UNIVERISTY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Advanced Computing and Informatics Systems»

ΠΜΣ «Προηγμένα Συστήματα Πληροφορικής»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Development of a software tool for the emulation of automatic video-game playing based on the Artificial Evolution of Neural Network Topologies** Ανάπτυξη Εργαλείου Λογισμικού για την Εξομοίωση Αυτοματοποιημένου Ελέγχου Βιντεοπαιχνιδιών βασισμένο στην Τεχνητή Εξέλιξη Αρχιτεκτονικών Νευρωνικών Δικτύων
Student'sname-surname: Ονοματεπώνυμο Φοιτητή:	**Georgios Bardis** Γεώργιος Μπαρδής
Father's name: Πατρώνυμο:	**Andreas** Ανδρέας
Student's ID No: Αριθμός Μητρώου:	**ΜΠΣΠ 16021**
Supervisor: Επιβλέπων	**Dionisios Sotiropoulos, Assistant Professor** Διονύσιος Σωτηρόπουλος, Επίκουρος Καθηγητής

2021/ Δεκέμβριος 2021

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Dionisios Sotiropoulos **Assistant Professor**	**Efthimios Alepis** **Associate Professor**	**Evangelos Sakkopoulos** **Assistant Professor**
Διονύσιος Σωτηρόπουλος Επίκουρος Καθηγητής	Ευθύμιος Αλέπης Αναπληρωτής Καθηγητής	Ευάγγελος Σακκόπουλος Επίκουρος Καθηγητής

1. ABSTRACT

Artificial intelligence (A.I) has become an important part of game industry and in most cases it determinate the game design. The automation in game it is needed for the game to become more entertained for the player. The evolution of the games is often base on how efficient is the gameplay of the non-player characters (NPCs) that the player must defeat to complete the game. A.I is not used only in game but in military, medical, corporate and advertising applications. Game A.I is the effort of game industry to surpass the scripted gameplay of the games as far as the NPSs interaction how sophisticated is the environment and how complex the system of the game can become and how to make it as interactive as possible for player. System like that learn from user input, user inputs ca be certain action such as how the player fights what button it prefers to use or how the player it interacts with the environment as whole. These inputs help the algorithm to evolve and adapt his own non-prescript behavior and develop new techniques for the NPCs to interact with player. Another use of A.I in games is to create an NPC that learn from other NPC how to play, after some generations of training has the knowledge how to defeat specific NPCs. The goal of our research is the second example how to use artificial intelligent techniques to train an NPC to defeat another NPC in battle. In this paper we present a genetic algorithm (GA) for the generation of evolving artificial neural networks called NEAT and we use this algorithm in a game called street fighter. We will discuss how this algorithm can train an NPC to defeat other NPC in battle.

2. INTRODUCTION.

Artificial intelligence in games:

The term game A.I is referring to a set of algorithms that can generate intelligent behavior in non-player characters (NPC's). Artificial intelligent is a concept that goes back on 1950s (NIM). As the game industry starts to evolve and the game are become more complex algorithms has started to become more sophisticated to keep the player attention on the game. Modern games have environments populated with characters and objects of many types and all these objects need of human level intelligence. In recent years game developers focus of creating highly efficient A.I in small subsystems such as pathfinding, decision making etc.

Artificial intelligent in game does not mean necessary that we need a model that needs to learn from players interactions only, for example developers can create a set of possible moves or events for the NPCs that can perform in specific situations or events as the game evolves. This model called finite-state machine (FSM) or finite-state automation (FSA, plural: automata), finite automation, or simply a state machine. This model can have infinite states at a given time and based on interaction of the player or environment if the condition is satisfied then chooses the state to play. But this model as we understand can be very difficult to implement it in a huge environment with many objects and players and with millions of decisions that must be taken so the game progress. An example case that these algorithms are not feasible is the strategy games imaging after few rounds of gameplay the player will understand every move of enemy A.I and the game experience will become repetitive. In early game development it was a fine solution but in early games it can be very difficult to use in every situation. Nowadays developers are using more complex A.I in games, this complexity

makes the games more engaging and attractive to the end user. The solution for this problem it came from the Monte Carlo Search Tree (MCST) algorithm which is a is a heuristic search algorithm, which uses randomness for deterministic problems difficult or impossible to solve using other approaches. This algorithm prevents the repletion of FSM, MCST has all the set of moves from the NPC and visualize them, afterwards for every possible move of the player analyzes every possible response and after consideration its response to players moves.

Modern games use Behavioral Decision Trees. The first time they are evaluated they begin from the root and each child is evaluated from left to right. Child nodes are ordered by priority. If all a child node's conditions are met, its behavior is started. When a node begins a behavior, that node is set to 'active'. The next time the tree is evaluated, it again checks the highest priority nodes, then when it comes to a 'running' node, it knows to pick up where it left off. The node can have a series of actions and conditions before reaching an end state. If any condition fails, then the algorithm returns to the parent node. The parent selector moves on to the next priority node child. This algorithm provides more flexibility to create more interactive A.I but this algorithm gives the illusion that NPCs are learning from our moves but the only thing that happens is that simply checks the nodes to which node is open to make decision based on the players move or decisions.

To achieve adaptive behavior in games we must use genetic neural networks. To start training such a network we first collect all possible moves that the player can use ex. (Map all controller buttons to specific moves or actions) or specify environment parameters such if go this way you will fall and die all this information are called inputs. Also, we must decide when the game could end (Player dies) and of course what is the goal of game for example in our experiment the goal is to defeat the enemy player by hitting him and it loses all his life (hit points). To be more accurate every goal that we give to our algorithm we must assign a score for each one. This way the algorithm can rank up and avoid mistakes that can cause the end of the game.

The market of game industry is high competition between games of the same genre so for the companies and for developers the A.I is selling point. However, for a team of developers to create such a complex model has many constrains. Because games use high amount of CPU and GPU power so they can run graphics operations and environmental operation the amount of CPU and GPU power remains for the A.I is limited. Imagine games like Civilization which in every round must process thousands of moves and decision for every function in round.

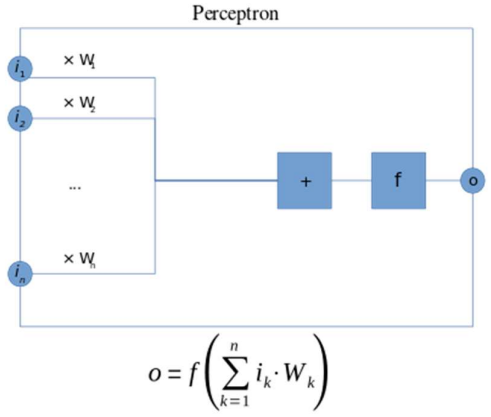
Neural Network:

What could be a neural network? A neural network is a series of algorithms that endeavors to acknowledge underlying relationships in an exceedingly set of knowledge through a method that mimics the approach the human brain operates. Neural networks will adapt to ever-

changing input thus, the network generates the most effective potential result while not having to revamp the output criteria. The conception of neural networks, that has its roots in computing. A neural network just like the human brain's neural network. A "neuron" in an exceedingly neural network could be a mathematical relation that collects and classifies info consistent with a design. A neural network contains layers of interconnected nodes. every node could be a perceptron and is sort of a multiple rectilinear regression. The perceptron feeds the signal created by a multiple rectilinear regression into associate degree activation operate which will be nonlinear. In an exceedingly multi-layered perceptron (MLP), perceptrons are organized in interconnected layers. The input layer collects input patterns. The output layer has classifications or output signals to that input patterns might map. Hidden layers fine-tune the input weightings till the neural network's margin of error is token. it's hypothesized that hidden layers extrapolate salient options within the computer file that have prognostic power relating to the outputs. This describes feature extraction, that accomplishes a utility like applied math techniques like principal part analysis. Neural networks have several applications today in nearly each field we will use them like enterprise designing, trading, business analytics etc.

Neural Network Architectures:

1. Perceptrons computational models of a single neuron. Perceptron was originally coined by Frank Rosenblatt in his paper, "The perceptron: a probabilistic model for data storage and organization within the brain". Perceptron additionally called feed-forward neural network. To train a perceptron needs a back propagation, we must give to algorithm a pair dataset of input and outputs. The use of perceptron is limited but we use it by combined them with other neural networks.



Perceptron (ref: [21])

Figure 1. Perceptron

2. Convolutional Neural Networks uses the back propagation in feedforward net with several hidden layers. Each neuron cell take inputs and follows it with a non-linearity. The whole network expresses a single differentiable score function. Convolutional

Neural Networks are quite different from most other networks. They are primarily used for image process however also can be used for alternative varieties of input like audio.

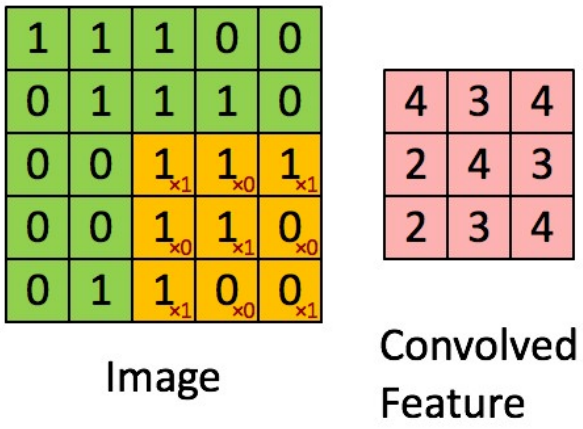


Figure 2. Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature (ref: [22])

3. Recurrent Neural Networks originally introduced in Jeffrey Elman's "Finding structure in time" (1990). It is basically perceptrons, the difference between perceptrons it is perceptrons are stateless. Standard neural networks have fixed size in vectors as input which is a limit in usage in many situations. RNNs are very powerful because of the hidden state they have which help them to store more data and have more efficiency than the vanilla neural networks. Also, they are nonlinear and this give them for dynamic update of their hidden neurons.

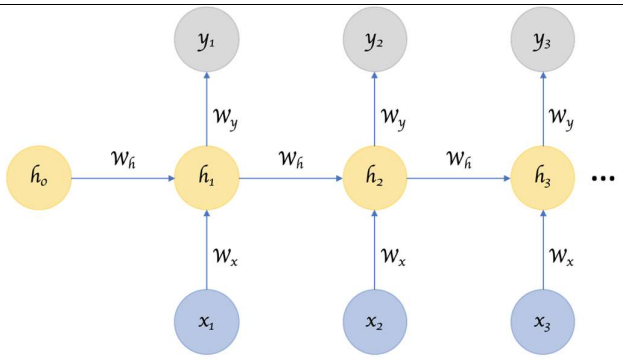


Figure 3. A Recurrent Neural Network (ref: [23])

4. Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. LSTM has a feedback connection and this helps because it cannot only process single data point but can process entire sequences of data points (ex. Sound or video). LTSM try solving the problem vanishing exploding gradient problem by introducing gates and an explicitly defined memory cell. Memory cell does not forget their values until "forget gate" tells them to do it. LSTM has also the input gates that can introduce new values to the cells and output

gate which determine when to pass data from vector to cell and the two next hidden.

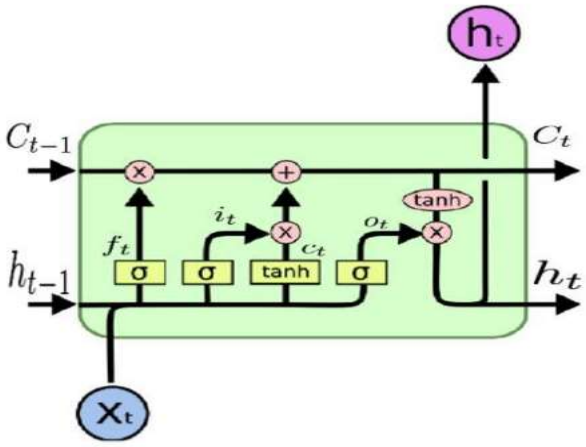


Figure 4. Long Short-Term Memory (LTSM) (ref: [27])

5. Gated Recurrent Unit. GRUS are and improvement of standard recurrent neural network. The difference is that can keep the information from long time before. This helps because they are not losing any info through time or remove any info even be irrelevant to prediction.

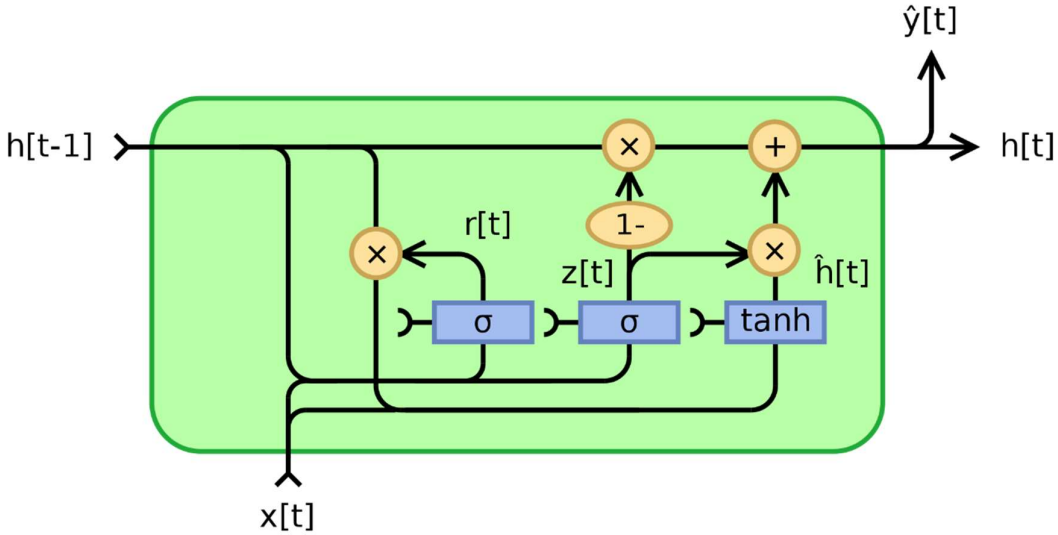


Figure 5. Recurrent neural network with Gated Recurrent Unit (ref: 24])

6. Hopfield Networks. It is consistent by many percetrons and can overcome the XOR problem. All the neurons are fully connected with each other. Every node get an input before trainning, the networks are training by getting values of every neuron and after which weights are calculated. But the weights are not change their values after this but only after trained for one or more patterns. The network can invariably converge to at least one of the learned patterns. The problem hotfield networks is that are very

limited in capacity, can only memorize 0.15N patterns in it's energy function.

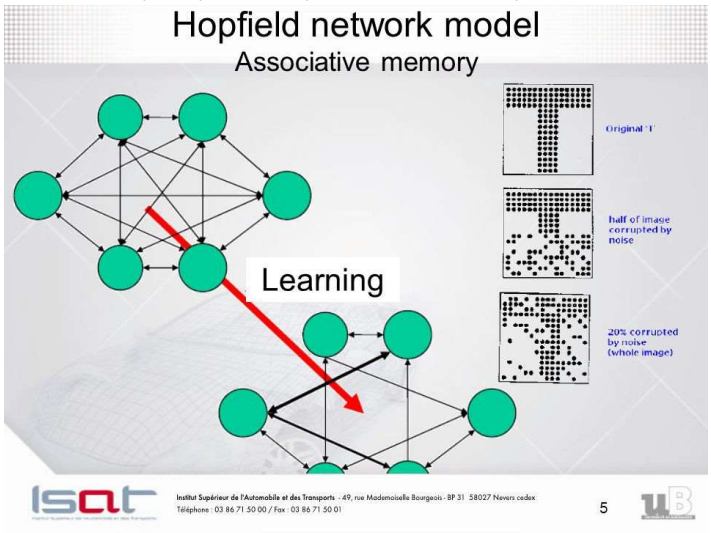


Figure 6. Hopfield network model (ref: [5])

Genetic algorithms, according to [2,6]:

A genetic algorithm is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. The process of the natural selection works by selecting the best individual from the current population. The produced offspring's that inherits from their parent are selected and added to the next generation. The fitness it is the factor the factor that make the offspring's better, if the parent has a good fitness their children would have better. This process keeps on until the fittest individual is found.

The process begins by selecting the starting population for the algorithm using default or random values. Each individual has a set of variables called genes each gene run through from a fitness function. After each iteration fittest of the individuals are selected from the population and added to the next generation using the reproduction function, this functionality is repeated for defined number of times and the end the algorithm presents the best of the population according to the fitness function. Let us discuss each of these concepts further.

1. Fitness function. This function it fulfils the criteria of the algorithm that helps with the reproducing with fittest of the individual of every generation. For each iteration it assigns a fitness score for every individual and this determines if the individual will pass to the generation.

2. Selection function. Gets as arguments the population and the results of the fitness function and base on how we want to reproduce the population it selects the individuals.
   For example the selection function can select the individuals by value if the select function returns a Boolean value. If the selection function gets raw values, it can calculate the average of the score and keep only a percentage of the population and finally passes the remaining population in the reproduction function.
3. Reproduction function. It handles the expand of the population based on the existing members and determines how the population of the change over time. It is most complex part of the algorithm and has a significant impact of creating the algorithm. The reproduction of the population can be achieved with mutation or crossover or both combined.
   a. Mutation is where every new member of the population is created base on single individual, for each new individual create a new one with the same characteristics.
   b. Crossover. Is the more complex of the two methods because it is based on combination of the existing individuals. Crossover combine the attributes of individuals but does so by applying a function of multiple organisms' attributes.
4. Termination function. The role of this function is to get the final population and return the best members base on the fitness score. The role of termination function it based on purpose of the algorithm

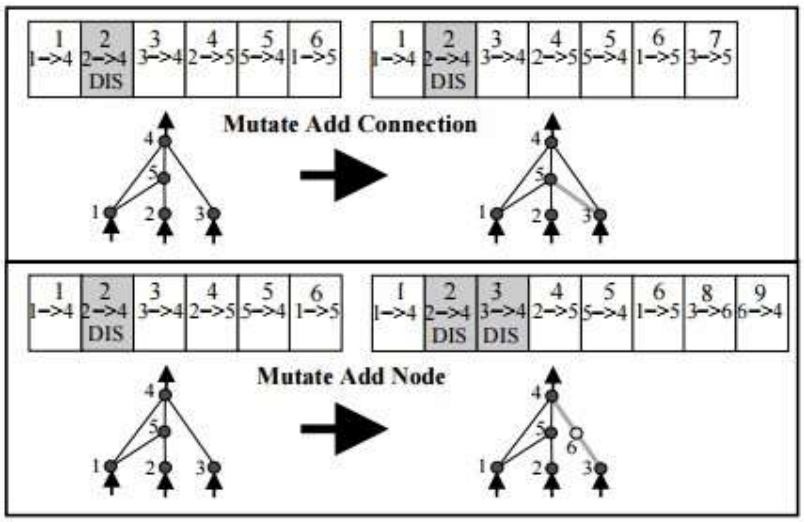
3. NEUROEVOLUTION OF AUGMENTING TOPOLOGIES (NEAT)

According to [25,7]

Neat is a genetic algorithm which is used for creating evolving artificial neural networks develop by Ken Stanley in 2002. It can change the weight parameters and the structure of network, in the attempt to find the balance between the fitness of evolved solutions. NEAT implements the idea that if you can start the evolution with small and simple networks and allowed them to become more complex after a defined number of iterations(generations).

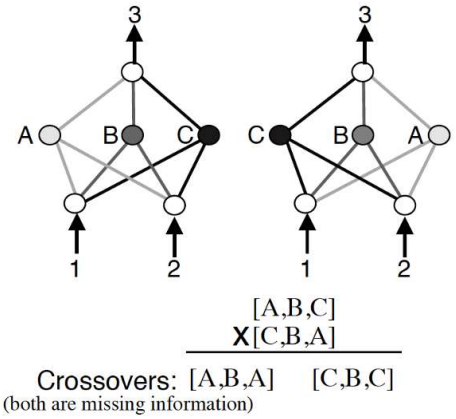
Encoding, in biology we have genotype and phenotype. A genotype is the genetic representation of the creature and phenotype is the actual creature. NEAT comes with a question how we want to represent this biology terminology in the algorithm. The way to handle the evolution in the algorithm is to handle the process of selection, mutation and crossover. Any encoding will use one of these categories direct or indirect. Direct encoding if we are talking about a neural network this mean that every node will link with to a node, connection of network. This means that always will be a direct connection between the genotype and phenotype. Indirect encoding, we specify parameters for creating the new gene. Indirect approach is more difficult to follow with knowing very well how the encoding will be used. The NEAT algorithm chooses a direct encoding.

Mutation, in NEAT mutation can mutate an existing connection or add a new into the network. If a new connection is assigned, it is randomly assigned a weight. If a new node is added between the two old nodes, all the previous connection is removed. The new start node is assigned with the weight of the old one and linked to previous old node with the weight of 1.



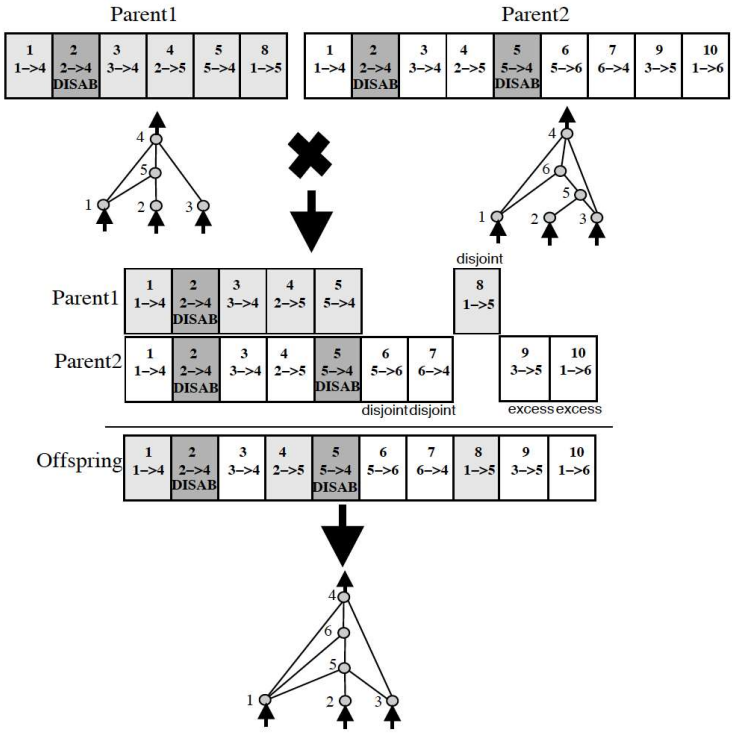
(ref: [26]Heidenreich, H., 2021. NEAT: An Awesome Approach to NeuroEvolution. Hunter Heidenreich (http://hunterheidenreich.com/blog/neat-an-awesome-approach-to-neuroevolution).)

Competing Conventions, the idea behind it is that just blindly crossing over through two neural networks can result to bad mutated and non-functional network. The case is that if these two networks are dependent on their central node and are both recombined out of the network, it will face a problem.



(ref: [26])

NEAT tackles this problem by assigning historical values to the new evolution when it is time for crossover, this reduces the chance of creating no-functional individuals.



ref: [26]

Speciation, Neat suggest speciation for protection of the new structures and have better optimization before the algorithm wipe out the entire population. How speciation works, splits up the population into several species based on topology and connection. Because NEAT uses historical markings in its encoding, this helps for measurement to be easier. Every individual in population must compete with other individuals with the same species. This helps structure to be more optimized with losing any case before eliminating the population. Also, NEAT has fitness sharing between species and this helps to improve the performance of the new species and have better optimization before being evolved.

A large goal of NEAR algorithm is that allowed minimal network to be evolved. The creator did not create the algorithm to found first the all the good networks and after try to reduce the size of the network, instead NEAT starts with minimal number of nodes and connection. So, the complexity of NEAT evolves as the time goes on and keep only the necessary. This can be achieved by creating a network without any hidden nodes. Every individual in the initial population is an input node or an output node, this with the help of speciation evolving minimal and high performing networks.

Now we describe how NEAT genetic encoding works and how solves the problems that specifically addresses.

Genetic encoding is a scheme that allows corresponding genes to easily line up when other genomes are in the process of crossover. Genomes are linear representations of network

connectivity. Every genome has a list of connection and each connection refers to two node genes that are connected. Node genes have in-node out-node and a specified weight for each connection when a connection gene is imminent to be created and an innovation number is assigned which allows to find the corresponding genes.

Mutation in NEAT changing connection weights and the network structure. Mutation occurs in two ways. By adding a connection mutation and adding a node mutation. Connection mutation adds a connection gene with random weight in two previously unconnected nodes. Adding a new node, it can be happened by splitting an existing connection and the new placed where the old connection used to be. The previous connection is disabled, and the new connection get the weight of 1 and the node also get the weight of 1. This approach helps the network to be optimized and make use of the structure and in comparison, with other algorithm can immediately make use of the new structure and not waiting for the networks to be evolved first to use the new structure.

Tracking Genes through Historical Markings, this provides the information of which gene match up with which gene between any individual inside the population. This information is coming from the historical origin of the gene. Two gene with the same origin has the same structure but possibly different weights. Thus, the algorithm needs to know which genes has the information of their and which they do not. Tracking historical genes needs computation. When a gene is generated, a global innovation number is incremented and assigned to the gene. A problem that occurs with the global innovation number is that is possible to assign different numbers in the same structural innovation. To resolve that problem, the algorithm keep track of the of innovations that happens in the current generation. Thus, ensure that identical mutation gets the same number.

This historical marking gives to NEAT a new capability to know for every gene which gene match up with which. When crossing over, these genes in both genomes with the same innovation numbers are lined up. These genes are called matching genes. Genes that do not match are either disjoint or excess. When composing the new offspring genes are choosing randomly from their parent at matching genes, whereas all excess or disjoint genes are always included from the more fit parent. This way, historical markings allow NEAT to perform crossover using linear genomes without the need for expensive topological analysis.

Adding new genes to population representing different structures and the system can population from different topologies. This causes problem because this kind of system cannot maintain diverse topological innovations. Smaller structures optimized faster and adding nodes and connections usually decrease the fitness of the network. Recently augmented structures have little hope of surviving more than one generation. The solution to this problem is protect the innovation through speciation.

Protecting Innovation through speciation. The idea is to divide the population into species, but this is a matching problem, and there comes the historical marking to give a solution to this problem. Excess and disjoint genes give us the compatibility distance, the more disjoin are the genomes less evolution history the share, so are more incompatible. To measure the compatibility the algorithm uses the above function:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}.$$

E and dare the excess and the disjoint genes, W is the weight differences of the matching genes, C1 C2 and C3 give us the option to adjust the importance of the 3 factors and finally the N is the number of genes in the larger genome.

Minimizing Dimensionality through Incremental Growth from Minimal Structure. NEAT creates an initial network without introducing any hidden nodes and layers. New structures are incrementally introduced as the mutation is happening and at the end only the structure is surviving can be useful through fitness evaluation. Since NEAT starting with minimal population the search is very efficient so NEAT as more performant and optimized in compare with other approaches like TWEANN.

Performance evaluation of NEAT, to evaluate system performance we use three experiments:

1. The XOR experiment which tests the increasing topology of the algorithm evaluation. To build XOR solving network we should grow new hidden unit in the staring genome. Two inputs must combine to a hidden node and as opposed to only output node. The two inputs should be combined at some hidden unit, opposed to the output node, because there's no operate over a linear combination of the inputs which will separate the inputs into the correct categories. These structural necessities create XOR appropriate for testing NEAT's ability to evolve topology.

2. The single pole-balancing or inverted pendulum problem is a standard experiment for artificial learning problems.

3. The double pole-balancing experiment, this is the advanced form of pole balancing, in which the cart has two poles instead of one with different mass and length to be balanced. Game Simulator Platforms.

4. GAME SIMULATOR PLATFORMS

There are many simulators out there on the internet that simulate many game platforms as PlayStation, Nintendo, android and many more. Each of these simulators are exposing several tools so you can interact with the game and game all the aspects of it such as the inputs of the controller, the speed of the game. Also helps to identify through memory how the AI of the operates and interacts with your inputs, you can identify patterns and spawn times through memory access. All the information is very important to create your AI algorithm, will all that information you can create a proper fitness function and experiment with your AI and how you can defeat your opponent or finish the stage.

Example of simulators:

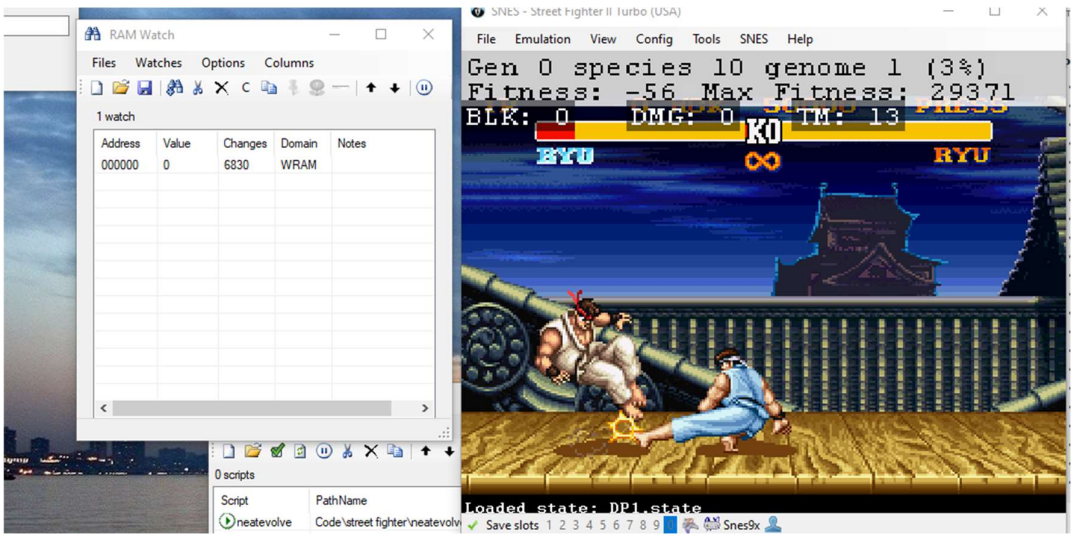
1. Gym retro. You can use gym retro to work on RL algorithms and study generalization. Before that, the focus of the emulator was to optimize agents and solve single task. With this emulator you can study the ability to generalize between games and similar scenarios.
2. AI-TEM: Artificial Intelligence Testbed in Emulator. AI-TEM is an environment created for the aim of testing game AI. it is created from Visual Boy Advance (VBA), an emulator of Nintendo Gameboy Advance game system (GBA). Commercial games will be played on VBA using the games' Roms.
3. Bizhawk is the emulators that we are using for our experiment so with go in more detail above.

For our experiment we use BizHawk emulator, this emulator is multiplatform and support full rerecording support and Lua scripting. The emulator supports almost all platforms as the above image show us.

Supported platforms and platform-specific documentation

- Nintendo Entertainment System, Famicom, Famicom Disk System
- Super Nintendo Entertainment System and Super Famicom
- Nintendo 64
- Virtual Boy
- Game Boy, Super Game Boy, and Game Boy Color
- Game Boy Advance
- Sony PlayStation
- Sega Master System, Game Gear, and SG-1000
- Sega Genesis, Sega-CD
- Sega Saturn
- NEC PC Engine (AKA TurboGrafx-16), including SuperGrafx and PCE CD
- Atari 2600
- Atari 7800
- Atari Lynx
- ColecoVision
- TI-83 graphing calculator
- Neo Geo Pocket
- Wonderswan and Wonderswan Color
- Apple II
- Mattel Intellivision
- Commodore 64
- ZX Spectrum

The core functionality that we use for our experiment is the ram watch that is provided by emulator.



With this functionality we can watch memory address of the current game. This memory addresses provides with very useful information that helps the algorithm to perform. For example, we can get the time, health of the players, distance of the player, damage and many other useful values. Another key functionality of the emulator is the Lua support and console. Also, the way to check what memory we want to watch the memory address that changes during the gameplay and we add value to them we can see them reflect to game itself.

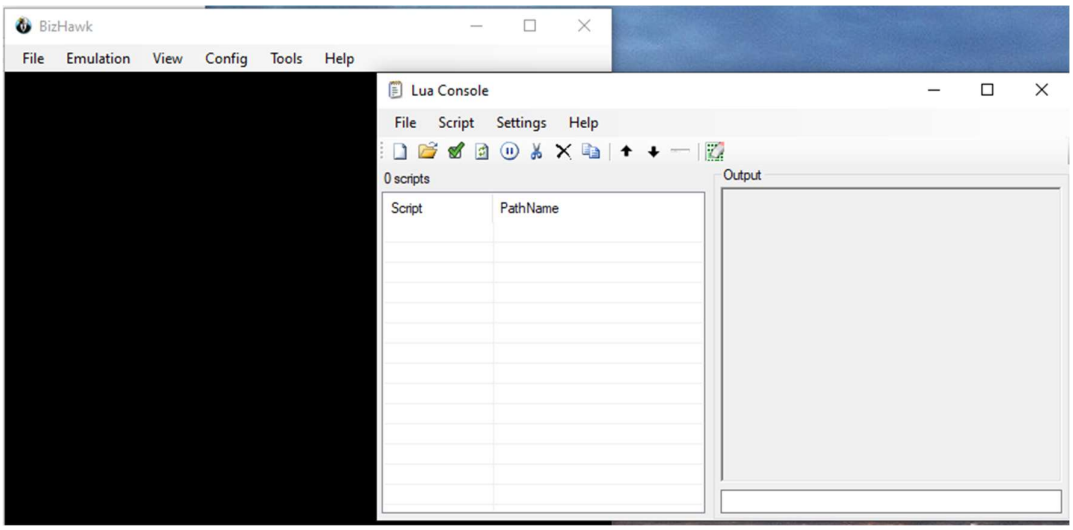
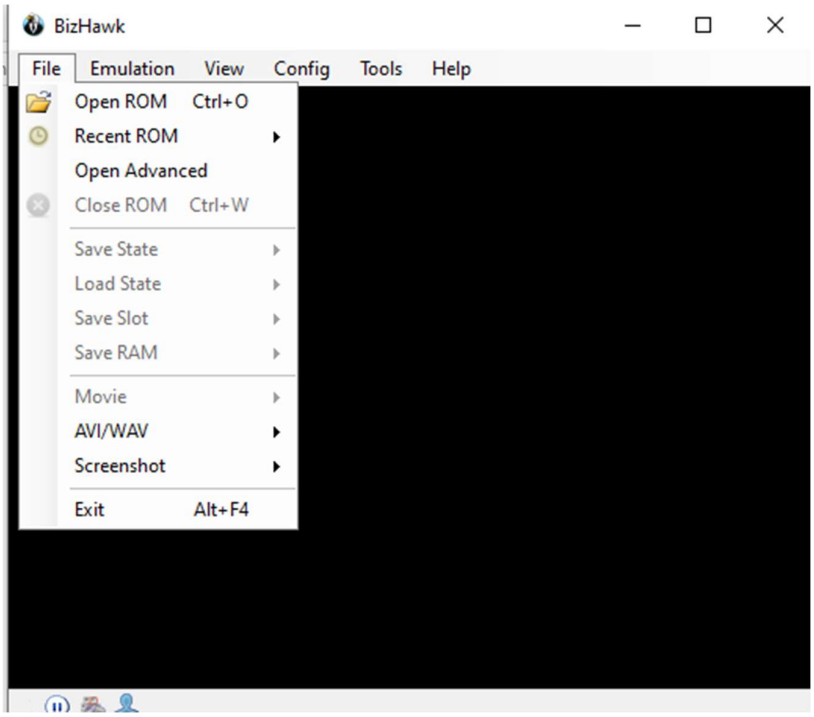


Figure 7. Lua console

With this console we can run any Lua command or script. How its functionality works, as a mention before the emulator provides us with memory access from Lua, as we get the values direct memory the algorithm produces some output moves and at the of this process, we can send through sockets the moves that we want our player to do.

Another key aspect of the emulator is that provides with save state functionality. This functionality is very useful because we want the algorithm to start every time from a specific time and the position of player must be the same for every round.



The emulator also provides us with and API for game controllers. We can connect as many controllers as we want through code and give direct commands through Lua commands.

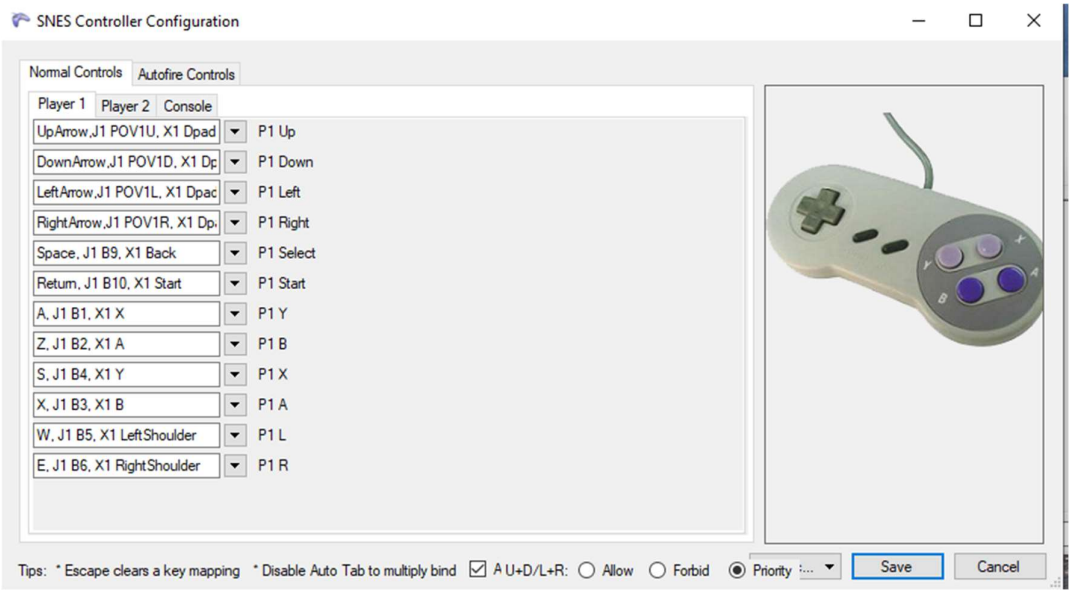


Figure 8. Emulator's controller options

The values that we get from the emulator:

1. Health of the player that the algorithm is handling
2. Position of the player that the algorithm is handling
3. Current state of the player. This mean that we need know each frame if the player is moving or is idle.
4. Where the player is facing
5. Health of the enemy player.
6. Position of the enemy player.
7. State of the enemy player.
8. Distance between the players.
9. Players Keystroke. This info refers to what key is using each frame
10. Players blocking ability. This is a custom input that calculated from players distance the current state and enemy state.
11. Players damaging ability. This is custom input that calculated from players distance and current state and enemy player state and the current damage that our player is doing.
12. Continuous Keystrokes. This is a customer input that calculated from players keystroke and the current state.

The above values are combined to create the fitness function that we use in NEAT. Above we will discuss how we use them in the fitness function.

Fitness functions parameters:

1. myHealth – enemyHealth. This factor it not used if the player does damage, we use this factor only if the player dies without do any damage to enemy player.
2. The most important factor is the player has done damage to enemy player so if the condition is valid. We calculate the remain health of the enemy player (176 - enemyHealth) * (176 - enemyHealth).
3. The remain health of our player fitness + (myHealth * 50). The more our player has the better for the fitness.
4. Calculation the time that the game has taken place fitness - (FrameCounter * 20). If the game holds a lot of time this is bad factor for the fitness.
5. PlayerWasStale is the factor that if the player is not moving has a negative number and if the player is moving has a positive outcome for fitness. If the player is moving, we give to this factor a score of 100 if the is not moving we decrease the value by 100 and final value will be added to the fitness (fitness + PlayerWasStale).
6. PlayerProximityAction is factor that has the same usage as the PlayerWasStale, we want to check if the player is moving but if he moving to make an attack and moving around and doing nothing. So, we check the distance between the players and an attacking distance threshold. If these two conditions are met, we decrease this factor by 30, if our player is moving always towards to enemy player to attack him this factor

does not change at all. So, this factor is a negative one to give penalty to the fitness if the player is not trying to attack.

7. PlayerContinuousKeystrokePunishment, this factor is applied after an observation of the training process. The player in the early state of training was trying to defend himself from attacks by moving up and down continuously, after we observe that pattern decide to put a negative factor to train the network to not use so frequent that pattern, we did want to stop it but reduce the usage of it. So, if the conditions are met, we decrease the value by 20 otherwise we did not change the at all and add the final value to fitness (fitness + PlayerContinuousKeystrokePunishment).
8. PlayerCloseDamageAbility, this factor is about if the player hits are registered or not. If the players hit are doing damage to the enemy player, we increase this value by 100, if the player hits are not registered, we decrease the value by 1, the final value is multiplied by 5. The decrease value is so small because we do not want to stop the player to try to hit but we want to give a push to try to get closer to enemy player so he can register the hit and do damage which is the most import goal of the game. (fitness + PlayerCloseDamageAbility * 5).
9. PlayerCloseBlockAbility, this factor is about blocking. If the player manages to block enemy's hits, we increase the value by 1. The increase of value is so small because the player the most time is on defense position and we do not want to stay always in defense but also try to attack. The final value is multiplied by 5 (fitness + PlayerCloseBlockAbility * 5).
10. PlayerMoveVariety, this factor is not so important as the other to achieve the goal of winning the enemy player but to make more realist the outcome of training. At every game we check the variety of the moves that out player does and if the player is always hit the same key, we give negative value of 200 to the factor, if the player does actually use different keys to hit the enemy the value does not change at all. This way we achieve in the late generations to make the player do combo moves and not only common moves (fitness + PlayerMoveVariety).

---

5. EXPIREMENTAL RESULTS

We are gone discuss some results from neat algorithm after 60 generation. Our first diagram is about the fitness of algorithm as the generations are evolving. It is very clear that as the model is developing the fitness is going better and better. The only spots at this diagram that are interesting is between generation 5-7, 7-9, 15-17 and 19-21. This is the generation that the model is trying new combination of moves against the CPU AI and as we can see clearly through this diagram is that our AI needs about 2 to 3 generation to find the combinations of to gain some results against the CPU. Also, after the 53 generation we can that our model is reaching a pint that it cannot evolve and more based on the fitness function that we proved

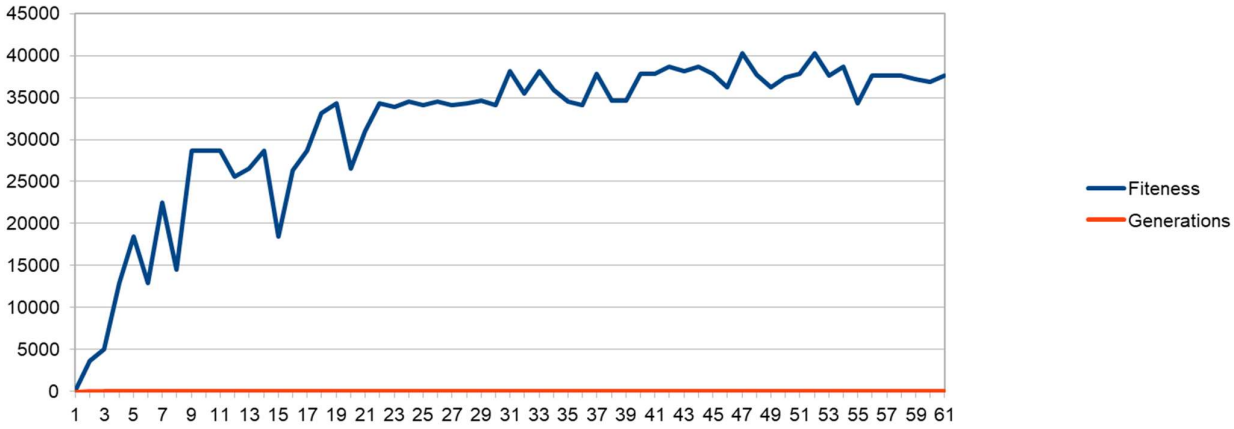


Figure 9 Fitness by generation.

In our second diagram we can the time that our AI is surviving against the CPU AI. As the model evolving and trying new moves so there are spices because is losing very quickly. Also, we can see that it is consistent above the 800ms of surviving time. This can be improved by giving more credit to our model about being more defensive than attacking, our maybe we can find the sweet spot about the attacking and defensive behavior of our model.

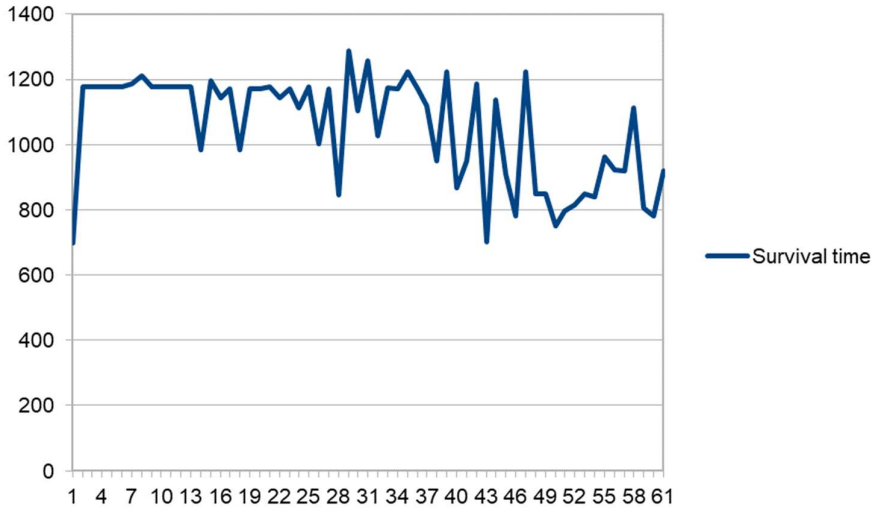


Figure 10 Fitness by survival time.

6. CONCLUSION

The research aimed to find the usability of NEAT algorithm to game AI and how it can make a model that can be efficient against the CPU AI and defeat opponent after 60 generations,

based on same factors that are given as inputs to the algorithm. The result indicated that neat algorithm could create a model that can defeat the AI of this game, also it can defeat different opponents, in normal difficulty. This is a very good result because this a fight game and there are many aspects that can make a player defeat the opponent as such defense, movement and attacking which can be split in multiple combinations with each other. This indicates that we can manipulate the model to work as we want, for example we can make the AI to be more defensive or attacking or we can make the player to learn special combos. By analyzing the results, we can learn that if we change the weight of the input that given to algorithm the AI plays very differently. In this case we have given more focus to player attacking the opponent, the fitness factor that give that result is PlayerWasStale, PlayerProximityAction. The PlayerWasStale is the input that indicates the movement of the player, if the player is moving towards the enemy, we give positive score else we penalize if not and PlayerProximityAction is about the movement again but it is mandatory to try to fight the opponent and not to move without any action. This is the inputs that balances the AI behavior, after many runs of the algorithm if this two inputs does not exists the player will be only defensive, because in the beggining our player will be doing nothing but stand still and not trying to hit his opponent efficiently, the only positive input will be defense because to defend is more easy than the offence, the AI will eventually will be doing only defense, so this two inputs is a way to give to the algorithm a way to start thinking towards not only to defense but to offence also.

Another good aspect of this implementation is that NEAT can be training more efficiently and make the player almost impossible to be defeated. This can be achieved by training the AI to use more efficient moves and counter attacks. A fitness factor that helps us to achieve more variety moves is PlayerMoveVariety. This factor penalizes the player if hit the key repetitively, with this input we have achieved to do special moves against the enemy player. So, if this factor goes a step further, we can make the AI to learn how to counter efficiently, or make it to learn more combos if we give positive result if the AI hits a combo of keys that can register more damage to enemy player.

To sum up NEAT it can be uses as AI in games, as my research indicates it can be used also in many different games to solve any kind of scenario to defeat or complete a scenario of a game. The only limitation a have found is the variety of tools that can be found to run a game and get the information which is important to run the algorithm. Also read memory to get the values can be difficult sometimes based on the complexity of the environment of the game that you are running this goes also to writing into the memory if it is needed to.

Based on the conclusion, we can train this model with more players as an opponent so will be more efficient to a more variety of scenarios and opponents. Also, a recommendation for further research will be training a different model or the same model to fight each other without making it impossible to defeat each other. Maybe we can train the game with a different algorithm and fight each other to see which of the two is the best for this game specifically.

7. BIBLIOGRAPHY

1. NeuroEvolution can optimize and evolve neural network structure, and the NEAT algorithm was one of the first to show it as a viable approach! (https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f)
2. Kiely, P., 2015. EVOLUTIONARY ALGORITHM-Introduction to Genetic Algorithms, (https://blog.floydhub.com/introduction-to-genetic-algorithms/)
3. Knafla, B. Introduction to Behavior Trees (https://web.archive.org/web/20131209105717/http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/)
4. Konstadinov, S., 2017. Understanding GRU Networks (https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be)
5. Le, J., 2018. The 10 Neural Network Architectures Machine Learning Researchers Need To Learn (https://data-notes.co/a-gentle-introduction-to-neural-networks-for-machine-learning-d5f3f8987786)
6. Mallawaarachchi, V., 2017. Introduction to Genetic Algorithms — Including Example Code (https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3)
7. Omelianenko, I., 2018. Performance evaluation of NEAT algorithm's implementation in GO programming language. (https://www.researchgate.net/publication/325854836_Performance_evaluation_of_NEAT_algorithm%27s_implementation_in_GO_programming_language)
8. Pfau, V., Nichol, A., Hessel, C., Schiavo, L., Schulman, J., Klimov, O., 2018. Gym Retro (https://openai.com/blog/gym-retro/)
9. Shummon Maass, L. and Luc, A. 2019. Artificial Intelligence in Video Games
10. An overview of how video game A.I. has developed over time and current uses in games today (https://towardsdatascience.com/artificial-intelligence-in-video-games-3e2566d59c22)
11. Stanley, K. and Miikkulainen, R., 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2), pp.99-127.
12. Venkatachalom, M., 2019. Recurrent Neural Networks -Remembering what's important (https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce)
13. Wittek, P., 2014. Pattern Recognition and Neural Networks (https://www.sciencedirect.com/topics/computer-science/hopfield-network)
14. https://cs231n.github.io/convolutional-networks/
15. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
16. https://en.wikipedia.org/wiki/Finite-state_machine
17. https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games
18. https://gamedev.stackexchange.com/questions/51693/difference-between-decision-trees-behavior-trees-for-game-ai
19. https://www.geeksforgeeks.org/genetic-algorithms/
20. https://sites.google.com/site/fightinggameai/ai-tem
21. https://el.wikipedia.org/wiki/Perceptron
22. Morioh.com. 2021. Social Network for Programmers and Developers (https://morioh.com/p/e1778f1e1b66).
23. Venkatachalam, M., 2021. Recurrent Neural Networks - Remembering what's important - gotensor. (https://gotensor.com/2019/02/28/recurrent-neural-networks-remembering-whats-important).
24. https://en.wikipedia.org/wiki/Gated_recurrent_unit
25. Le, J., 2021. The 8 Neural Network Architectures Machine Learning Researchers Need to Learn - KDnuggets.(https://www.kdnuggets.com/2018/02/8-neural-network-architectures-machine-learning-researchers-need-learn.html/2).

26. Heidenreich, H., 2021. NEAT: An Awesome Approach to NeuroEvolution. Hunter Heidenreich (http://hunterheidenreich.com/blog/neat-an-awesome-approach-to-neuroevolution).
27. Medium. 2021. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. (https://medium.com/@nehapatil364/fundamentals-of-recurrent-neural-network-rnn-and-long-short-term-memory-lstm-network-b53f747fefaf).