Teaching Robots to Play Video Games; A study into the effectiveness of population based TWEANN reinforcement learning techniques

Jack Wilson Moorin  
*Faculty of Computing, Engineering and Media*  
De Montfort UniversityLeicester, United Kingdom  
P17190172

*Abstract* — The field of ‘Soft Computing’ concerns the design and implementation of computerized mathematical models, often a neural network, which through data or observation can learn to perform a task or function. ‘Reinforcement Learning’ is one such method, training the network to make actions within a defined environment in order to maximize the output of a value function measuring its performance [1]. In recent years, the capability of traditional approaches of training a neural network have been increased with the introduction of topology and weight evolving artificial neural networks which attempt to optimize the topology of a network as well as its weights. This paper, submitted as the thesis component of a Masters’ Degree course undertaken by the author in Intelligent Systems and Robotics at De Montfort University, documents the attempt to improve upon the effectiveness of two existing population-based TWEANN reinforcement learning techniques through the evaluation and hybridization of two of the most popular techniques: the Neuroevolution of Augmenting Topologies and Population-Based Reinforcement Training algorithms.

Keywords — soft computing, mathematical models, neural network, observation, function, reinforcement learning, value function, performance, topology and weight evolving artificial neural network

# Introduction

Computational Intelligence (CI), also known as Soft Computing, regards the study and development of intelligent systems which through either compiled empirical data or experimental observation of a function, or task, attempt to learn the underlying process which it will therefore be able to replicate. This ability of “learning from examples and experience” facilitates the computerization of distinctly biological and linguistic paradigms such as “recognizing objects” as well as “understanding and responding to language” [2] creating systems which are therefore capable of “solving problems that usually require the ability of human beings” [3].

With respect to the field of CI, the premise of the study documented by this paper is the research, implementation and evaluation of two established topology and weight evolving artificial neural network algorithms which through reinforcement learning control the neuroevolution of a population of artificial neural network models with intent to optimise the populations best network. With respect to the two TWEANN algorithms chosen to be implemented, the methods chosen by the author are the ‘Neuroevolution of Augmenting Topologies’, NEAT, algorithm developed by Kenneth O Stanley in 2002 and the variation of ‘Population-Based Reinforcement Learning’ implemented by Google’s DeepMind team as part of their AlphaStar project to develop an artificially intelligent agent that could play the Blizzard real-time strategy game *StarCraft II* which could achieve Grandmaster ranking in its competitive league [15][16]. Following the paper’s initial review of academic literature relevant to the preceding study and the research, implementation and evaluation of the existing algorithms, the author will attempt to define a third TWEANN algorithm using the observed research data with the intent to create a new method which improves upon the performance of the two algorithms by considering the compiled research on each method and the advantages and disadvantages of each approach, which will then be evaluated in the same manner as the two existing algorithms implemented. Within the final component of the paper the author will provide final conclusions as to the efficacy of each of the existing TWEANN algorithms and the newly created algorithm.

# Background

## Study Definition

The purpose of the study carried out by the author and documented within this paper may be succinctly defined as the further development of reinforcement learning based topology and weight evolving artificial neural network techniques through the research, implementation and evaluation of two existing TWEANN methods: namely the NEAT and PBT algorithms, leading to the creation of a new TEANN method as defined by the author following consideration of the research that has been compiled. The purpose of this paper therefore becomes the subsequent documentation of the referenced study contains herein the compilation of the carried-out research, descriptions of the implementation of each of the chosen TWEANN algorithms and evaluations of their performance across multiple defined environments.

## Field Research

One of the main pillars of the field of Computational Intelligence is Artificial Neural Networks, a system inspired upon the biological networks which exist within the brains of animals, within which a collection of nodes, or neurons, are connected together by a series of weighted edges, or synapses to form a network. Through a machine learning algorithm, the weights of these edges are adjusted to enable the model to approximate a function or execute intelligent behaviour within an environment by feeding input data through its topology. Each of the nodes within the model is grouped into a layer which determines its purpose within the network. In the typical feed-forward network topology the connecting edges are directed and the overall topology forms a directed acyclic graph, which is to say that nodes only receive input from those in preceding layers and that there are therefore no cycles formed [4], meaning that information always travels through the network in a single direction. The first layer holds the input neurons which output the external values that are input into the network model. The final layer holds the output neurons which produce the ultimate result of the network. A network topology may also include a number of layers that exist between the input and output layers and are known as hidden layers as their exact function within the ‘black box’ system of the network is unknown. A feed-forward neural network without any hidden layers is known as a single layer perceptron while a network which contains at least one hidden layer is known as a multi-layer perceptron. Each neuron within a hidden layer or the output layer has an activation function which takes in the weighted sum of the outputs of its input neurons and a given bias and returns the value the neuron will output. If every node in either the hidden layer or the output layer receives input from every node in the layer directly preceding its own, the network is known as being fully connected. If however, the nodes are not connected in this way, the network is known as being partially connected.

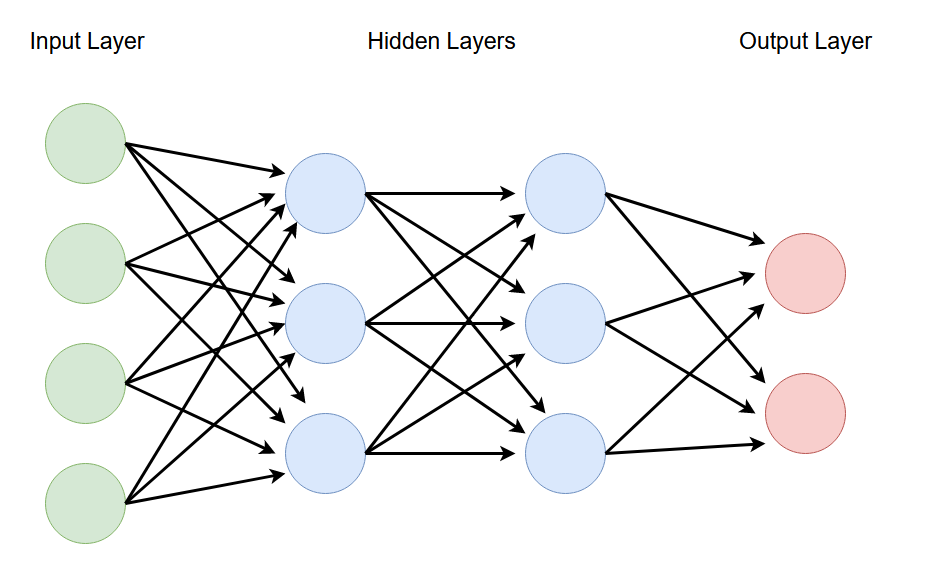


Figure 1: An example diagram showing the topology of a fully connected multi layer perceptron, with four inputs, two outputs and two hidden layers each with three neurons.

Neuroevolutionary machine learning is a subfield of CI concerning the creation of neural networks through the use of evolutionary algorithms [5] developed within the field of Evolutionary Computing, another subfield of CI concerning combinatorial optimization problems [6] with the intent of applying evolutionary strategies to the optimization of the network’s weights. This idea is further expanded upon with the creation of ‘Topology and Weight Evolving Artificial Neural Network’ or TWEANN algorithms which consider the optimization of both the networks’ weighting parameters and its model hyperparameters: parameters which control the learning process of the network relating directly to its model, namely it’s topology and size. Though not an explicit criterion of TWEANN algorithms, the TWEANN methods explored within this paper may define only feed-forward networks which may be either a single or multi-layer perceptron and either fully or partially connected.

Both the NEAT and PBT algorithms use Reinforcement Learning, to train their population of networks. One of the three most common archetypes of machine learning alongside supervised learning which maps a potential input data to the desired output state of the model when presented with the given input taken from a set of provided training data and unsupervised learning which attempts to discern any patterns that occur within the training data. Unlike these methods Reinforcement Learning does not learn through a compiled set of training data but rather through attempting to optimize its actions within an environment to maximize its performance as measured by a given value function. The use of this value function only, to output the current fitness of a network model and not a provided set of explicit training data showing the optimal output of the network, means that the method must instead make use of both exploitative search techniques, using the value function’s output performance metric to converge upon local optimum weight parameters for the model, as well as explorative search techniques, which experiment with weight parameter values not previously considered by the model in an attempt to avoid the issue of early convergence faced by pure exploitative search techniques [7]. In other words, it must both use data from previous experience to “*repeat decisions that have worked well so far*” while at the same time making “*novel decisions, hoping to gain even greater rewards*” [8].

Within the application of video games, artificial intelligence is often used to control characters in the game which aren’t controlled by the player, known as non-player characters or NPCs and other functions not visible to the player such as the procedural generation of content [9]. Through AI, NPCs can display intelligent behaviours which may adapt to the game world similarly to if they were being controlled by a human. AI powered NPC opponents were first popularized in the so-called ‘golden age of arcade games’ in which they were used to implement gradually increasing levels of difficulty to players of early arcade games. More recently, these AI opponents, often colloquially referred to as ‘bots’, have been applied within multiplayer games in the place of other human players, should they be unavailable, to ensure the game is still playable. With respect to this study, the intelligent agents defined by the implemented TWEANN algorithms act as these ‘bots’ within a series of implemented multiplayer game environments with the exception that there is no human player and the agents are competing only against themselves.

# Literature Review

The Neural Network model was first hypothesized by McCulloch and Pitts, a neurophysiologist and a mathematician respectively, in 1943 in the paper ‘A logical calculus of the ideas immanent in nervous activity’ in which they proposed that the “[binary] *character of nervous activity, neural events and the relations among them can be treated by means of prepositional logic*” may be accurately conveyed by a ‘net’ as their behaviour may be “*described in these terms*” [10] and simulated intelligent behaviour through a connected circuit. This initial binary model was later expanded upon by psychologist Donald Hebb in the 1949 book ‘The Organization of Behaviour’ who proposed the strengthening of neural pathways following successive use, as part of his proposed ‘Hebbian Learning’ model based on the concept of neural plasticity; the nervous systems ability to “*change its activity in response to intrinsic or extrinsic stimuli by reorganizing its structure, functions or connections*” [11][12]. This led to the quantification of these pathways’ respective strengths within the neural network model through the introduction of weighted edges enabling the model the ability to alter its connections as in neural plasticity.

As video games have become more popular and technology has improved, the size and complexity of modern video game environments has increased dramatically, creating a need for more complex AI programs often developed specifically for application within a single game. In their 2015 paper ‘Artificial Intelligence in video games: towards a unified framework’ Safadi, Fonteneau and Ernst cite the pressing need for ‘comprehensive’ agents capable of understanding the various aspects of the complex environments of modern video games as well as the production inefficiency of developing a bespoke AI system for each new application [13]. This demonstrates a clear necessity for new AI techniques which simplify the process of creating advanced game AI.

The genetic algorithm Neuroevolution of Augmenting Topologies algorithm was proposed by Kenneth O. Stanley and documented in his 2002 paper ‘Evolving Neural Networks for Augmenting Topologies’ which during testing he finds outperforms Grau’s Cellular Encoding and Gomez and Miikkulainen’s Enforced Subpopulations algorithms, the contemporary neuro-evolutionary techniques at that time. Stanley attributes the success of his algorithm to the use of historical markers to track genes across topologies which enables crossover between the network models’ genotypes, a speciation threshold which protects networks that are in the process of optimising a new topology and subsequently suffer in terms of performance from direct competition with models that have been optimised to protect structural innovation and the incremental complexifying the model population from a simple initial single layer perceptron [14]. Optimisation of the network models occurs through the crossover of the model’s genotypes, a genetic operator used to combine the information of two ‘parent’ solutions to create generate an ‘offspring’ solution, with genes from better performing networks more likely to appear in the resultant offspring. The functionality of the algorithm has been extended into multiple variants built upon the same principles including L-NEAT, a variant with incorporates back propagation intended for application within data classification [22].

As mentioned within the introduction section the instance of Population-Based Reinforcement Learning implemented in this study is the variant defined by the Google DeepMind team in the paper ‘Population Based Training of Neural Networks’ as part of their project to develop an intelligent agent which could reach Grandmaster level in the real-time strategy game *StarCraft II’s* competitive league, by using information gained across the whole population to optimise the models [15][16]. Models are optimised through a combination of an exploit function, which uses information about the best models in the population to optimise the model’s weights and hyperparameters, and an explore function which perturbs the model parameters in order to explore the solution space. Within the paper the DeepMind team’s exploit function use roulette-wheel based selection to choose a model to copy the weights and hyperparameters from and the explore function to create new hyperparameters by randomly perturbing its current topology however they also posit that the exact method in which these functions work should be dependent on the intended application of the resultant network model and may need to be changed in order to provide the best results.

# Methodology

Given the need for improved AI techniques to keep up with the increasingly large and complex environments of modern video game applications identified in the literature review, it’s clear that this a valid area of research and worthy of the time taken to complete this study.

Following the initial decision of the study to implement the three TWEANN algorithms and a suite of video game environments within which to trial and evaluate them, the development of the described work begun within the Visual Studio integrated-development environment [17] using the popular computer programming language Python [18]. Both the Visual Studio IDE and the Python programming language were used during the development of this project as a considerable affinity has been gained with both due to the previous experience using each during the development of other pieces of coursework during this Masters’ Degree.

Another important tool used in the development phase was the pygame library available for the Python programming language, a library is created solely for the development of video game applications created using the Python language [19]. The pygame library provided easy to use functionality for creating windows, processing input events and drawing the sprites used in each game. Without the use of this library the development of the suite of TWEANN testing environments would have undoubtedly taken much longer and the quality of the other work completed during this study would have dropped due to having less time available to work on them.

The development of the sprites used within the suite of video game environments used for testing the TWEANN algorithms was completed using an online pixel art application called Piskel [20]. Piskel supports creating both static and animated sprites as well as then exporting those sprites to a variety of different file types, allowing them to be used in other applications.

# Hybrid Solution Design

Both being reinforcement learning based TWEANN algorithms the NEAT and PBT share a considerable number of similarities, such as the fact that they both maintain a population of multiple network models which are jointly optimised using the fitness, weight and hyperparameters from across the entire population. The pivotal difference between the two comes in the way in which this information is used to populate the next generation of models; the NEAT algorithm uses crossover, one of the main principles of genetic algorithms, between the population with the chance of a gene from a parent being included in its offspring relative to its fitness meaning that genes from the better performing networks are more likely, but not certain, to appear in the next generation in place of those from the worse performing networks, while the PBT algorithm uses a combination of an exploit function to perturb each model towards one of the best achieving models and an explore function which uses randomness to ensure the solution space is explored and to attempt to prevent early convergence. Even though both these methods obviously differ in the way they use the fitness of the population’s models, it remains clear that the core issue faced by both is that same core paradigm of all reinforcement learning methods as identified within the field research. The next generation of networks must begin to converge towards the weights and hyperparameters of the highest performing models of the previous generation in order to optimise themselves however, enough diversity must be maintained within the new generation that the solution space may be more fully explored and the issue of early convergence can hopefully be avoided.

The reproductive strategy implemented by the hybrid aims to improve upon that of the defined NEAT and Population-Based Reinforcement Learning algorithms by considering both approaches and thereby enable the determining of a new approach combining the advantages of both. Therein having considerably researched both methods I would suggest that the crossover reproduction method used within the NEAT algorithm could be improved by further increasing its exploitative element to increase convergence as though this risks the previously discussed issue of early convergence upon a local optima, given the high number of network models that can be safely used within its population without causing the program to crash due to the algorithm’s efficiency, considerable variety may still be maintained across this wide population.

Whilst having also researched PBT extensively, I chose not to further extend upon it as I have found that whilst the logic of its approach to enable convergence through the exploit function whilst also still maintaining diversity through the explore function appears to be sound, in practice the random altering of the network model’s topologies without regard to their fitness in fact hampers diversity across the population as the models it defines actually decrease in fitness, causing the convergence orchestrated by the roulette-wheel selection quickly causes the entire population to converge upon the first mildly optimal structure discovered.

With these approach in mind, within the hybrid algorithm I would seek to extend upon the NEAT approach’s exploitative element in order to make full use of the expansive population of models and groups of species within its population. To that end I have decided to further increase the chance of the newly created offspring network models containing the genes of previously highly successful models by implementing the roulette-wheel based selection process, used within the PBT algorithm when choosing a network model to copy, into the NEAT algorithms decision of which two network models within the population to use as parents for a new offspring model. By implementing this selection process within the method I hope to increase the chance of more successful networks being chosen to be used as parents for the next generation and thereby lead to faster convergence within the hybrid method.

# Suite of TWEANN Testing Video Game Environments Implementations

## Testing Suite

With respect to the development of the multiple different video game environments, the different instances of each environment were implemented as specialist class variants of a base application class which contained the variables and behaviours common across each video game instance including, an instance of a pygame application window, it’s dimensions and a Boolean variable which controlled whether the game had been exited by the player. Unfortunately due to the time constraints imposed upon the project’s development cycle, the menu interface that had been planned to be implemented to allow the user to easily chose and switch between game environments was not created. In its place users must simply leave the declaration of the particular specialisation of the app class they intended to use uncommented within the main code file while the declaration of the other app classes are left commented out as can be viewed in the below figure.

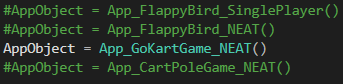


Figure 2: The declaration of specialized instances of the app class within the main code file. in order to run a chosen game environment with a TWEANN algorithm the respective declaration must be left uncommented whilst the others are commented out.

## Flappy Bird Game

The first of the implemented video-game environments used to trial the TWEANN algorithms was a recreation of the popular ‘Flappy Bird’ mobile game. The implementation of the game created for the testing suite and shown in the below figure differs from the traditional game in that it allows for multiple players to play simultaneously, each controlled by one of the network models within the population of models created by a TWEANN algorithm.

Within the app\_class.py file, which contains the definition of the app class, the fitness function for each TWEANN algorithm to call that runs the flappy bird game and returns each bird’s score is defined. As each function main purpose is to evaluate and return the fitness of each model they may be discussed as one here due to their functional similarity. First the array used to hold the network models controlling the bird players and the array used to hold each bird player are emptied. Then the list of networks input into the function is looped through and appended to the network model array, with a bird player being appended to the player array to be controlled by the newly appended network. Once the new generation of network models and their respective birds have been created the main game loop is entered, which does not exit until the game is either exited or there are no players remaining within the game. Within this loop the necessary input events are processed; each bird player is updated based on input from its respective network model and the game world is drawn onto the application screen. When the loop exits each network model is assigned a fitness value equal to the score achieved by the bird player it was controlling and the game is then reset for the next generation.

Within the initialisation function of the specialist instance of the app class used for the implementation of the flappy bird game the variables defined include: the application screen and its dimensions, the scrolling speed of the game background, the arrays holding the network models and the bird players, the background image asset loaded from memory and the y coordinate of the gap within the first set of pipes to reach the players.

During the frame update of the game the background image is scrolled and the pipes move left towards the player. If the pipes have moved past the player/s and have gone off the left side of the screen, they are then moved to the right off the screen with a new randomly set gap position. Every network model within the current population is then updated within a loop. If the bird the model is controlling is still in the game, the necessary input data for the frame is input into the model, namely the current distance between the bird and the pipes, the bird’s current velocity within the y axis, the y coordinate of the bird and the y coordinate of the middle of the gap between the pipes. If the network model’s single output value determines the bird should jump then the jump function will be called within the respective bird player. The models player will then itself be updated to move it in regard to its current velocity. If the bird is then still alive, it may die here if it collides with the top of the screen or the ground at the bottom, it will check for a collision with the pipes. If the bird is still alive following this check then it’s score will be increased by an increment of *0.1*.

The draw function for the game draws the game background, the currently alive bird players and the pipes as well as the on-screen text output.

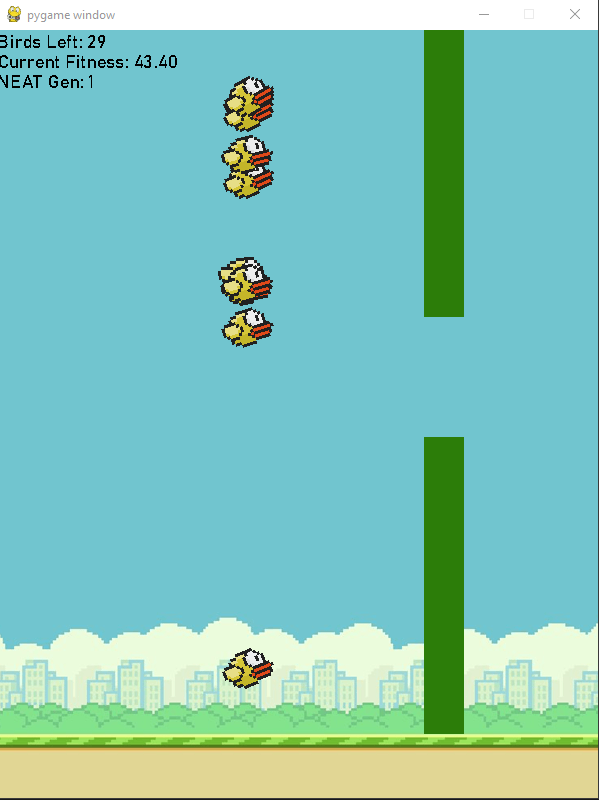


Figure 3: Application window showing the created Flappy Bird game environment. As is displayed by the textual output in the upper left corner of the window, twenty-nine birds remain within the game, each of which has a current fitness value (or score) of 43.4.

## Go-Kart Game

The second video-game environment implemented was a Go-Kart game in which the models must navigate a go-kart around a track using the information provided by a multi-directional ray cast to inform it’s steering.

As with the flappy bird game, the fitness function used by the TWEANN algorithms to evaluate the fitness of each network model is used to control the starting and resetting of the game and works in the same way. First the current array of network models and players is emptied and the repopulated using the new generation of network models provided by the algorithm. Then the main loop of the game begins which will loop until either the game is exited or there are no players remaining in the game. When the loop exits the fitness of each network is assigned from the score of its respective player and the game is reset.

Within the initialisation function of the specialist instance of the app class used for the go-kart game the application screen and its dimensions are defined and the background image is loaded from memory.

Within the frame update of the game a ray cast operation is performed from each currently alive car player to from the angles of *-90*, *-45*, *0*, *45* and *90* degrees relative to the current heading of the car which returns the distance from the car to the point of the edge of the track where each ray cast hits. Each of these five distances is then input into each network model with the first output of the model determining whether the car will turn to its left and the second whether it will turn to its right. Each kart player is then updated to move forwards in regard to its current heading and check if the kart has collided with the edge of the track, in which case it is dead or in the case that it has not collided has its fitness incremented.

The draw function for the game draws the game background, the on-screen text and each currently alive kart player and it’s five ray cast rays.

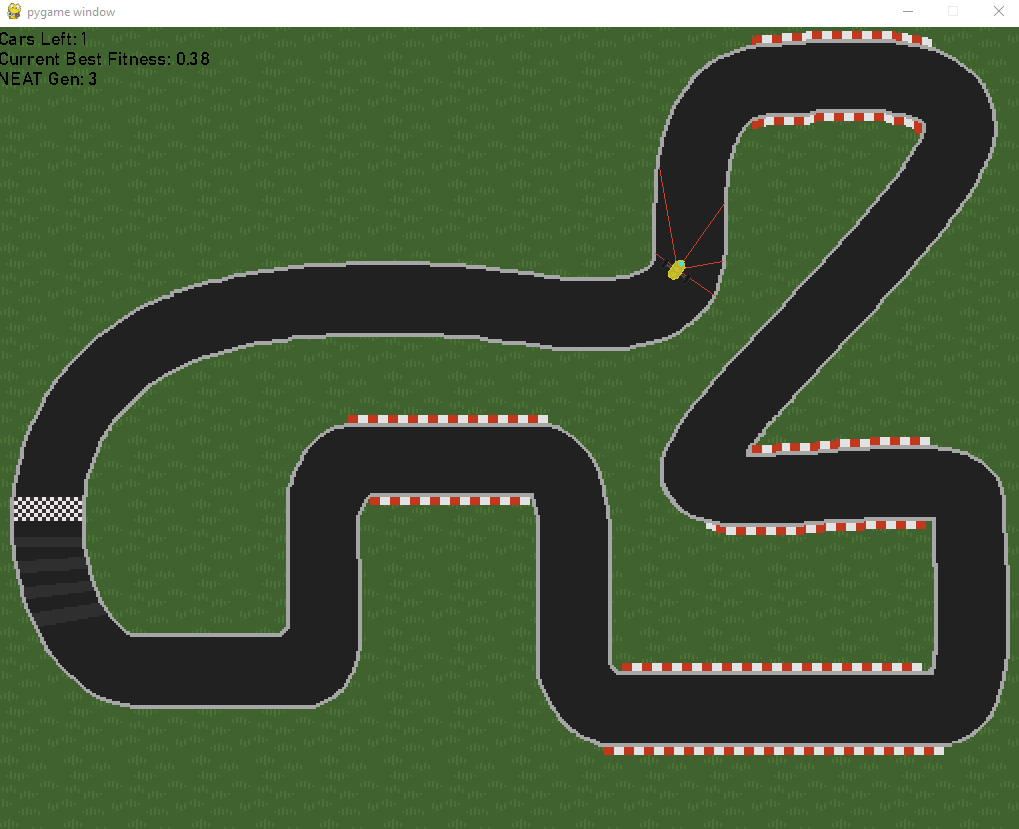


Figure 4: Application window showing the created Go-Kart game environment. Currently only a single car player remains with a fitness of 0.38.

## Cartpole Game

The final game environment implemented was a recreation of the cartpole balancing test, often used to demonstrate the ability of networks trained on reinforcement learning in which the network must control by “*moving the pivot point* [the cart] *under the centre of mass*” with the goal of “*keeping the cartpole balanced by applying appropriate force to a pivot point*” [21].

As with the other game environments, the fitness function called by the TWEANN algorithm to evaluate the fitness of its models controls the starting and restarting of the game. Further analogous to the other fitness functions it empties the current network model and player arrays, repopulates them and then enters the main loop of the game. The main loop will not exit until the game is exited or the number of alive players has reached *zero*.

Within the initialisation function of the game the player and network model arrays as well as the application screen and its dimensions are defined.

Within the game’s frame update the cart’s current x coordinate, it’s speed and the current rotation of its pole are input into the network model. As with the Go-Kart game, the network models two outputs determine whether the cart will move to the left and the right respectively. The cart is then updated with regards to this input and the balance of the pole affected by its movement. If the pole tips outside of the defined boundaries for it to be balanced within the player dies or if the pole is still within bounds has its score incremented.

The draw function for the game draws each player and the on-screen text output.

Unfortunately, due to the time constraints placed on this project by its limit duration of available work before it must be submitted, the implemented physics that were meant to control the balancing of the pole upon the cart do not reach the standard expected of the traditional Cart-Pole used in reinforcement learning. Within the implementation of the game, it is impossible to balance the pole successfully as it moves too quickly to react to. Were further development time available, this issue would be the first thing to be improved upon.

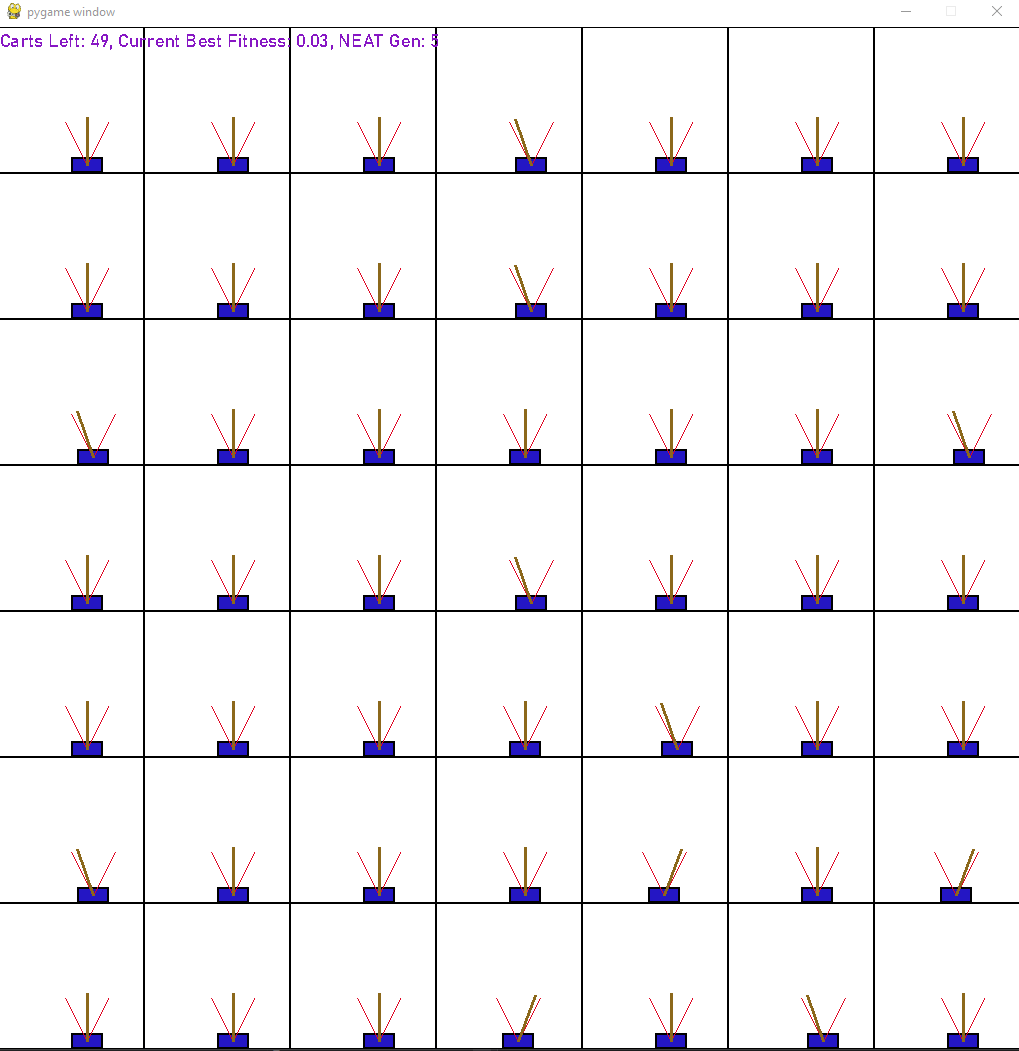


Figure 5: Application window showing the created Cart Pole game environment. Each cart may move only within the bounds of its square enclosure to balance its pole within the red outline boundaries.

# Solution Implementations

## Implementation of the Common Elements of TWEANN Algorithms

Due to the shared nature of the NEAT, PBT and hybrid algorithms as they are of course all TWEANN algorithms, elements of the required implementation for each can be considered as largely, if not exactly, the same for each algorithm. Therefore the implementation of these methods is divided by the single element of the respective TWEANN algorithms within which they differ; their method of reproducing a new generation of network models using fitness of the previous model generation to inform the process. Aside from this dividing difference, the only other identifiable area in which these methods may differ further is the necessary encoding of the models within the NEAT and hybrid algorithm, which are built upon the principles of genetic algorithms inspired by the process of natural selection, in the form of a genome structure which enables the models to be crossed over during reproduction. While this encoding is not inherently crucial to the PBT algorithm, it has also been decided to be encoded in this way to provide the functionality of mutation available to genetic algorithms so that the PBT network models topology may be as easily altered as those of the NEAT and hybrid algorithm.

The encoding of the defined network models as genomes is enabled through the implementation of two classes; the gene classes in the genes.py file to represent each individual gene within the larger genome, and genome.py which holds a collection of gene objects.

Within the genes.py file three different gene classes are defined; a specialist default node gene class, a specialist default connection gene class and a base gene class which the node and connection gene class inherit their shared attributes and behaviours. From the base gene class both gene types inherit the required behaviours to both mutate themselves and crossover with another given gene. The specialist node gene class defines four attributes unique to node genes namely, it’s input bias, its current output value, the activation function it uses to calculate its output and the aggregation function it uses to calculate it’s input value from the collective output values of its input nodes. The specialist connection class on the other hand defines only two; it’s weight and a Boolean that defines whether it’s being used in the current model.

In the implemented node gene class, all nodes have been set by default to use the sigmoid activation function and the sum aggregation function however for the benefit of variability during testing a series of activation and aggregation functions were implemented within the utility files activations.py and aggreagations.py respectively. Within activations.py the sigmoid, tanh, sin, gauss, relu, elu, lelu, selu, softplus, identity, inverse, log, exponential, absolute, hat, square and cube activation functions are define while in aggregations.py the product, sum, max, min, absolute maximum, median and mean aggregation functions are implemented. Implementing the functions in a separate file to the one they’re used in means that only the activation and aggregation function used needs to be imported which in turn decreases the memory overhead of the suite of game environments application.

In order to simplify the process of handling the multiple parameters necessary for a TWEANN algorithm to be executed, a further utility file, config.py, was implemented to contain all of the necessary data. Based upon the same implementation as the genome encoded representation of the network models the file defines four classes; a config parameter class to hold a single parameter of the overall configuration, an unknown config item class to be instantiated following any formatting errors in the input config file, a config base class to contain only the parameters that must be configured by the user for an instance of a TWEANN algorithm to run and a specialist config class used that contains all of the parameters that may be configured by the user. During an instance of one of the algorithms, a config object of either the base or specialist class is defined and passed to it, so that it may refer to it whenever it needs to know the value of a parameter.

In order to further simplify the process of considering the many parameters included within an instance of a TWEANN algorithm, the user inputting values for the config object was implemented through way of a config text file which would then be read in from memory when the algorithm is first executed. The implementation in this way means that users not familiar with programming could still make use of the program and begin to understand how the method works by considering its necessary input data without having to experiment with its code. The input process was then simplified even further with the implementation of a file writer tool which could write provided parameter values to the config file so that when the user switches between game environments which uses network models with a different number of input and output values or used a different population size for its generations, these parameters would be set automatically when the new environment is executed.

The population class defined within the population.py file holds all of the current generation of network models and implements the core of the evolutionary algorithm which the user calls to begin the current instance of one of the TWEANN algorithms. When the ‘run’ function is called, the user provides the fitness function which will evaluate and return the fitness of each of the current network models and the number of generations for the algorithm to run. The function defines a while loop which runs until the number of generations exceeds the provided limit or a fitness termination criterion set within the config file has been reached. Each loop a reporter object outputs the start of a new generation, further detail on this class later, and the fitness function is called. Once a fitness has been assigned to each of the network models, the initial generation of networks are all initialised as single layer perceptrons with randomised weight values, the network model with the highest fitness value is checked against the network model with the highest discovered fitness throughout the current instance of the TWEANN algorithm and replaces it if its own fitness exceeds that of the current best seen model. The next generation of models is then created using the current generation of models and their fitness values. If every species within the current generation has stagnated, then the reporter object outputs the extinction of the population and asks the user if they would like to restart the current instance of the algorithm with another set of randomly defined network models or otherwise to raise the ‘complete extinction’ exemption and close the program. This is because if none of the species of models within the population have shown area for improvement, then the algorithm has no way of further optimising them. If there are still non-stagnant species within the population, then the new generation is divided into species and the reporter object outputs the end of the current generation. Although the end of the looped creation of further generations of network models is never reached due to the time to evaluate each generation of network models taking a fair amount of time and the number of generations the algorithm is set to run for in this study always being set to a thousand, which would take a considerably long amount of time to reach, as the implementation of the algorithms within the testing environment is meant to for evaluation only and not to actually use any of the generated networks, when the end of the loop is reached the reporter object will output the fitness of and return the best found solution.

The function used to check a species of model within a generation for having stagnated is defined within the default stagnation class defined within the staganation.py file, with ‘default’ again referring to the possible inclination of the user to define other methods of checking for stagnation within a species. The default method implemented will define a species of model as having stagnated if it has not improved for a number of generations specified within the config object.

The models within the population are divided into species as defined by the speciate function within the default species set class in the species.py file based on the similarity of their hyperparameters. The models are divided into groups in this way to protect structural innovation among the population by allowing models to compete only with other models of the same species meaning that models with new structures that have yet to be optimised will not have to compete with models that have had their structure optimised and is one of the main principles of NEAT as laid out by Stanley [14]. The function which determines the similarity of the hyperparameters between two network models outputs this similarity as a value between *zero*, indicating no similarity although this is theoretically impossible, and *one* indicating that the network’s hyperparameters are the same. The threshold value which this similarity must at a minimum reach in order for two networks to be considered of the same species is defined within the config object and for the basis of this study is set at *0.6*. First the best representative of each of the current species is chosen by finding the network model within that species closest to the species current representative. Every other model not chosen as a representative is unspeciated and added to a list which is then iterated through. Each model within the list is then compared with the representative of each of the current species. If the model has a calculated similarity value which meets or exceeds the defined threshold value with the representative with which it is most similar, it then becomes a member of the species for which that model is representative. If however it does not meet this threshold value with any of the representative models a new species is then defined for which the model becomes the best representative. This idea of speciating the model population is not considered within the implementation of the PBT algorithm however, it remains included within the files imported to enable it’s use in the event that this is changed in the future.

To assist in the process of evaluating the performance of the algorithm, a reporter class is defined within the reporting.py file and used to output the current status of the TWEANN algorithm instance as well as a number of metrics regarding the fitness of the previously evaluated generation of models to the console window during the execution of the testing suite, an example of which can be seen in the below figure. It is through this output that the performance of the methods was evaluated.

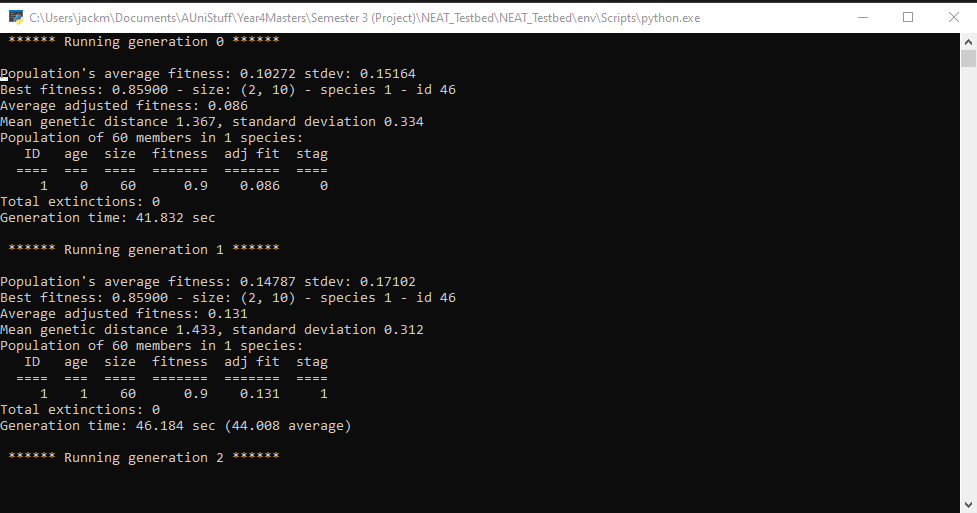


Figure 6: A screenshot showing the output of the reporter object into the console window during an instance of NEAT. Statistics shown include the average fitness of the population, the fitness of the best model and the total time taken to compete the generation.

## Implementation of the Neuroevolution of Augmenting Topolgies (NEAT) Algorithm

The method of reproducing the next generation of network models from the previous generation within the NEAT TWEANN algorithm is defined within the default reproduction class within the reproduction.py file. The term ‘default’ reproduction indicating that similar to the multiple activation and aggregation functions defined, another method of reproduction could be defined here by the user in future however the current instance of reproductions remains the default. First any species that have stagnated are filtered out from the current generation, so as not to impact the production of the next generation. If following this process there are no species left then an empty generation of models and species set is returned which triggers the complete extinction handling events documented earlier. Next the fitness values of the models are adjusted to lie within the range *zero* to *one*. Then the number of new members to be generated for each species is determined and begins a loop within which to generate these new models. For each new model to be generated from each new species, two models are chosen at random from that species and then crossed over to create a new child model which is then mutated and added to the new generation. Once this loop is complete the newly created generation of models is returned, so that they themselves may be evaluated and begin the process again.

## Implememtation of the Population-Based Reinforcement Learning (PBT) Algorithm

The variant of the population-based reinforcement learning algorithm taken from the DeepMind paper in which the algorithm was defined [15], uses a combination of an exploit function which uses roulette-wheel based selection to choose a model from the previous generation to copy the weights and hyperparameters of and an explore function which randomly perturbs the models hyperparameters to ensure the solution space is sufficiently explored to create the new generation of models. Further analogous to the described implementation of the algorithm the exploitative element of the reproduction process is achieved via roulette-wheel selection of a model within the current generation, with each model’s respective fitness being relative to their chance of being chosen, to choose an existing network model to copy. The explorative element of the method is reproduced through the mutation of the genome object describing the associated network model. While the genetic mutation of a network model is not explicitly described within the original paper as a method of perturbing the network’s hyperparameters, such a structural change of the structure of the network ensures that the solution space is readily explored and does so by altering the networks topology as is discussed within the paper [15].

## Implementation of the Hybrid Algorithm

As described within the section of this report pertaining to the design of the hybrid function, the aim of its hybrid reproduction method is to combine ideas from both the NEAT and PBT algorithms in order to create another TWEANN algorithm which maintains the advantages of both. In pursuit of this goal, the structure of the hybrid algorithm was determined to be that of the NEAT algorithm with the addition of the roulette-wheel style selection used within the PBT algorithm to inform the hybrid algorithms choice of models to use from previous generations as parents to create the offspring network models in the new generation. As the roulette-wheel selection was previously implemented within the PBT method, implementing the same logic into the hybrid algorithm was relatively simple and the only task remaining became the integration of the selection model with the NEAT algorithm’s choice of parent models which when completed finished the implementation of the algorithm.

# Implemented Solution Testing

## Testing Process

Following the successful implementation of the NEAT, PBT and newly defined hybrid algorithm within the application, the suite of video-game testing environments also implemented were utilised in order to trial the ability of each TWEANN method. Each method was trialled within each of the implemented game environments for an allocation of ten model generations to examine how the best model within the population, the overall average fitness of the population and the standard deviation of the population’s fitness was affected over time. The output results have been compiled into the below series of graphs and are further referred to during the individual evaluation of each method within the following sections of the paper. Full sized versions of the included graphs and the results table from which they were drawn can be found in appendix B.

## Testing conducted within the Flappy Bird environment

The below graph shows the best recorded fitness value, the average fitness value and the standard deviation of the fitness values across the population of network models for each TWEANN algorithm across ten generations running within the Flappy Bird Environment. NEAT can be seen in blue, Population-Based Reinforcement Learning in orange and the hybrid algorithm in grey.

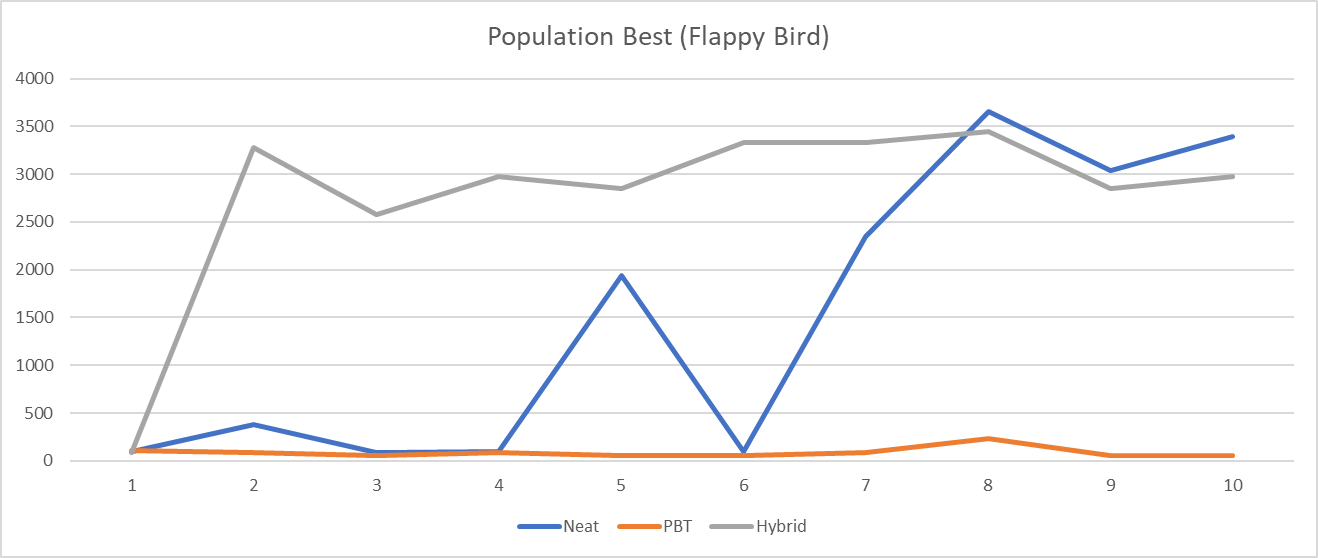


Figure 7: The population best fitness for each TWEANN algorithm across ten generations in Flappy Bird. A full-sized version and the associated table can be found in appendix B.

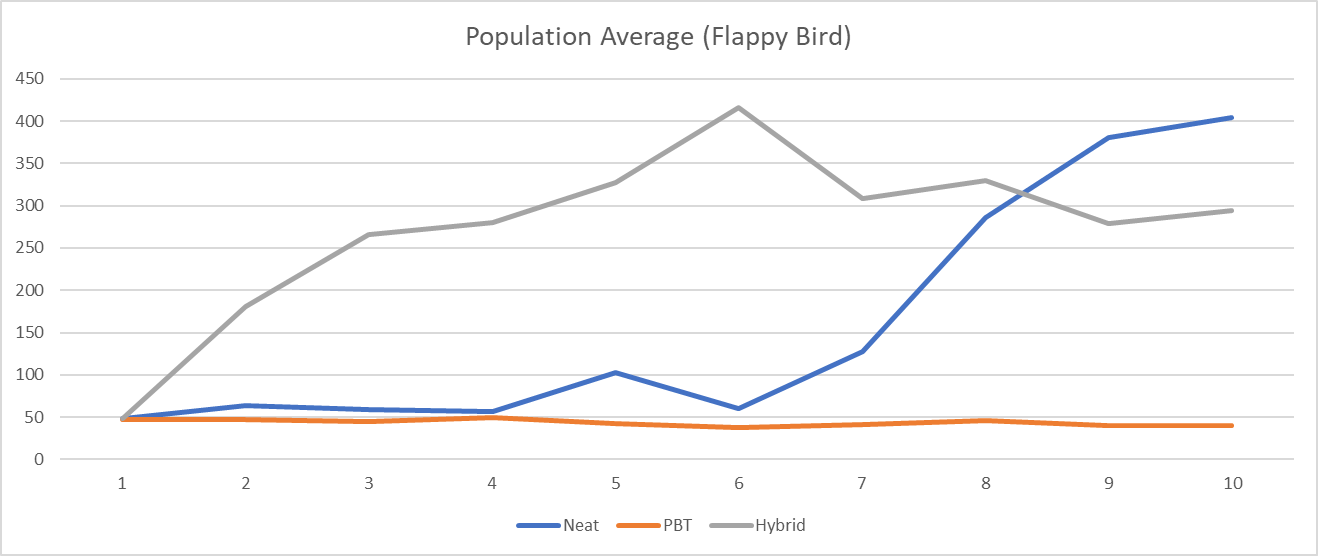


Figure 8: The population average fitness for each TWEANN algorithm across ten generations in Flappy Bird. A full-sized version and the associated table can be found in appendix B.

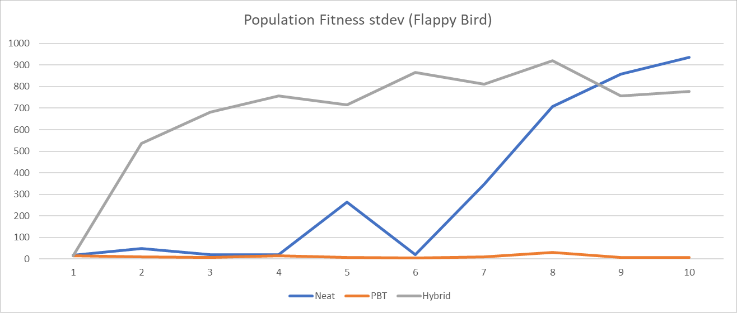


Figure 9: The standard deviation of the fitness across the population for each TWEANN algorithm across ten generations in Flappy Bird. A full-sized version and the associated table can be found in appendix B.

## Testing conducted within the Go-Kart game environment

The below graphs show the best recorded, average and standard deviation of the population of network models created by the implemented TWEANN algorithms across ten generations running within the Go-Kart game environment. As with the previous set of graphs NEAT can be seen in blue, PBT in orange and the hybrid algorithm in grey.

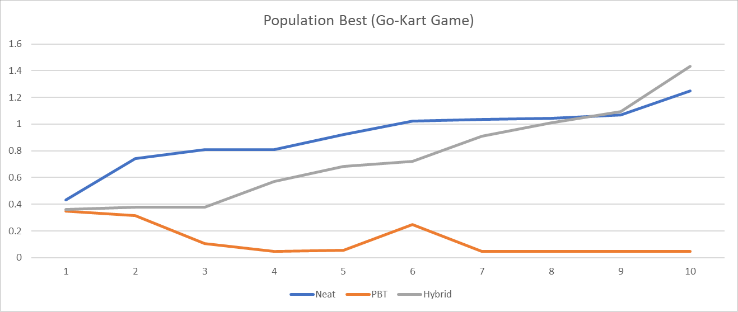


Figure 10: The population best recorded fitness value across the population for each TWEANN algorithm across ten generations within the GoKart game. A full-sized version and the associated table can be found in appendix B.

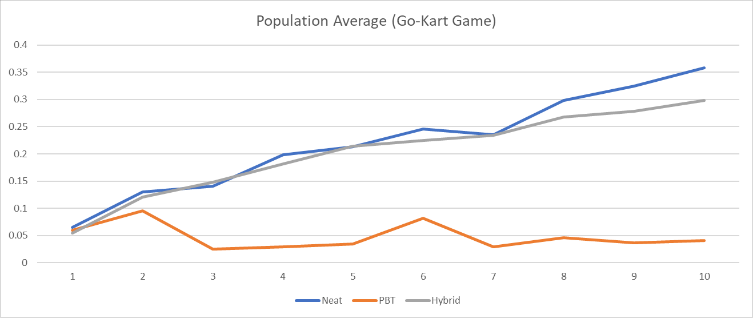


Figure 11: The population average recorded fitness value across the population for each TWEANN algorithm across ten generations within the Go-Kart game. A full-sized version and the associated table can be found in appendix B.

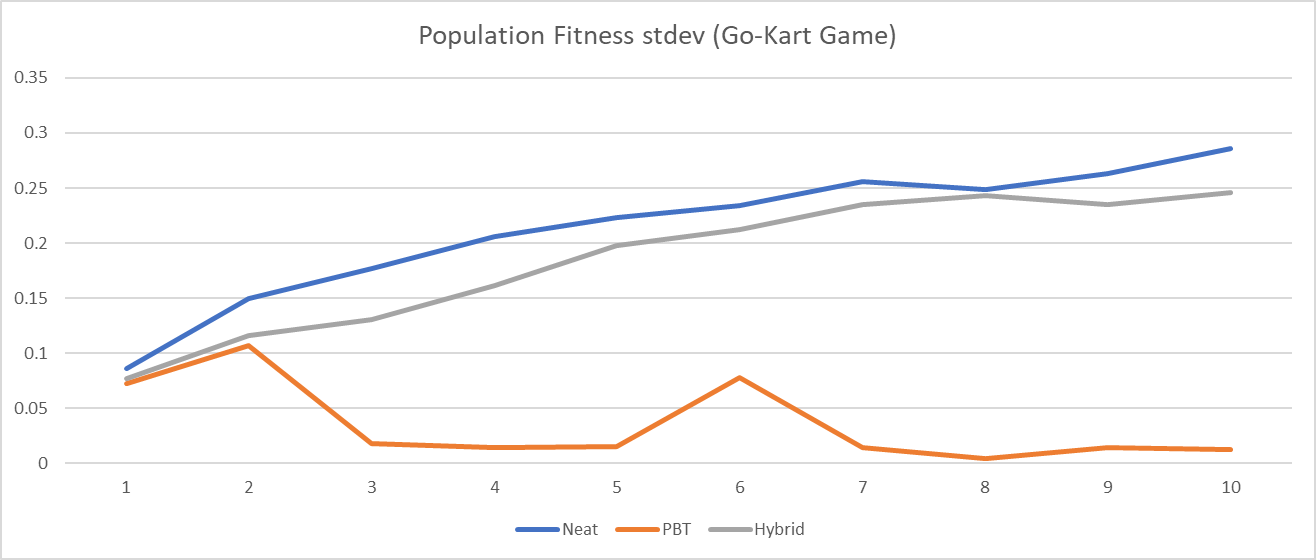


Figure 12: The standard deviation of the recorded fitness values across the population for each TWEANN algorithm across ten generations within the Go-Kart game. A full-sized version and the associated table can be found in appendix B.

## Testing conducted within the Cart-Pole game environment

The graphs below show the best recorded, average and standard deviation from the fitness values of the population of network models created by the TWEANN algorithms across ten generations running within the Cart-Pole game environment. NEAT can be seen in blue, PBT in orange and the hybrid algorithm in grey. Within figure 13 however, only the hybrid algorithm’s line can be seen as the best recorded fitness value for each generation of every TWEANN algorithm was the same value and as such the final line on the graph to be drawn obscures the previous two.

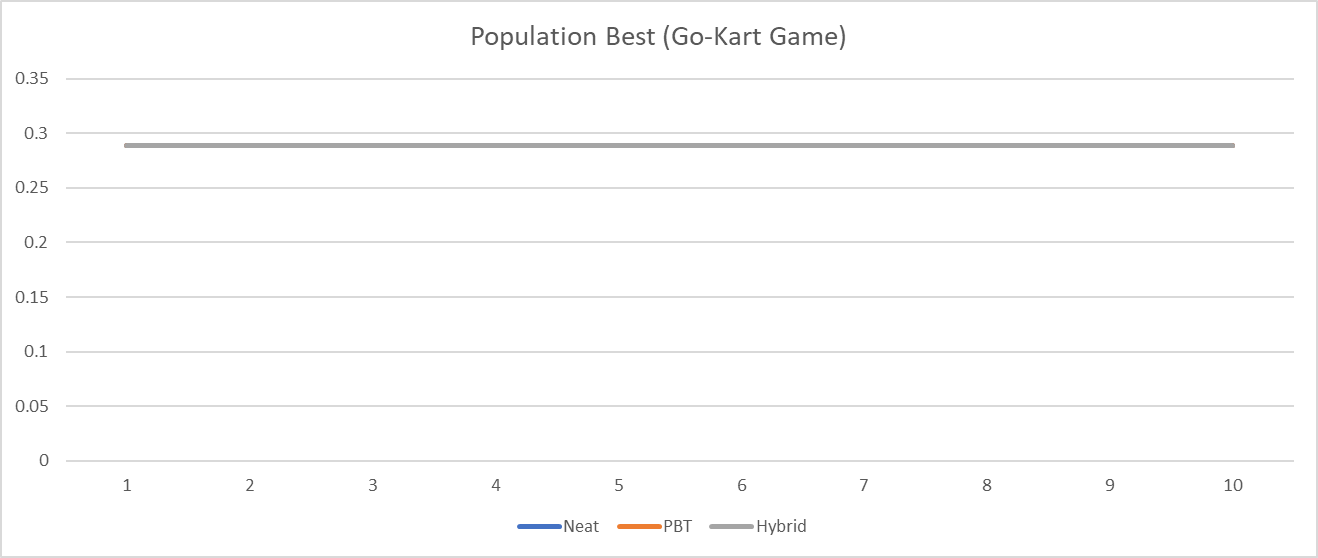


Figure 13: The best recorded fitness of the population of models created for each TWEANN algorithm across ten generations within the Cart-Pole game. Full sized version and the associated table located in appendix B.

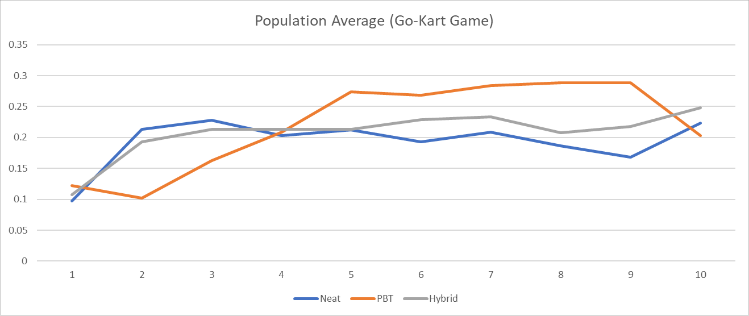


Figure 14: The average recorded fitness across the population of neural networks created by each TWEANN algorithm across ten generations within the Cart-Pole game. Full size version and the associated table can be located in appendix B.

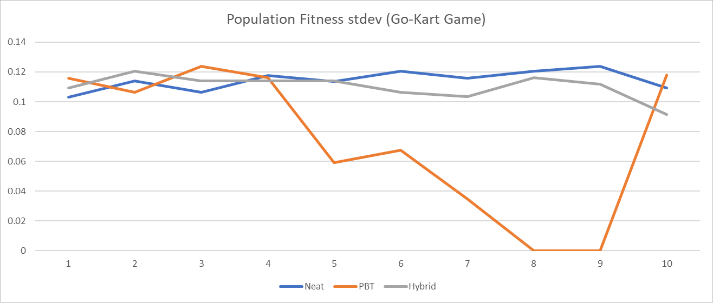


Figure 15: The standard deviation of the fitness values of the population of neural networks created by each TWEANN algorithm across ten generations within the Cart-Pole game. Full size version and the associated table can be located in appendix B.

# Implemented Solution Evaluation

## Evaluation of the Neuroevolution of Augmenting Topolgies (NEAT) Algorithm

Following the compilation of the results output from the testing process and the visualisation of these results in the above graphs it is apparent that the NEAT algorithm along with the hybrid algorithm, were the most successful TWEANN methods with the NEAT algorithm producing the network model with the highest fitness value in the Flappy Bird game at the end of the ten generations. It can be viewed in both figures 7 and 8, the dramatic increase in the best fitness value and the average fitness value of the entire population upon its seventh generation, which both continue to increase further showing that the algorithm successfully converged the following generation of models upon the new best network and were able to further optimise it. This convergence did not occur correctly the first time the algorithm generated a model with a high fitness value unfortunately. During its fifth generation a network model with a fitness of *1943.7* was created by the algorithm however, within the following sixth generation the algorithm did not converge upon this network and the population’s best network model fitness dropped back down to just *94.2*. The fact that the algorithm did not converge upon this clearly optimal network can likely be attributed to the fact that its high performance was a ‘fluke’ within a remaining population of largely unfit networks, we can interpret this from the fact that even with this network model the average fitness value of that fifth neat generation was just *103.216*, and the highly optimal network was not chosen to be a parent to an offspring network when the next generation was produced. The implemented hybrid algorithm meanwhile had the exact opposite experience on the other hand and was able to raise its highest fitness value from just *85.6* to a much larger *3279*, one of the highest fitness values recorded, by its just second generation. Considering what happened to the NEAT algorithm and the implementation of the roulette-wheel style selection within the hybrid algorithm this is very likely because the hybrid algorithm is less likely to use a network model with a low fitness as a parent for offspring because it uses the fitness of a model as the chance of it being chosen for crossover while, the NEAT algorithm choses from suitable parents entirely at random. Despite this setback in the algorithms generation of the models it is still very much worth noticing that even with this mistake it still managed to catch up to the hybrid algorithm very quickly and even finally exceed it with the highest performing network.

## Evaluation of the Population-Based Reinforcement Learning (PBT) Algorithm

The variation of the Population-Based Reinforcement Learning implemented in this study was by far the least successful of the implemented TWEANN algorithms as not only did it fail to achieve a single network with an even noticeable fitness but it also did not make good use of it’s knowledge of the fitness of it’s own previously generated models and failed to converge upon any of its higher performing networks, such as on its eighth generation of the Flappy Bird game when it’s best fitness jumped from *82.2* in the previous generation to *236.4* and PBT in fact made its ninth generation a series of worse models than in the seventh with a best fitness in the ninth generation of *50.5*, and even in the case of the Go-Kart game ended on a best model with a lower fitness than that of the best it had had in the first generation of models that were created completely randomly.

The fault for this notably poor performance can be attributed to the choice to implement this particular variant of PBT. Population-Based Reinforcement Learning as it is define by DeepMind within the 2017 paper ‘Population Based Training of Neural Networks’ [15], from which the implementation used in the study is derived, is simply as a reinforcement learning based TWEANN algorithm which creates new generations of network models by simply using an exploitative then an explorative optimisation function upon its current model generation before then announcing that the exact method in which these functions perform their respective optimisations of the models should be determined by those implementing the method and their specific behaviour should vary from application to application and provide only a suggestion as to one such method of implementing them which by their own previous declaration of the methods need to be tailored bespoke for each new implementation should be largely useless.

## Evaluation of the Hybrid Algorithm

As mentioned during the evaluation of the NEAT algorithm, it and the implemented hybrid algorithm were the most successful of the implemented TWEANN methods producing the similarly highly performing networks in both the Flappy Bird and Go-Kart games. Upon analysing the results compiled from the study it is clear that the implementation of the roulette-wheel selection mechanism within the reproduction method proved fruitful as the algorithm was able to immediately converge upon the higher performing network model and engage in further optimisation, increasing the final best fitness even more.

# Final Conclusion

## Critical Review of the conducted study

In considering the study documented in this paper, the element of the project believed by the author to be deserving of the most praise is the development of the suite of game testing environments used to analyse the performance of each TWEANN algorithm. This is because not only did these game environments prove difficult to successfully implement and integrate with each tested method but the fact that before they were even able to be implemented, a considerable amount of time was taken into the research and design process of game environments which would allow considerable numbers of players to interact with it at once and for their performance within the environment to be able to be measured accurately using a value function, which is of course the main caveat for a reinforcement learning method to be able to consider and improve within a provided environment.

Further praise it is believed should be given to the professional academic style of both formatting and writing adopted during the documentation of the study within this paper as it significantly improves the overall impression given to an individual reading this and provides them with confidence in this document as a notable scientific work.

The main issue with the implemented suite of testing environments is of course the fact that the implemented Cart-Pole game does not work as was intended and as such failed to provide results about the effectiveness of the TWEANN algorithms that could have benefitted the study. As mentioned within the implementation section, if there were more time available to work on the project this would be the area for development at the top of the list, to ensure that enough valid results are provided to enable the informed evaluation of the implemented TWEANN methods.

With respect to the professional quality of the completed study with the benefit of hindsight now that the project is finished, further effort could have been directed into the research and implementation of more varied and complex existing TWEANN algorithms in order to extend the comprehensiveness of the documented study. Having further knowledge of TWEANN methods, especially those which vary considerably from the workings of those already implemented, would provide an extensive understanding of new ways in which the creation of new network models can be informed by the fitness of the generation previous and further optimise the network models.

## Final Concluding Remarks

Despite the difficulties compile within the previous section, as the sole researcher involved in this study I believe that a commendable quality of work has been achieved and am happy to suggest that this paper be considered by other researchers working within similar fields as a potential referable piece of work.

# Acknowledgments

In consideration of this paper being completed as the thesis section of a Masters’ course, acknowledgements should be extended to the members of staff at De Montfort University whose involvement was intrinsic to this project. Firstly my largest and kindest acknowledgements must go to my academic supervisor for this project, Dr Vasileios Germanos, a Computer Science lecturer at De Montfort who’s input throughout the course of this project was invaluable and helped improve the quality of the completed work. Furthermore, I would also like to extended my gratitude to Professor Eerke Boiten, a professor in Cyber Security at De Montfort, who agreed to extend the time period for this project’s completion due to external circumstances related to the COVID-19 pandemic which unfortunately impacted on the amount of time I was able to dedicate to its completion.

With respect to another member of De Montfort University who is not a member of staff, I would also like to extend my gratitude to my course mate, close friend and roommate for the last two years of my course, Adam Hubble, who has provided me with immeasurable encouragement and motivation throughout the more trying periods of the coronavirus pandemic. Without his support I know that the quality of both this project and all other works completed as part of this course would have greatly suffered in terms of quality.

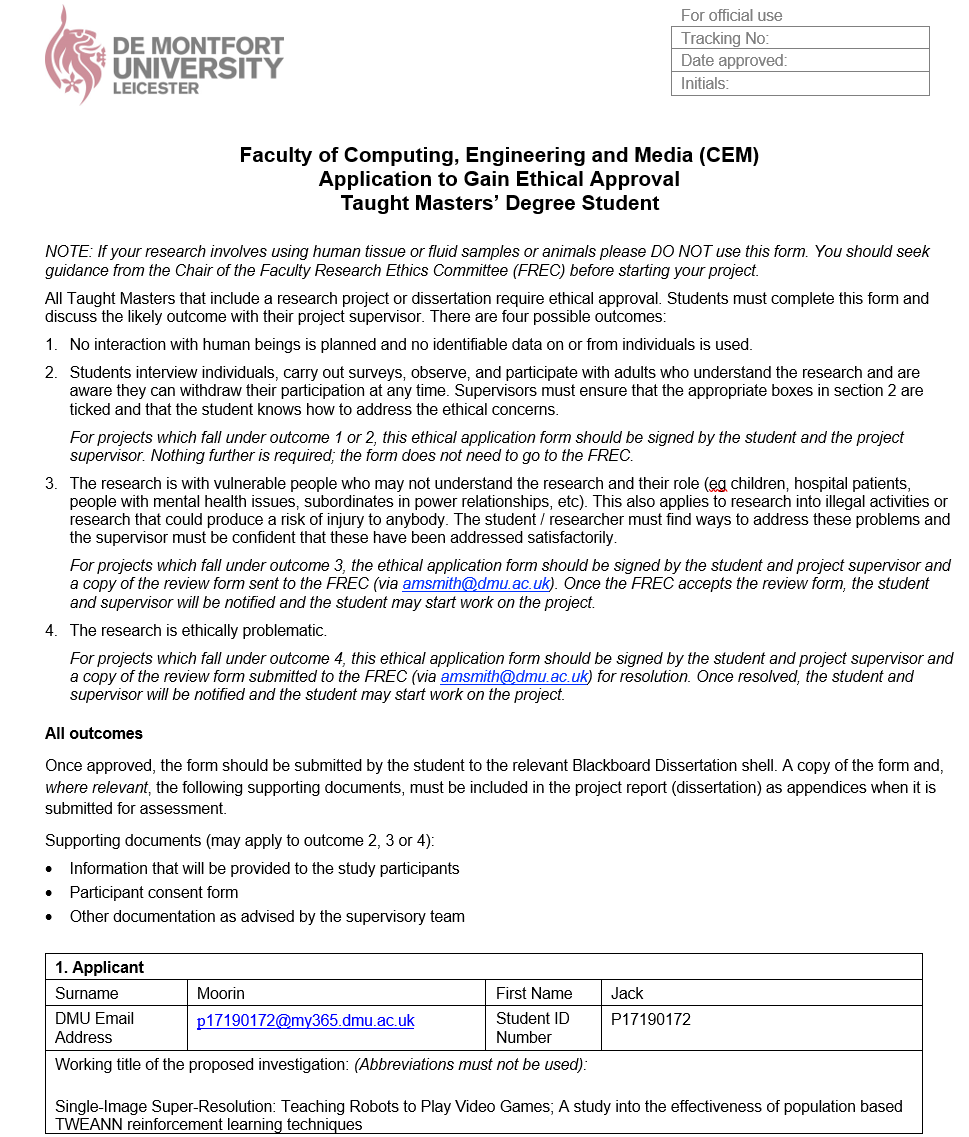
A professional acknowledgement should also be extended to the creators of the Neuroevolution of Augmenting Topologies and Population-Based Reinforcement Learning algorithms, Kenneth O Stanley and Google’s DeepMind team respectively, for making considerable breakthroughs in the field of Computational Intelligence with their reinforcement learning methods, extensively documenting each and for being the primary inspiration for this study.

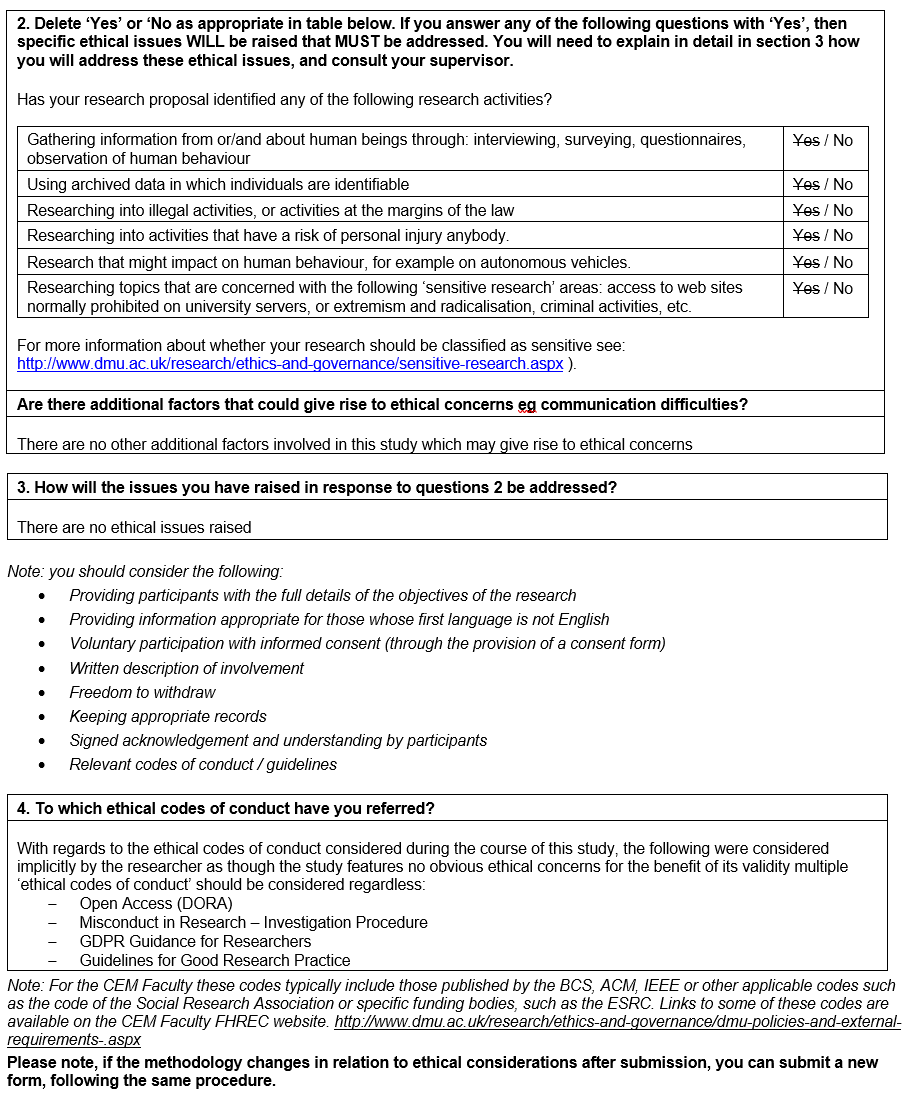
# References

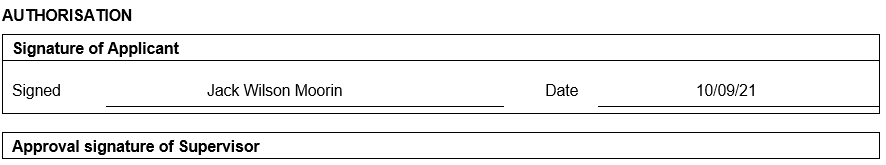
1. Hu J, Niu H, Carrasco J, Lennox B, Arvin F 2020, ‘Voroni-Based Multi-Robot Autonomous Exploration in Unknown Environments via Deep Reinforcement Learning’ IEEE Transactions on Vehicular Technology, vol. 69, no. 12, pp. 14413 – 14423.
2. IBM Cloud Education (2020) What is Artificial Intelligence (AI)? [Online] IBM. Available from <https://www.ibm.com/uk-en/cloud/learn/what-is-artificial-intelligence> [Acessed: 03/09/2021]
3. Hatton J-P (2006) A brief introduction to artificial intelligence. [Online] IFAC. Available from <https://www.sciencedirect.com/science/article/pii/S1474667015330226> [Accessed: 03/09/2021]
4. Zell, A 1994, *Simulation Neuronaler Netze*, 1st edn, pg 73, Addison Wesley
5. Stanley K O (2019) Neuroevolution: A Different Kind of Machine Learning. [Online] O’Reilly Media. Available from <https://www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/> [Accessed: 29/08/2021]
6. Dao T T 2011, ‘Investigation on Evolutionary Computation Techniques of a Nonlinear System’, *Modelling and Simulation in Engineering*, vol. 2011, no. NA, pp. NA
7. Kaelbling L P, Littman M L, Moore A W 1996, ‘Reinforcement Learning: A Survey’, *Journal of Artificial Intelligence Research*, vol. 4, NA, pp. 237 – 285
8. towards data science (2019) Intuition: Exploration vs Exploiation. [Online] towards data science. Available from <https://libraryguides.vu.edu.au/harvard/internet-websites> [Accessed: 10/09/2021]
9. Yannakakis G N 2012, ‘Game AI revisited’, *Proceedings of the 9th Conference on Computing Fronteirs*, NA, NA, pp. 285 - 292
10. McCulloch W S, Pitts W 1943, ‘A logical calculus of the ideas immanent in nervous activity’, *The bulletin of mathematical biophysics*, vol. 5, NA, pp. 115 – 133
11. Hebb D O 1949, *The Organization of Behaviour*, Psychology Press, NA
12. fronteirs in (2019) The Impact of Studying Brain Plasticity. [Online] fronteirs in. Available from <https://www.frontiersin.org/articles/10.3389/fncel.2019.00066/full> [Accessed: 10/09/2021]
13. Safadi F, Fonteneau R, Ernst D 2015, ‘Artificial intellignece in video games: towards a unified framework’, *International Jounral of Computer Games Technology*, vol. 2015, is. January 2015, Article no. 5, pp. 5
14. Stanley K O, Miikkulainen R 2002, ‘Evolving Neural Networks through Augmenting Topologies’’, *The MIT Press Journals*
15. Jaderberg M, Dalibard V, Osindero S, Czarnecki W M, Donahue J, Razavi A, Vinyals O, Green T, Dunning I, Simonyan K, Fernado C, Kavukcuoglu K 2017, ‘Population Based Training of Neural Networks’
16. Vinyals O, Babuschkin I, Czarnecki W M, Matthieu M, Dudzik A, Chung J, Choi D H, Powell R, Ewalds T, Georgiev P, Oh J, Horgan D, Kroiss M, Danihelka I, Huang A, Sifre L, Cai T, Agapiou J P, Jaderberg M, Vezhnevets A S, Leblond R, Pohlen T, Dalibard V, Budden D, Sulsky Y, Molloy J, Paine T L, Gulchere C, Wang Z, Pfaff T, Wu Y, Ring R, Yogatama D, Wünsch D, McKinney K, Smith O, Schaul T, Lillicrap T, Kavukcuoglu K, Hassabis D, Apps C, Silver D 2019, ‘Grandmaster level in StarCraft II using multi-agent reinforcement learning’, *nature*, vol 575, NA, NA
17. Visual Studio (2021), Visual Studio IDE. [Online] Microsoft. Available from: <https://visualstudio.microsoft.com/> [Accessed 12/09/2021]
18. Python (2001), Welcome to Python. [Online] Python Software Foundation. Available from: <https://www.python.org/> [Acessed 12/09/2021]
19. Pygame (2021), *News*. [Online] pygame. Available from: <https://www.pygame.org/news> [Accessed 12/09/2021]
20. Piskel (2021), *Piskel App*. [Online] piskelapp. Available from: <https://www.piskelapp.com/> [Accessed 12/09/21]
21. Greg Surma (2018), *Cartpole – Introduction to Reinforcement Learning (DQN – Deep Q-Learning)*. [Online] medium. Available from: <https://gsurma.medium.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288> [Accessed 16/09/2021]
22. Chen L, Alahakoon D 2006, ‘NeuroEvolution of Augmenting Topologies with Learning for Data Classification’, *IEEE*, 2006 International Conference on Information and Automation, NA, NA

# Appendices

Appendix A:









Appendix A: The Ethical Approval Application form completed to gain approval for this paper and the project it documents to be able to be considered as an acceptable thesis for the author’s Masters’ Degree

Appendix B:

Population Best When Tested in Flappy Bird

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 99.6 | 383.3 | 87.7 | 97.1 | 1943.7 | 94.2 | 2350.4 | 3657.8 | 3040.2 | 3391.9 |
| PBT | 104.9 | 82.2 | 53.8 | 82.2 | 50.5 | 50.4 | 82.2 | 236.4 | 50.5 | 50.4 |
| Hybrid | 85.6 | 3279 | 2573.5 | 2977.6 | 2848.6 | 3335.9 | 3335.9 | 3446.9 | 2848.6 | 2977.6 |

Table 1: The population best fitness as recorded across ten generations of each TWEANN algorithm being implemented in the Flappy Bird game.

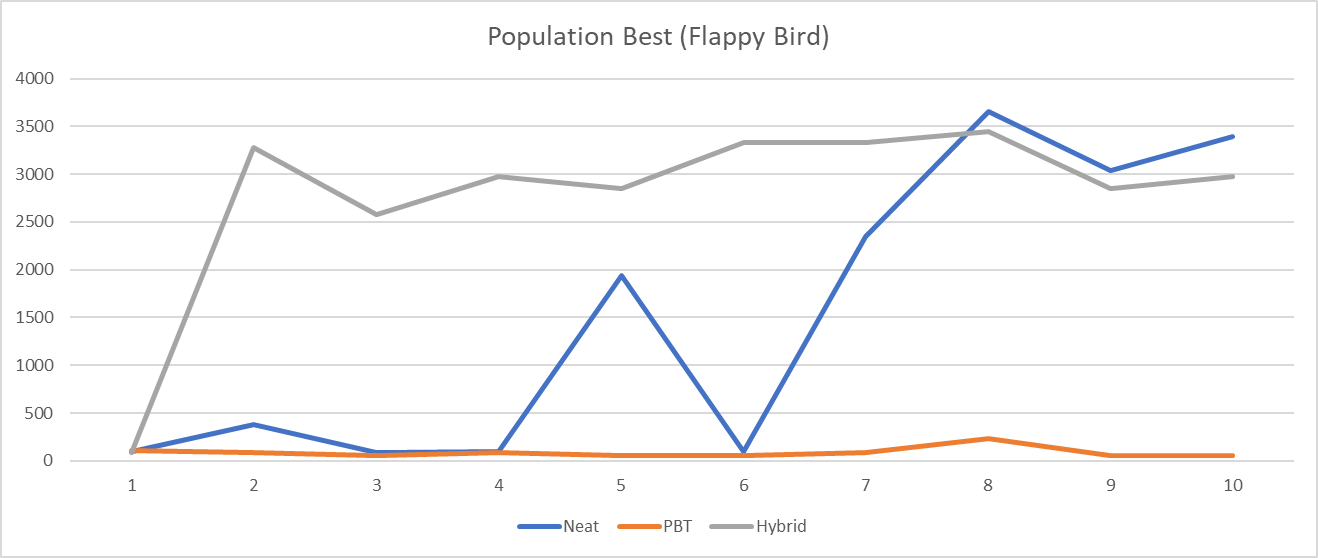


Figure 16: Full size version of figure 7.

Population Average When Tested in Flappy Bird

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 47.832 | 63.678 | 59.38 | 57.192 | 103.216 | 60.622 | 128.088 | 285.794 | 380.444 | 403.74 |
| PBT | 46.974 | 46.63 | 45.316 | 49.598 | 42.236 | 37.552 | 40.816 | 46.308 | 40.5 | 39.888 |
| Hybrid | 48.69 | 180.35 | 266.152 | 280.288 | 327.284 | 415.582 | 308.522 | 329.524 | 278.624 | 294.234 |

Table 2: The population average fitness value as recorded across ten generations of each TWEANN algorithm being implemented in the Flappy Bird game.

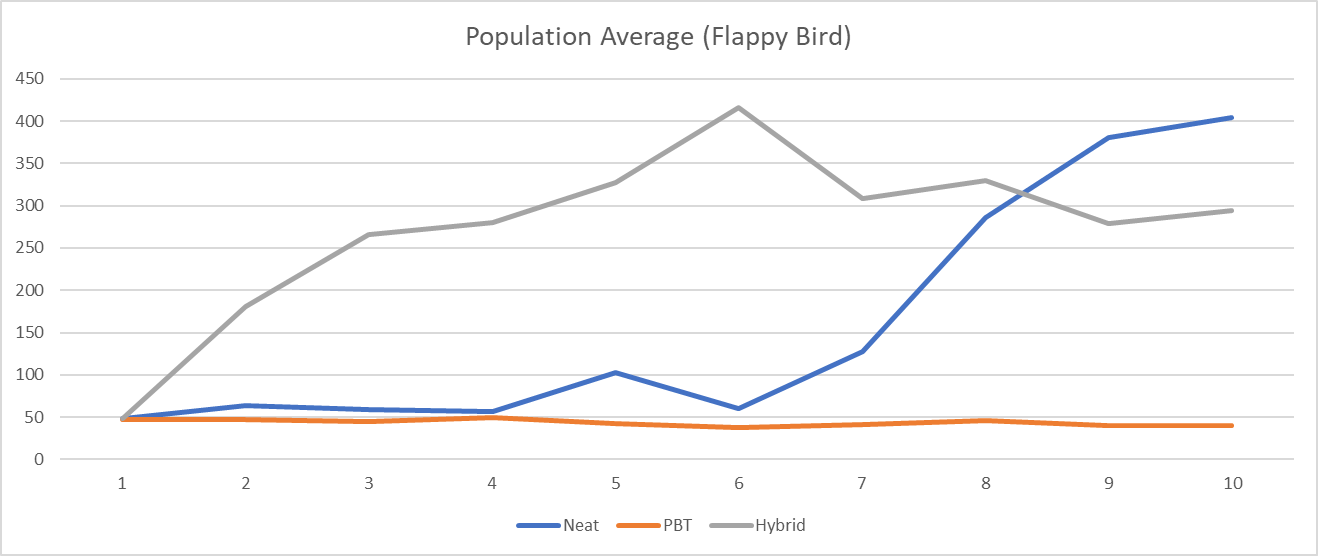


Figure 17: Full size version of figure 8.

Population Standard Deviation When Tested in Flappy Bird

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 16.82752 | 49.22482 | 19.42581 | 20.95374 | 263.7059 | 21.05968 | 346.4983 | 708.0437 | 858.6793 | 935.5293 |
| PBT | 14.21732 | 9.69817 | 7.16791 | 14.75358 | 7.26085 | 4.74444 | 8.80816 | 31.55635 | 6.85139 | 6.55538 |
| Hybrid | 15.15186 | 537.1058 | 681.5807 | 757.742 | 715.7178 | 866.612 | 810.2556 | 919.8322 | 758.1006 | 778.2576 |

Table 3: The population's standard deviation as recorded across ten generations of each TWEANN algorithm being implemented in the Flappy Bird game.

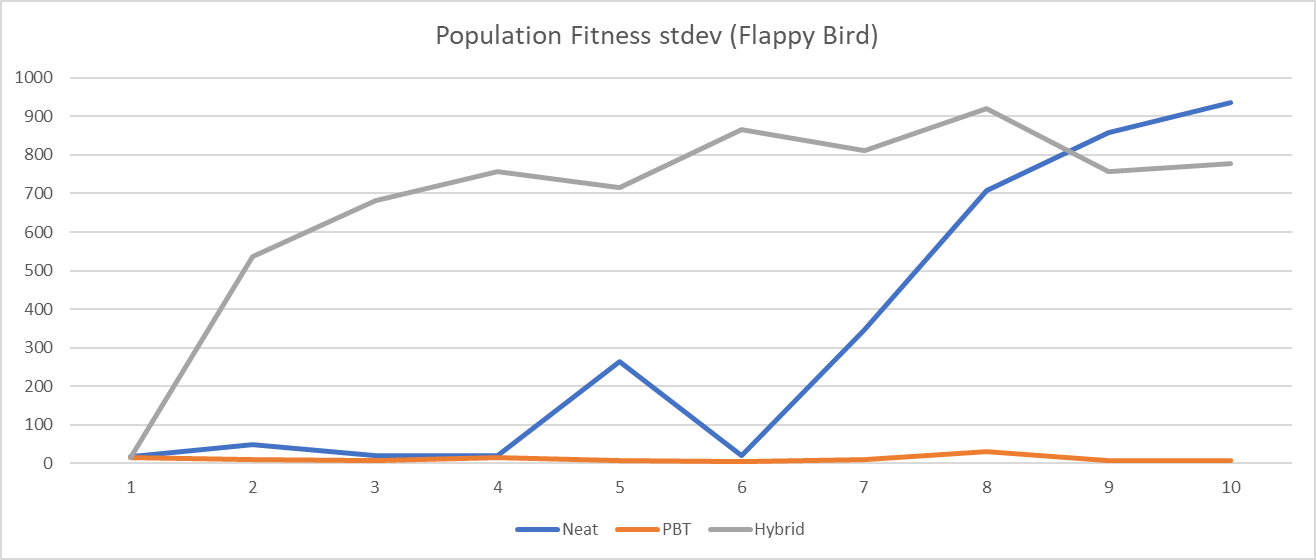


Figure 18: Full size version of figure 9.

Population best recorded fitness when tested in the Go-Kart game

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 0.431 | 0.744 | 0.81 | 0.81 | 0.92 | 1.023 | 1.034 | 1.045 | 1.07 | 1.25 |
| PBT | 0.347 | 0.315 | 0.107 | 0.047 | 0.055 | 0.247 | 0.047 | 0.047 | 0.047 | 0.047 |
| Hybrid | 0.363 | 0.377 | 0.377 | 0.572 | 0.683 | 0.721 | 0.91 | 1.012 | 1.093 | 1.431 |

Table 4:The population best fitness as recorded across ten generations of each TWEANN algorithm being implemented in the Go-Kart game.

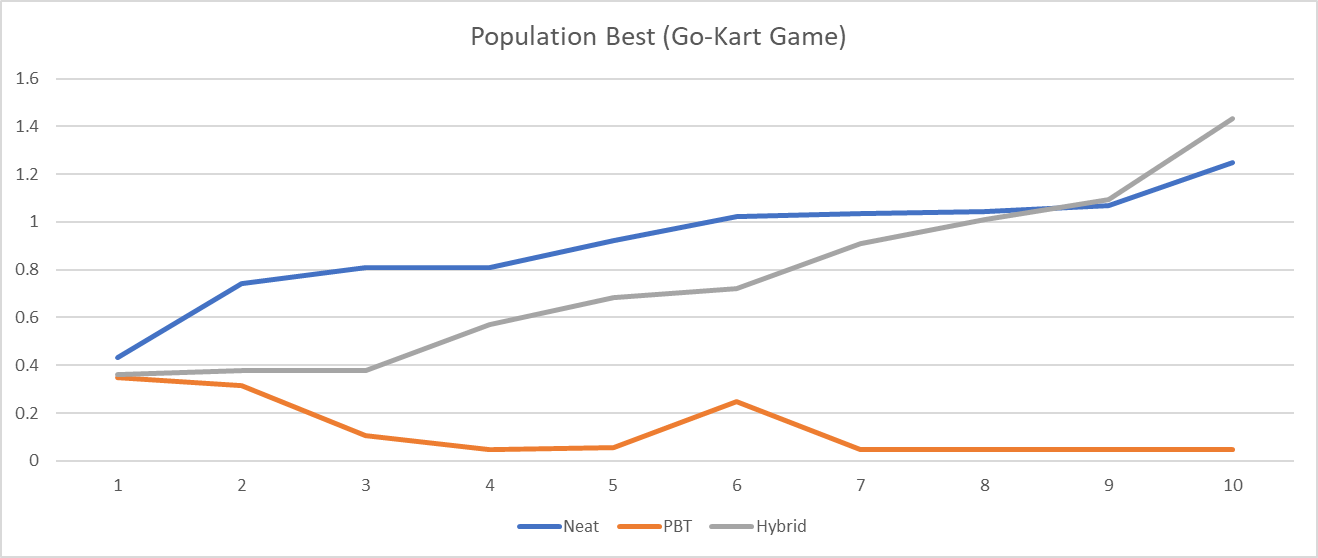


Figure 19: Full size version of figure 10.

Population average recorded fitness when tested in the Go-Kart game

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 0.06498 | 0.1307 | 0.14097 | 0.19827 | 0.21345 | 0.24567 | 0.23543 | 0.29785 | 0.32456 | 0.35783 |
| PBT | 0.0596 | 0.09527 | 0.02482 | 0.02917 | 0.03508 | 0.0822 | 0.0297 | 0.04632 | 0.03685 | 0.0405 |
| Hybrid | 0.05473 | 0.12025 | 0.14838 | 0.18215 | 0.21432 | 0.22467 | 0.23421 | 0.26783 | 0.27845 | 0.29831 |

Table 5: The population average fitness as recorded across ten generations of each TWEANN algorithm being implemented in the Go-Kart game.

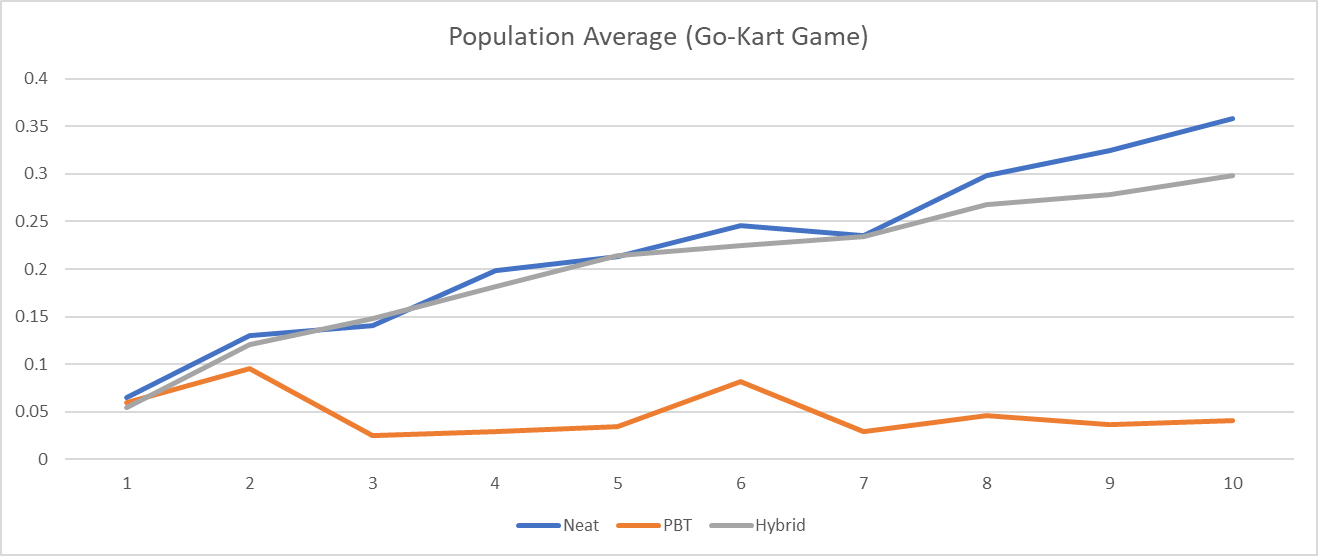


Figure 20: Full size version of figure 11.

Population standard deviation when tested in the Go-Kart game

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 0.08573 | 0.14975 | 0.17666 | 0.20593 | 0.22341 | 0.23412 | 0.25563 | 0.24843 | 0.26326 | 0.28625 |
| PBT | 0.07254 | 0.10646 | 0.01791 | 0.01456 | 0.01484 | 0.07819 | 0.01401 | 0.00407 | 0.01383 | 0.01236 |
| Hybrid | 0.07678 | 0.11624 | 0.13094 | 0.16153 | 0.19763 | 0.21223 | 0.23498 | 0.24324 | 0.23523 | 0.24553 |

Table 6: The standard deviation of the fitness values of the population of network models created by each TWEANN algorithm across ten generations in the Go-Kart game.

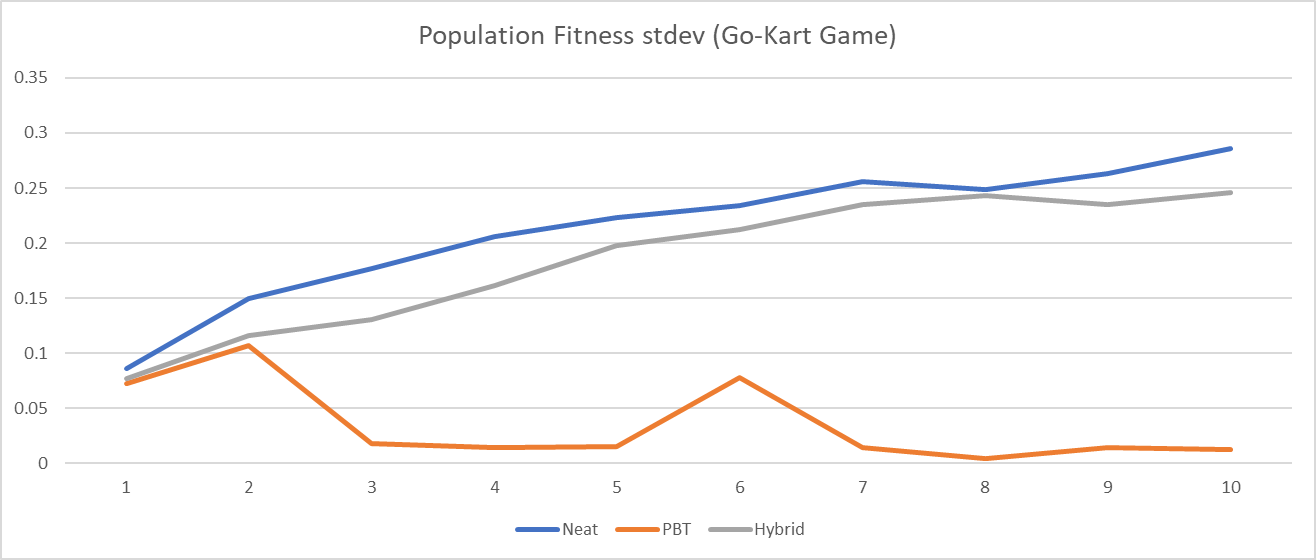


Figure 21: Full size version of figure 12.

Population best recorded fitness when tested in the Cart-Pole game

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 |
| PBT | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 |
| Hybrid | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 | 0.289 |

Table 7: The best fitness values of the populations of network models created by each TWEANN algorithm across ten generations in the Cart-Pole game.

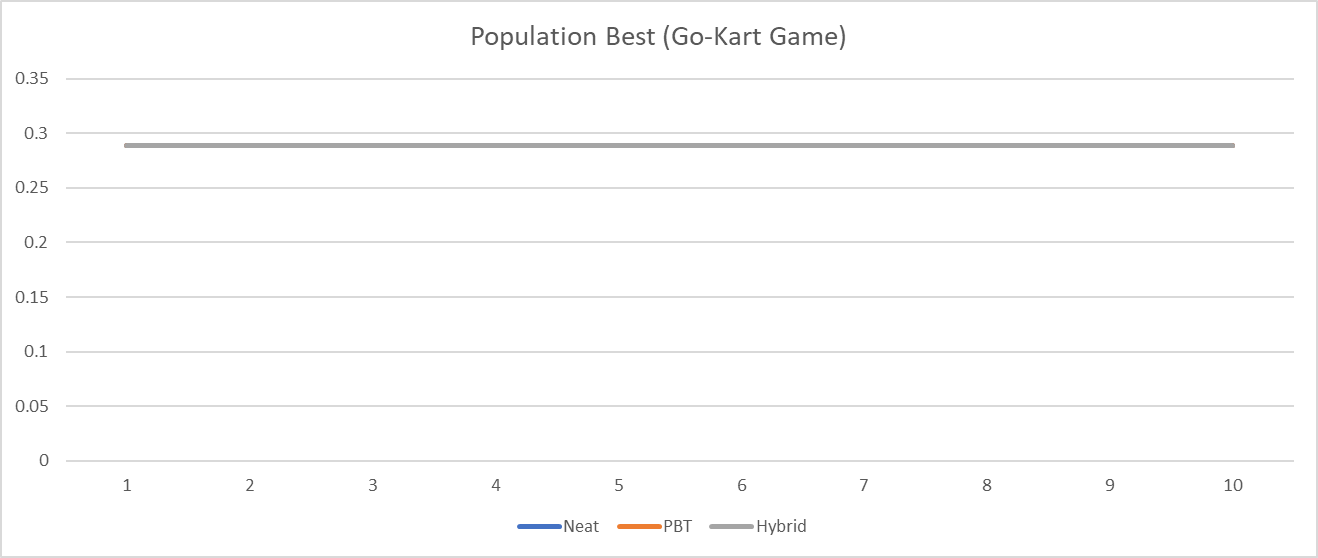


Figure 22: Full size version of figure 13.

Population average recorded fitness when tested in the Cart-Pole game

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 0.09727 | 0.21335 | 0.22835 | 0.20302 | 0.21247 | 0.1931 | 0.20827 | 0.18686 | 0.1678 | 0.22335 |
| PBT | 0.12245 | 0.10231 | 0.16259 | 0.2082 | 0.27388 | 0.26884 | 0.28396 | 0.289 | 0.289 | 0.20296 |
| Hybrid | 0.10731 | 0.19306 | 0.21329 | 0.21329 | 0.21327 | 0.22843 | 0.23343 | 0.20812 | 0.21824 | 0.24855 |

Table 8: The recorded average fitness value of the population of network models created by a TWEANN algorithm across ten generations of the Cart-Pole game.

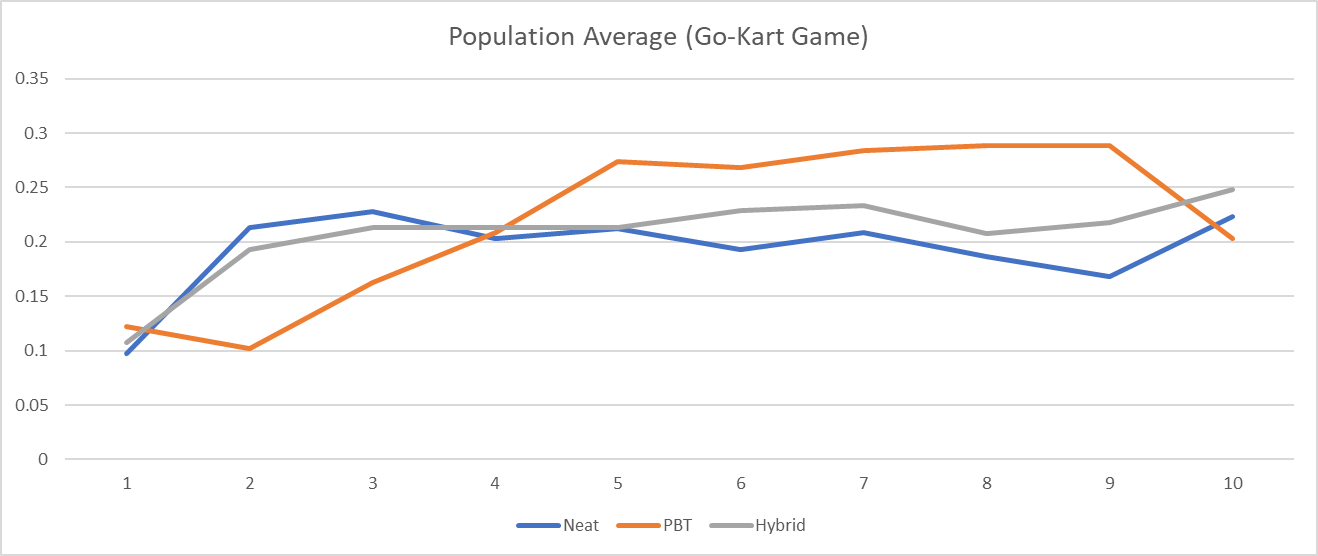


Figure 23: Full size version of figure 14.

Population fitness standard deviation recorded when tested in the Cart-Pole game

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Neat | 0.10316 | 0.1139 | 0.1065 | 0.11767 | 0.11351 | 0.12051 | 0.11595 | 0.12068 | 0.1237 | 0.10925 |
| PBT | 0.11597 | 0.10632 | 0.12385 | 0.11603 | 0.05922 | 0.06763 | 0.03492 | 0 | 0 | 0.11805 |
| Hybrid | 0.10919 | 0.12055 | 0.11399 | 0.11399 | 0.11402 | 0.10636 | 0.10329 | 0.11615 | 0.11187 | 0.09157 |

Table 9: The standard deviation of the populations of network models fitness as created by a TWEANN algorithm across ten generations of the Cart-Pole game.

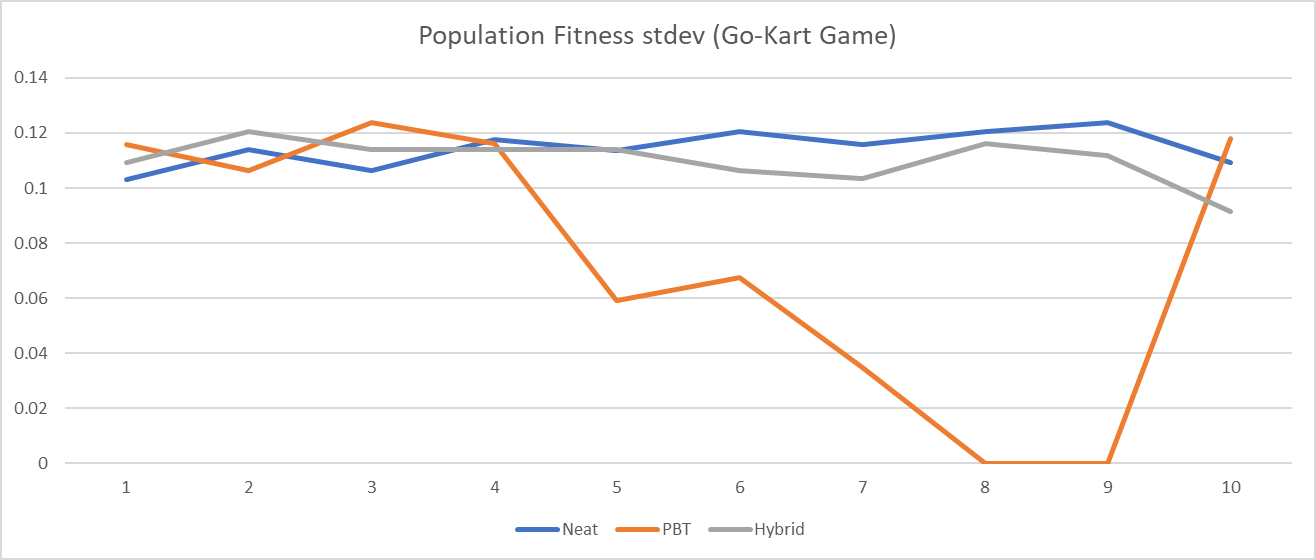


Figure 24: Full size version of figure 15.