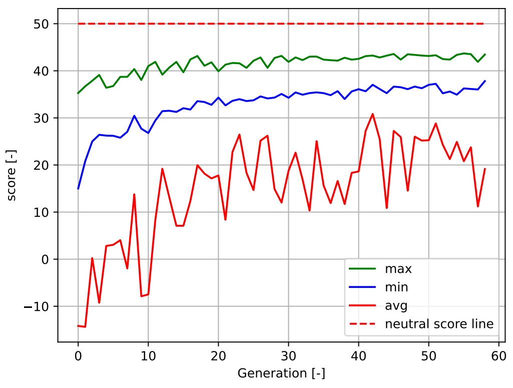
Finally, a *tanh* activation function is applied to every node in order to remove the linearity between the output and input. Also it is modified such that the node values (approximately) vary between -1 and 1 in similarity to the inputs:

$$output = tanh(output) * 2 \tag{2.2}$$

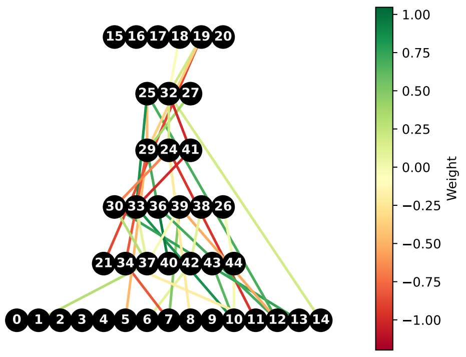
2.1. Performance evaluation

A simulation was done to verify the algorithm could find and converge to a solution successfully on set of simple training rules. For the training of this robot it was chosen to evaluate the x rotation of the robot (normalised between -1 and 1) and the difference between the current torso height and initial torso height. These 2 qualities were evaluated by integrating them over the simulation steps and normalising by the number of steps, which has the additional advantage of rewarding networks by number of steps spent upright). Additionally, the network was penalised for any y-direction velocity.

The initial population count was set to 40, with a max network count set to 40, a maximum offspring count of 8 and c_1, c_2, c_3 = [1.5,1.5,0.4] with δ_{max} = 0.25. The model has 15 inputs (torso 2D position(2), y-velocity(1), joint positions(6) and velocity(6)) and 6 outputs (6 target joint positions)



(a) Score evolution per generation iterated for a simple standing evaluation



(b) Generation 40 best phenotype with enabled(continuous) and disabled(traced) connections drawn and coloured according to the weight they hold (direct input-output connections are hidden for readability

Figure 2.4: Results for the performance evaluation

The simulation converged to a final solution after 59 iterations and the visual results can be found in the report.md file. The score evolution, shown in Figure 2.4a shows that the algorithm did manage to converge to a solution using Neuralevolution, with feasible solutions near generation 40. The final results show that, despite the high frequency of joint movement (which wasn't penalised in this training), the network that initially would fall down and move randomly eventually evolved and learned to balance itself whilst not moving around much.

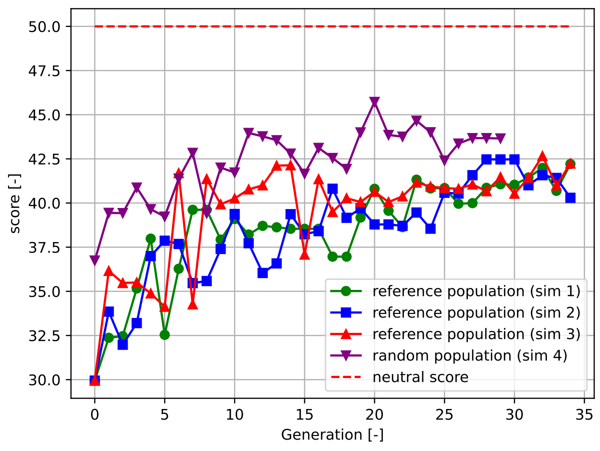
Also important to note from Figure 2.4 is that the algorithm is, as intended, prioritising the stability of the top species over that of newer, lower performing species which is clearly seen in the variations in the minimum and maximum scores. The population average also does not fluctuate significantly which indicates that previous beneficial structural innovations are successfully maintained in the population, ensuring a good genotype base for future mutation rounds.

Lastly, the simulation was allowed to run until generation 59 although it's possible to see that convergence is met at generation $\approx$ 40. The last 20 generations lead to random "noise" mutations that unnecessarily increases the size of the phenotype.

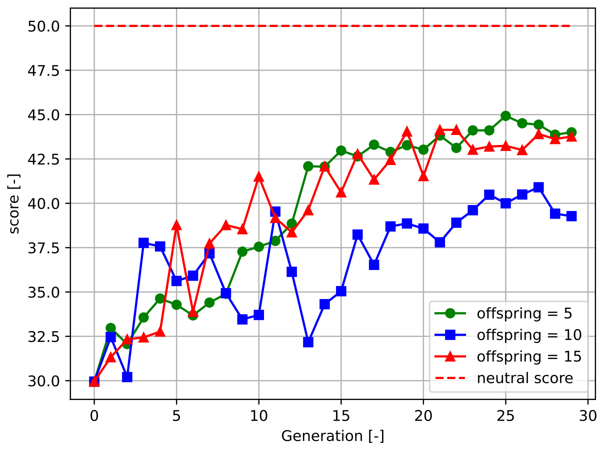
2.2. Critical Algorithm parameter adjustment

Most of the factors described in the implementation section have a significant impact on the neuroevolution process and have been tweaked several times in order to achieve the best training efficiency. In order to adjust them properly, several simulations under similar conditions have been performed.

Before the parameter analysis, it's important to recognize the chaotic nature of the algorithm which affects how much a final solution is dictated by these parameters. Doing the same 3 simulations starting off from the exact same reference population results in unique evolution patterns with very similiar solution convergence as seen in Figure 2.5a. However, starting from a random population (and thus a different initial solution) results in significantly different results. For the purpose of this section, all simulations are run off of the same reference population.



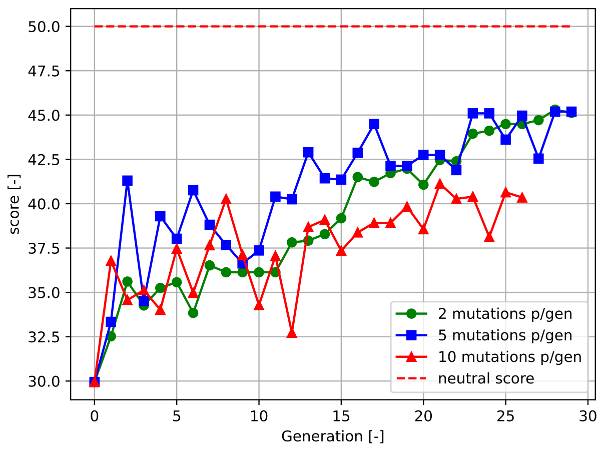
(a) Control group sensitivity study



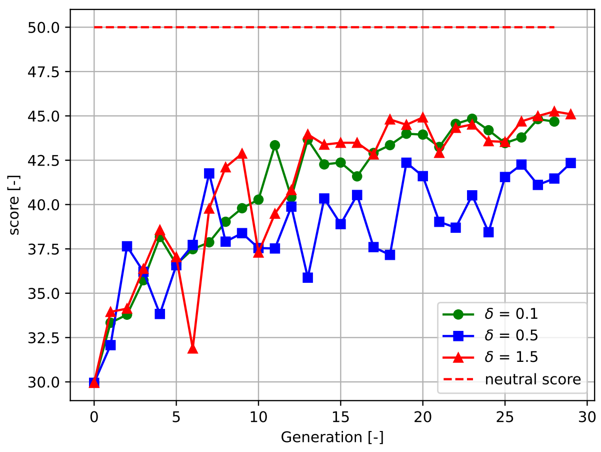
(b) Simulations with increasing values of max specie offspring

Figure 2.5: Solution sensitivity with critical population parameters

Firstly, in terms of population management, it was seen that the max number of offspring and max number of networks parameters critically defined how the population evolved through generations. As seen in Figure 2.5b, a suitable value for the max offspring per species is around 5, more than that and the population starts to stagnate as there is less space for species and thus less diversity and less opportunities for structural innovations. Also interesting is that at a too high of a max offspring parameter value, the species mechanism becomes irrelevant and the fluctuations that same mechanism tries to avoid are much more pronounced throughout the generations as new innovations struggle to re-adjust their weights.



(a) Simulations with increasing values of max mutations per generation



(b) Simulations with increasing values of threshold comparability distance δ_{max}

Figure 2.6: Solution sensitivity with critical network parameters

In terms of network parameters, the critical parameter in the mutation process is the mutations per generation parameter. As seen in Figure 2.6a, a lower value gives a more stable convergence but can predictably be a slower overall method and a value that's too high will lead to complete instability and worst convergence. It was chosen to adopt a value close to 5 mutations per generation as it was found to be a good balance between convergence stability and convergence time. In terms of crossover, the δ_{max} is the dominant parameter and from Figure 2.6b it can be argued that keeping this value low (around 0-0.4) allows for the proper conservation of innovations and thus a better and more swift convergence over the same generations.

Other parameters such as the max number of networks and starting brain count were chosen in order to maximise diversity without compromising computational cost too much.

3 Bipedal controller training

3.1. Training and results

Training the algorithm to walk is a challenge since walking requires the network to purposely lose balance (forwards) and use the legs in a coordinated manner to maintain temporary balance. The first incentive given to the network is to move forward by increasing its y coordinate, however, due to aforementioned stability problems, the network would most times simply converge to the same solution as of section 2.1 but using vibrations in the leg joints to move forwards incrementally. In order to achieve more human movement, the network was also given points for cycling each leg intermittently and severely penalised for have both legs in the same place, not switching legs every 150 simulation steps and falling upside down.

The same algorithm parameters that were used for the performance evaluation were again used for this training with the exception of the max mutations per gen parameter, that was changed to 1 in order to correctly preserve the sensitive leg coordination innovations. Several simulations were then run under the same rules, the results of which can be observed on Figure 3.1 and on the report.md file.

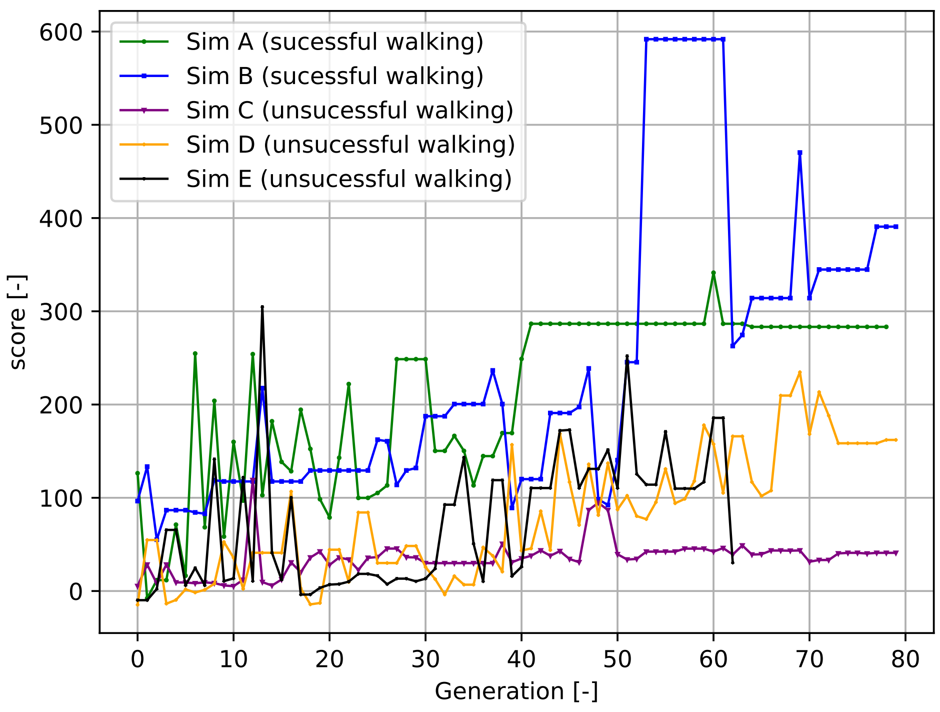
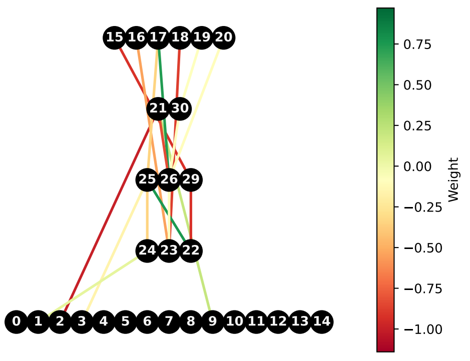


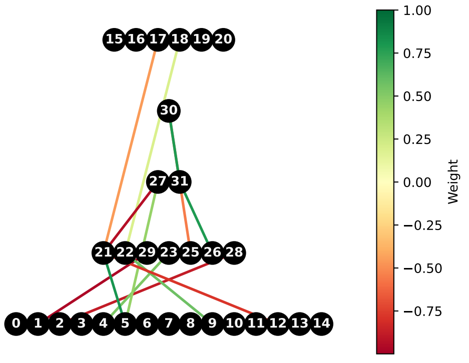
Figure 3.1: Best network score per generation for 5 different results of the same simulation

3.2. Results analysis

In terms of results, one immediate conclusion from Figure 3.1 is that networks with successful walking patterns have a higher end score than those who don't achieve a walking pattern, which makes sense as the objective function will give far more points to networks that travel the farthest. Additionally, even though the number of mutations was set to 1, it can be seen that most networks still display significant variance of results over generations as compared to the same situation in Figure 2.6a. This can be explained by the higher impact a single mutation has on a smaller network (compare Figure 3.2a with Figure 2.4b), which in conjunction with the delicate system required to coordinate all joints correctly makes each mutation all the more significant. In other words, this parameter change permitted a more detailed development of each network's structure, which in turn increased the instability of the evolution process much like a high value for max mutations per gen parameter would.



(a) Phenotype of the best network of sim B (gen=80)



(b) Phenotype of the best network of sim A (gen=80)

Figure 3.2: Phenotypes of successfully walking networks (direct input-output connections are hidden for readability)

The final network's dimension of Sim A and Sim B can be found in Figure 3.2. Despite the small dimension, both networks managed to display self balance, leg coordination and distance travelled in a "humanoid" way. However, both network's performance are visibly limited by their structural size when observing their movement. Sim B, which is considered the most successful simulation, walks by launching the left leg forward, anchoring it (Left leg) on the floor and pushing the right leg forwards and a bit beyond the left leg to give extra momentum. Sim A does much the same, however it relies less on the extra momentum after anchoring one leg and more on repetition of the cycle to move itself. Both these movement styles have the dependency on one leg for balance and the other for support in common which strays from the human like walking pattern humans exhibit. Although the minimal dimension of the network could explain some of this behaviour, a more just and solid explanation is the model itself lacks torso bending movement and arms/neck joints and links, all of which aid the humanoid body in equilibrium and would remove the need for one of the legs to hold that function. One interesting consequence of this behaviour is that it is possible to achieve an alternative solution that results in a quite remarkably stable leg skipping controller, not discussed in detail here but present in the report.md file.

In terms of body equilibrium, both networks display significant use of the upper leg to torso joints in order to move the torso back and forwards according to leg movement. Although this makes the torso fluctuate violently, it also allows for rapid leg movement which allows for a higher distance travelled for both simulations (launching the leg forward forces the torso backwards and the network predicts and prepares for it). This equilibrium unfortunately comes to an end at the end of the simulation, as the network discovered that diving forward near the time limit of the simulation always gave it extra points.

The observations made so far serve to understand that the algorithm managed to properly obey to the objectives and training rules set at the start of the simulation and successfully managed to evolve networks that despite showing very rough walking mechanisms, require very little computational power and structural dimension.

4 Conclusion

It was shown that the implementation of the NEAT algorithm with some modifications managed to achieve a good solution convergence under 100 generations whilst maintaining low computational times. The Neuralevolutionary process implemented showed good solution consistency on simpler tasks and allowed for precise control of the evolution process through the variation of specific NEAT parameters.

The mechanisms employed by the developers of the NEAT algorithm were tested and their effects were observed through the evolutionary process of the populations. Through a series of simulations it was found that the evolution process can be tuned to account for a more precise or a faster evolution by adjusting the mutation rate, aswell as tuned for what type of mutation is more untactful, how fast a network grows and how diverse a population can be, among others, through the careful adjustment of each parameter that describes the model.

This genetic algorithm managed to find fairly good solutions for movement controller, considering the very minimal and limiting structural dimension. These solutions are a direct reflection of the minimal approach taken since the resulting networks prioritise developing the least amount of movements needed for a rudimentary movement strategy, which does not include effective movement of the knee and ankle joint but it includes coordination of both these joints in aiding the waist-torso joint to move freely. As this is a more complex set of training rules and because the minimal structure of the networks makes them much more susceptible to lose critical information through normal mutations each generation, only a few of the trained populations actually resulted in a successfully walking network. Furthermore, due to the small size, these networks are not well trained for variations in the simulation conditions and as such they will most likely perform significantly worse if the starting position is slightly changed or a small surface irregularity is introduced.

Nevertheless, these results show that these solutions are possible and that controllers of this small size are easy to solve for and perform fairly well. It's possible to improve on these results by testing with a more rough terrain and applying random external forces on the network controlled robot in order to force it to better balance itself outside the perfect conditions of a simulator. Additionally, the training rules can be further reinforced with restrictions such as bend movement and knee movement based on human movement patterns in order to achieve more ”natural” looking movement.

Finally, an important improvement to this work could be the use OF a model with proper arms and torso joints. The addition of these joints could help the robot achieve better results under roughly the same number of iterations, as it is reasoned that currently the robot is significantly hampered by its physical limitations. The addition of these extra joints could also allow for the movement in 3D, although the training rules would have likely had to change significantly.

References

[1] F. J. Gomez and R. Miikkulainen, “Solving non-markovian control tasks with neuroevolution,” in *Proceedings of the 1999 International Joint Conference on Artificial Intelligence*, 1999.

[2] F. Gruau, D. Whitley, and L. Pyeatt, “A comparison between cellular encoding and direct encoding for genetic neural networks,” in *Proceedings of the First Annual Conference*, MIT Press, Cambridge, Massachusetts, 1996.

[3] X. Yao, “Evolving artificial neural networks,” in *Proceedings of the IEEE*, 1999.

[4] K. O. Stanley and R. Miikkulainen, “Evolving neural networks through augmenting topologies,” *Evolutionary Computation*, vol. 10, Jun. 2002. DOI: `https://doi.org/10.1162/106365602320169811`.

[5] K. O. Stanley and R. Miikkulainen, “Efficient evolution of neural network topologies,” in *Proceedings of the 2002 Congress on Evolutionary Computation*, Jul. 2002. DOI: `https://doi.org/10.1109/CEC.2002.1004508`.

[6] P. Hallaj, *Neat: The evolutionary path to smarter ai solutions*, `https://medium.com/@pouyahallaj/neat-the-evolutionary-path-to-smarter-ai-solutions-a74354d1c7ed`, [Accessed 12-06-2024], 2023.

[7] S. Bling. “Mari/o - machine learning for video games,” Youtube. (2015), [Online]. Available: `https://www.youtube.com/watch?v=qv6UVOQ0F44`.

[8] G. Oleg and N. Alexander, “Mutation control in neat for customizable neural networks,” vol. 2865, 2019. [Online]. Available: `https://ceur-ws.org/Vol-2856/paper5.pdf`.

[9] D. E. ”Goldberg and J. Richardson, “Genetic algorithms with sharing for multimodal function optimization,” in *Proceedings of the Second International Conference on Genetic Algorithms*, 1987.