THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

A Comparison of Performance between DQN and NEAT/Hyper-NEAT in Game Environments

by

Chunzi Lyu

School of Information Technology and Electrical Engineering,

University of Queensland.

Submitted for the degree of Master of Information Technology

in the division of ITEE.

June 12, 2017

Chunzi Lyu

chunzi.lyu@uqconnect.edu.au

Monday, 12 June 2017

Prof Michael Brünig
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia QLD 4072

Dear Professor Brünig

In accordance with the requirements of the Degree of Master of Information Technology in the School of Information Technology and Electrical Engineering, I submit the following thesis entitled

" A Comparison of Performance between DQN and NEAT/Hyper-NEAT in Game Environments"

The thesis was performed under the supervision of Dr. Helen Huang. I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Chunzi Lyu

To

my parents who supported me unconditionally.

Acknowledgements

I would like to express my special thanks to my supervisor Dr. Helen Huang as well as our head of school ITEE Prof Michael Brünig for giving me this golden opportunity to do this wonderful project on the topic of reinforcement learning and evolutionary algorithms, which I am truly interested in.

Secondly, I would also like to thank my parents and friends who have helped me a lot in finalizing this project within the limited time frame.

Abstract

Reinforcement learning algorithms and Evolutionary algorithms are two methodologies to address a RL problem. Previous researches have been dedicated to polish and optimize a single methodology's implementation for higher performance, but few was doing comparative research between the two. This project is trying to fill in the blank by conducting an empirical comparison on the performance of two algorithms from each methodology in terms of accuracy, time, stability and network structure to learn the strengths and weaknesses of the two. The selected algorithms are Deep Q-learning Network (DQN), NeuroEvolution of Augmenting Topology (NEAT) and Hyper-NEAT. They will be implemented in ALE environment and a self-powered gird world game. The project will provide a detailed analysis on the results of the comparison with the aim to gain an intuitive understanding of the two methodologies by discovering the strengths and weaknesses of the corresponding algorithms and the reason behind.

Contents

List of Figures

List of Tables

Chapter 1

1. Introduction

Ever since the astounding success of Alpha-go triumphing over the human player Lee Sedol at the Go contest in 2016, the world's attention has been drawn to Artificial Intelligence, and for people already working in Machine Learning field, to the sub-branch reinforcement learning. This victory signalled that machine learning was no longer simply about big data classification, but was making progress in the realm of true intelligence. Reinforcement learning introduces the concept of an agent, and addresses the problem of making the most-rewarded decision as a subjective entity in a known or unknown environment. It could be seen as a learning approach sitting in between supervised and unsupervised learning, since it involves labelling inputs, only that the label is sparse and time-delaying [1]. In life, we are not given labels of every possible behaviour in the world, but we learn lessons by exploring strategies on our own, hence RL provides the closest problem setting to the learning process of a human brain, which also explains the excitement that the progress elicits from machine learning scholars.

So far there has been two ways to approach a reinforcement learning problem -- Markov decision process and evolutionary computation. Markov decision process is a mathematical framework that models the world as a set of consecutive states with values, and inside this world there is a rational agent that makes decisions by weighing rewards caused by different actions. If it is an unknown environment, meaning the state values are unknown, the agent may begin by interacting with the world first, observe the consequences, and after a while with enough experiences, it may exploit the knowledge and make the optimal decisions. It builds on top of the rules drawn from observing human intelligence in behavioural psychology experiments, rather than modelling on a grander scale the rules that cultivates human intelligence, which is where evolutionary computation comes from. Evolutionary computation is a family of algorithms that apply the concept of evolution to the computation area as a searching technique to find the fittest solution. More specifically, it utilizes concepts and rules

in the *Theory of Evolution* such as mutation, crossover and fitness, and models the computer to perform natural selection in search of an optimal solution [2]. In other words, it does not attempt to build intelligence from scratch, as long as it renders quasi-intelligent results.

This project conducts an empirical comparison between the two representative algorithms of reinforcement learning and evolutionary computation, aiming to discover their advantage and disadvantage, and the rationales that constitute these characteristics. Researches in the past mainly focuses on the optimization of a particular algorithm, but only few was doing comparison in more than one test cases between the two [3]. This project picks deep Q-learning Network (DQN), NeuroEvolution of Augmenting Topology (NEAT) and Hyper-NEAT algorithms to be the candidate for comparison, and the comparison is operated under self-created game environment and environments supported by ALE.

Figure 1：Arcade Learning Environment

Game environment provides the perfect testbed for solving reinforcement learning problems [4]. One year before the breakthrough of Alpha-go, the DeepMind published a paper on Nature revealing the success of deep Q-learning in achieving human-level performance on 49 Atari games, and one year before that, a group of Ph.D students at the University of Texas also trained Atari game agents only with neuroevolutionary approach and achieved the state-of-the-art performance. Looking further into the past studies, it is obvious that the history of testing algorithms in game environment is almost as old as the reinforcement study itself. Inspired by this observation, this project also tests algorithm performance in game environment that is

supported by ALE that provides dozens of game environments in Atari specifically for improving RL algorithms.

Chapter 2

2. Previous researches

This chapter introduces the related RL works that have been carried out in the game environments.

2.1. Game Environments for Research

2.1.1. Atari games

Atari game refers to the games created for the Atari 2600 game console developed in the late 1980s. It was a very popular gaming hardware that has been sold on market for over a decade. Atari games are a wide range of competitive games involving ball play, shooting, driving etc., which are ideal environments for evaluating AI algorithms. There have been emulators for Atari 2600 such as Stella, but it was Arcade Learning Environment (ALE) published in 2013 that grasped the researchers' interests in Atari games.

ALE is established on top of Stella. Stella emulates the fundamental hardware interactions such as receiving joystick motion inputs and outputting the screen data, on top of which, ALE creates richer interface that allows users to change the frame rate, switch on/off the sound, reset the game, etc. It also accepts commands in a variety of supported programming languages such as C++, python and ruby for environment interactions such as getting the RGB/ grayscale screen, getting the legal action set or minimal action set, getting the accumulated rewards and checking whether the game is terminated, which are customised specifically for AI researchers.

Developers of ALE chooses Atari games to build the RL interface out of many reasons, but essentially, it is because firstly, the games are challenging enough for the current algorithms to work on, but not too complicated that may require certain domain knowledge to function, limiting the possible complexity of games; also, with the screen resolution being 210x160 and

legal action set of size 18, the state and action is simple enough for the algorithms to process [23]; finally, the provided games cover a proper range of reinforcement learning tasks that are qualified to test the "general competency" [22] of the AI technique.

This project uses Atari games as the testing environment for the algorithms and implements codes that function in ALE.

Figure 2: 210 x 160 screen resolution for PITFALL! [22]

Figure 3: Atari controller, including an 8-direction joystick and a fire button

2.1.2. Grid World

Grid World is one of the first environments that were introduced to evaluate the reinforcement learning algorithms. It provides the most direct mapping between a MDP and the description of the environment (Figure.4). It is similar to a 2D RPG game (Figure.5), as the object arrangement/ the screen pixels remains mostly the same while the player moves from tile to tile, meaning that the game state can be roughly defined by the player position on map.

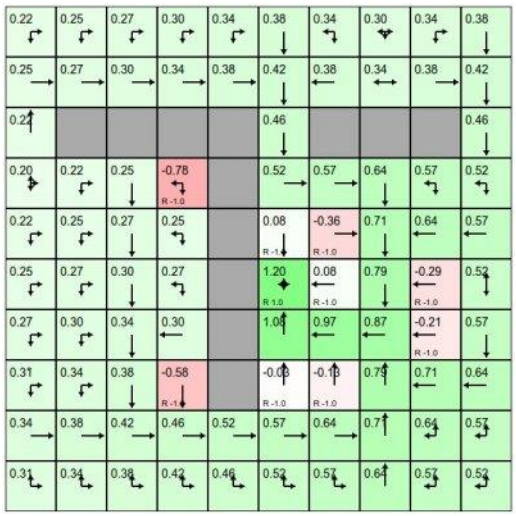


Figure 4: Grid World Experiment

Figure 5: RPG screenshot

The Grid World environment usually contains four objects: agent, goal, wall and the pit. Player should control the agent to move towards goal position while avoiding from falling into pits. Pit falling terminates the game and outputs a negative reward. Wall object blocks the agent's path and send the agent back to where it was if moving into it. Reaching the goal object will terminate the game and receive positive rewards, which is also the wining condition. It is the abstraction of many 2D games, and researchers may arrange the objects to form any problem they would like solve.

This project sets up multiple Grid World environments for observing the performance of both algorithms.

2.2. Previous Study on Reinforcement Learning Task in Game Environment

Initially, researchers created game-like environment for experiment purposes. Algorithms are performed on benchmark tests such as cart-pole balancing, double pole balancing [10] and soccer-bot [3], which are very simple code written for algorithm testing. Later, as console industry matured and simulation devices were made on personal computers, people started to build their AI programs as plug-ins on top of source code of the game. There were researchers observing cooperation strategies in Pacman game [15] (see Figure.6), developing Deep Learning algorithms on Atari 2600 games in Arcade Learning Environment [4], and implementing NEAT on Super Mario with Nintendo simulator (see Figure.7). Algorithms performing more difficult tasks can be examined under sophisticated environments.

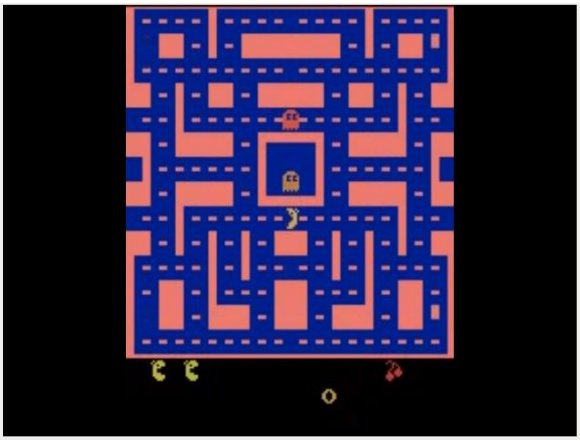


Figure 6: Pacman

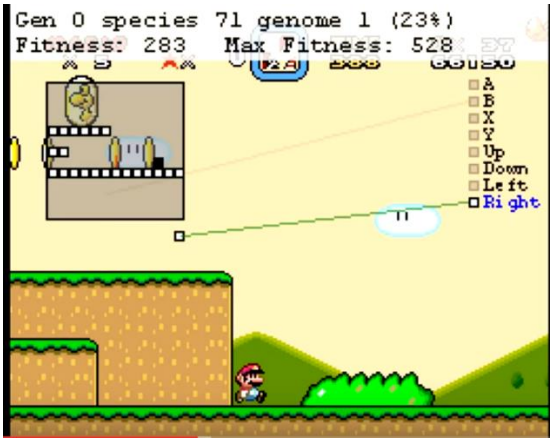


Figure 7: MarI/O

In 2014, neuroevolutionary methods were applied to develop reinforcement learning agents to play Atari games with little domain knowledge. Researchers picked four genetic algorithms, which were Conventional Neuro-evolution with fixed topology, CMA-ES that

evolves fixed-topology neural net more sophistically, NEAT that optimizes both topology and weights, and Hyper-NEAT with indirect encoding based on NEAT, and paired them with three different state representation – object representation, raw pixel representation and noise-screen representation. State representation is implemented via convolutional neural-net, and the output was fed to either a CMA-ES, NEAT or Hyper-NEAT model to train the game for optimal actions (action space: 9 directions and a fire button). Results show that direct encoding methods bring the best performance on object representation as input, while Hyper-NEAT is good at scaling to raw pixel data of higher dimensionality. Results are also compared with previous Q-learning performance, from which authors draw the conclusion that Q-learning is bad at making decisions in large state spaces, while neuroevolutionary methods demonstrate promising state-of-the-art performance that successfully address this challenge.

In 2015, the DeepMind combines Q-learning with deep convolutional and fully connected network to build a reinforcement learning agent that also plays Atari games. It is applied to 49 of the Atari 2600 games and the results are comparable to human players, furthermore, 29 of the results surpass 75% of the human score. The convolutional network is used as value function approximator, which takes in only the high-dimensional raw pixel data (210x160) and score as input, and outputs the optimal Q value. To solve the divergence problem caused by a nonlinear function approximator, a method called experience replay is applied to decorrelate the action sequences.

Table 1: Network Architecture that DeepMind implemented [13]

Layer	**Input**	**Filter size**	**Stride**	**Num filters**	**Activation**	**Output**
Conv1	84x84x4	8x8	4	32	ReLU	20x20x32
Conv2	20x20x32	4x4	2	64	ReLU	9x9x64
Conv3	9x9x64	3x3	1	64	ReLU	7x7x64
Fc4	7x7x64				ReLU	512
Fc5	512				Linear	18

This project will reference some of the techniques from the previous researches and conduct empirical comparison on their performances.

Chapter 3

3. Theory

This section introduces reinforcement learning and evolutionary algorithm, the rationales behind the two methodologies and the selected algorithm of each category. In the end, it touches on the previous studies that involves game environments to be the testing platform for the optimization of each algorithm.

3.1. Reinforcement Learning

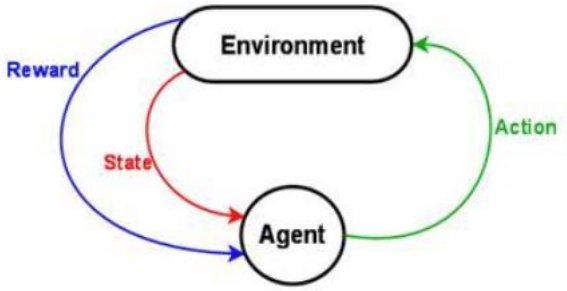


Figure 8: Reinforcement Learning Problem. [1]

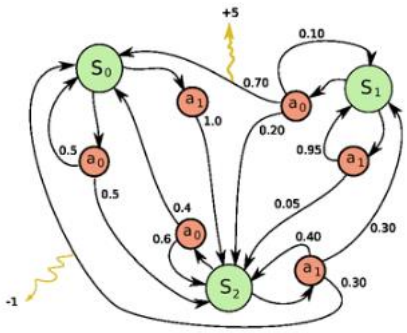


Figure 9: Markovian Decision Process. [1]

It is mentioned above that RL problems introduce the concept of an agent, and correspondingly, there is an entity called environment that receives agent's actions and returns real-valued rewards. Actions are encouraged by being rewarded, and discouraged by being punished, just as the way behavioural psychology views human's learning of behaviour. The Markov Decision Process (MDP) formalizes this psychology problem in computation world under the assumption that the consequence of an action is only related to the last action the agent took, not the preceding ones [1]. This assumption, also known as the Markovian environment, greatly simplifies the problem setting and sets up the boundary for the algorithms, however, finding a solution remains a difficult task. MDP models the decision-making as

sequences of interactions among the agent and the environment. The interaction sequence can be put as a list of state, action, reward. When the agent is in state-1, it takes the action a1, receives reward r1, and it ends up in a new state s2, and the sequence goes on in that order.

$$H = [s_1, a_1, r_1, s_2, a_2, r_2, s_3 \dots]$$

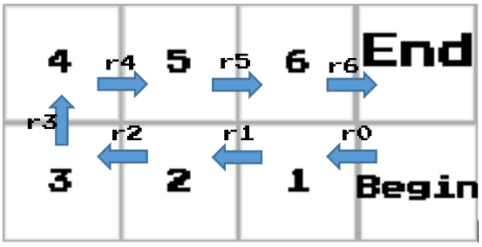


Figure 10: A grid world example

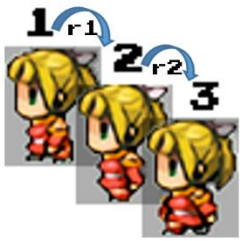


Figure 11: Consecutive states example

Every decision process ends when the termination condition is met. See Figure 4 as an example. Supposed that the agent starts from the initial state s_0, and follows the action policy[1] π all the way to the end-state s_7. The reward it receives at the initial state can be calculated like this:

$$\mathrm{R} = r_0 + r_1 + r_2 + r_3 + r_4 + r_5 + r_6$$

However, the rewards that can't be reached immediately have a chance to disappear or diminish over time, for example the currency exchange rate, therefore, a discount factor γ is used to scale down[2] the future reward.

$$\mathrm{R} = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \gamma^4 r_4 + \gamma^5 r_5 + \gamma^6 r_6 = \sum_{k=0}^{6} \gamma^k r_k$$

The same rule applies to the continuous environment, where state successively changes in the same region at each frame (see Figure 5). In the Markovian environment, a state value is used to keep track of how much future rewards any action policy may incur beginning from this state, for example, the value of state(end) is 0, meaning no action is allowed in state(end) other than ending the sequence and receiving a reward 0. State(6) is of value $v(6) = \mathrm{E}[r_6 + v(6)]$, since the only action allowed in state(6) is "moving north", and the action is deterministically leading to state(end). However, if the action is stochastic and actions towards four directions are allowed, $v(6)$ value will be the expected return of all the adjacent tiles:

[1] An action policy is a function maps a state into an action, just as what the arrow bridging between states is doing.

[2] Technically, γ can be 1, which generates far-sighted strategy.

$$v(6) = \sum_{a\epsilon(north,south,east,west,)} \pi(a|6)(R_s^a + \gamma \sum_{s'\epsilon(6,1,end,5)} v_k(s')).$$

Q value or action value represents the rewards incurred by an action in a given state. Moving east in state(6) will receive $q(6, \text{north}) = r_6 + \gamma v(end)$, while moving in any direction will receive $q(6, a) = r_6^a + \gamma \sum_{s'\epsilon(6,1,5,end)} P_{6s'}^a v(s')$. Essentially, state value is the expected return of all the possible q-values in the given state, and q-value is the expected return of all the state values the action leads to. The system may calculate a table of values for the grid world example through value iteration.

Table 2: A table of state values for Figure 2

-3	-2	-1	0
-4	-5	-6	-7

Table 3: A table of q values for Figure 2

-1 -1 -1 -1	-1 -1 -1 -1	-1 -1 -1 -1	Goal
-1 -1 -1 -1	-1 -1 -1 -1	-1 -1 -1 -1	Pit

3.1.1. Deep Q-Learning Network (DQN)

A table can be used to store all the relevant state values and action values (see table 1 and 2), however, sometimes the state space is too large to do value storage in tabular form, for example, the pre-processing in DeepMind paper takes an input of size 84 x 84 x 4, meaning there will be about 10^{67970} states in total, which is more than the number of atoms in the universe [1]. In cases like this, a value function approximator is used to represent values. Technically, the approximator can be any function that properly estimates the value, however,

since neural network usually provides a better fit to the data, ANN is most researchers' first choice to make an approximator. In fact, the DeepMind has implemented a 5-layer Convolutional Neural Network for function approximation in the 2015 research. A Neural-net can be used to represent state value, action value or the action policy (see Figure 12). Researchers usually go with the third one, since it requires less iteration to generate the best policy.

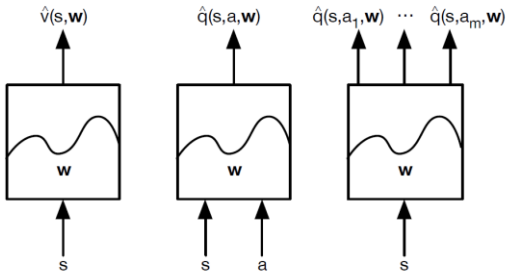


Figure 12: Function approximator [19]

Therefore, knowing the state transition probability and the discount factor, one should be able to calculate values for all the states and actions through policy evaluation and improvement, and function approximation. However, in most RL problems, the model is rarely informed at the beginning, so we expect the agent to explore the environment randomly on its own and observe consequences, and do some proper exploitation at the same time. ε-greedy algorithm combines the demand for both exploration and exploitation, where ε stands for the rate to perform greedily, 1-ε of rate to act randomly. After taking the action A, we observe a reward R, and update $\mathrm{Q}(\mathrm{S},\mathrm{A})$ by following this equation:

$$\mathrm{Q}(\mathrm{S},\mathrm{A}) = Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A)$$

Here, α is the learning rate that determines how much of the difference should be considered. If α is 1, the equation will be exactly the same as a Bellman equation[3]. This process is called SARSA (S, A, R, S', A'). It updates the initially randomized value function towards the true *estimation* $R_{t+1} + max_{s'}\mathrm{V}(s_{t+1})$, and updates q function towards $R_{t+1} + max_{s'}\mathrm{Q}(s_{t+1},\mathrm{a}')$. Hence the Q-update begins from a very inaccurate set of values, but the idea is, for enough iterations of updates, the values for states and actions are heading towards the right direction, and will eventually converge to the optimal values (see Figure 13 and 14 for intuition). This approach is named Deep Q-learning network, because it involves deep neural network to

[3] Bellman equation: $\mathrm{Q}(\mathrm{S},\mathrm{A}) = R + \gamma Q(S',A')$

approximate the value function, and it utilizes Temporal Difference (TD) method on Q value learning.

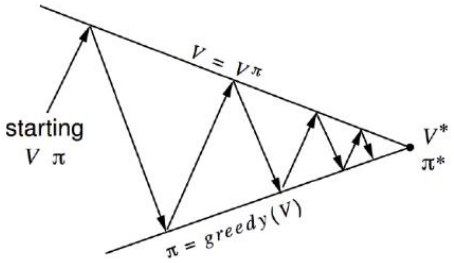


Figure 13: Convergence [19]

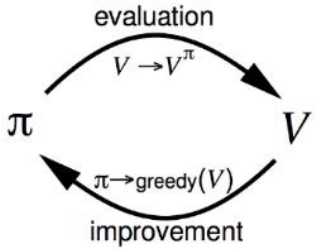


Figure 14: Convergence [19]

3.2. Genetic Algorithms

Genetic algorithm (GA) is a sub-area of evolutionary computation. There are also evolutionary programming, evolutionary strategies and genetic programming [5] paralleled to GA that basically express the same rationality with some structural differences [6]. As stated in the introduction, evolutionary computation applies the principal idea in the *Theory of Evolution*, and creates a prototype of natural selection procedures on solutions labelled by fitness value. The procedure is illustrated in Figure 15, which involves population initialization, fitness function calculation, crossover and mutation, survivor selection and termination [7].

The first population is randomized in terms of value and/or, if allowed, structure. A fitness function is then applied to each chromosome, meaning a member of the population (see Figure 17 for more terminology), to calculate their fitness values. The fittest chromosome will receive higher chance of crossover. From the second generation and further, each chromosome has a defined possibility to mutate. Finally, the system compares the fitness value of the latest generation, and determines whether the termination condition is met. If met, the fittest chromosome will be returned, and the system terminates; otherwise, the loop continues, and at the last generation specified, NEAT returns the fittest chromosome as the final solution.

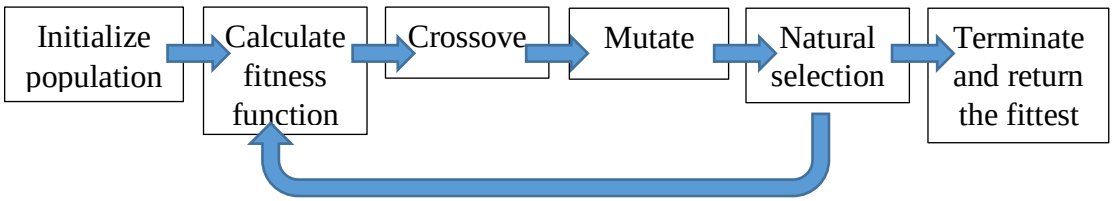


Figure 15: Natural selection procedures [7]

For the prototype to work, the system needs a fitness function and an encoding scheme. An encoding scheme transforms a solution from real-world representation, a.k.a phenotype, into its abstract form, or genotype, that accepts crossover and mutation. In some cases, a phenotype space and a genotype space can be exactly the same, for example, when in search of a 41-character string "to be or not to be, that is the question." [8], the representation and the encoded form are both made up of ASCII characters. However, in a more complicated setting, the phenotypical solution can be a behaviour, while its genotypical form can be a neural network.

Fitness function punishes the bad solution by rating lower fitness value, and rewards the better ones with higher fitness value, which seems similar to the concept of "reward" in RL. The implementation of a fitness function is task-specific, but the goal is always to push the evolution closer to the optimal solution. In the "to be or not to be" example, to calculate the fitness of the string "ifs djdfh? erglop ", the fitness function may return 1 for each correct letter, and 0 for the incorrect ones, or it may return the difference of positions each letter is in an ordered 26-alphabet sequence. If to decide the fitness value of solutions that plays video games, the function may consider the score of the play, the time duration of agent that remains alive etc. Sometimes different designs can achieve the same thing.

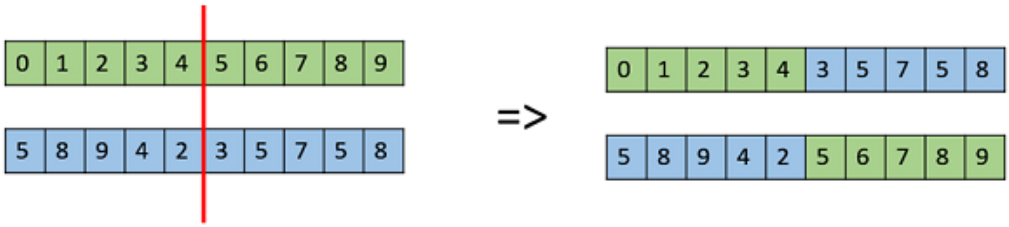


Figure 16: One point crossover example [7]

Crossover is an effective technique to produce new generations with higher fitness, since fitter parents are rewarded with higher opportunity to reproduce. This round of selection can be implemented with Roulette Wheel Selection, Stochastic Universal Sampling, Tournament Selection or Rank Selection [7]. The crossover process should generate children with combined features from both parents, however, the implementation details are specific to the encoding scheme. See Figure 10 for a crossover example. Mutation plays a key role in preventing the population from getting stuck in the dead end. When the mutation rate is zero, at some point of the evolution, the chromosomes will remain forever to be the closest thing the initial population can get, but they may still be far from the optimal solution. However, with a mutation rate of 1, the search becomes completely random, and the evolution is pointless. Choosing a mutation

rate with caution is suggested. Mutation is similar to ε in greedy algorithm, as they both control emphasis on either exploration or exploitation.

It is observed that when a variety of solutions are competing against each other, the evolution will terminate within fewer populations [9]. Solutions are then speciated to encourage inter-species competition. However, if not implement properly, new species may get eliminated too early over the evolution due to late-maturing. Some genetic algorithms addressed this problem by introducing protection mechanism to keep that from happening.

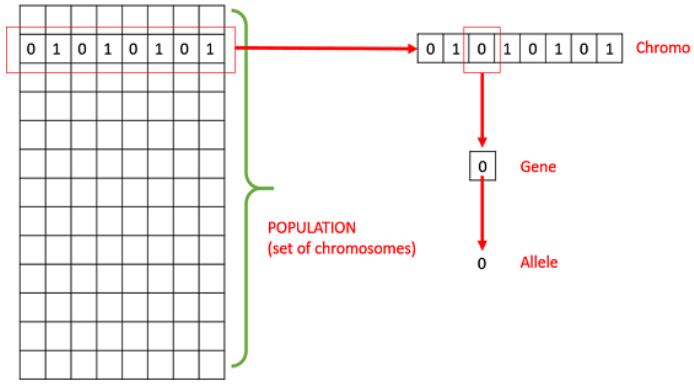


Figure 17: Terminology [7]

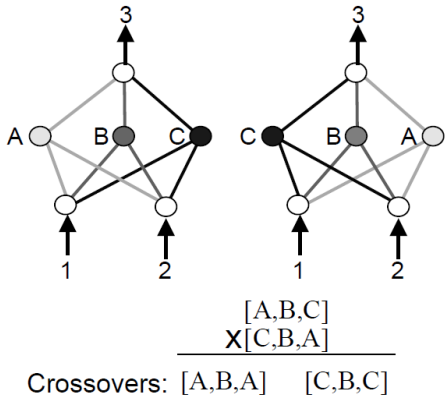


Figure 18: conventions [11]

3.2.1. NEAT

The evolution of neural network (neural evolution) is a sub-problem of genetic algorithm. When the genotype is a neural network, an efficient encoding scheme that allows both the weight and the topology to crossover and mutate needs to be developed. Researchers have tried many alternatives: binary encoding, graph encoding, non-mating and cellular encoding [10]. However, some of them sacrifice efficiency for simplicity, some attempt low-level abstraction without yielding good results, and some give up on crossover altogether since it may lead to the loss of function [11]. Graph encoding was also faced with the problem of competing conventions – it does not recognize the same neural network with trivial differences (see Figure 12). In addition, since most algorithms randomize the initial population, the complexity of search space usually grows out of control over generations. These issues combined with the innovation protection problem constitute the predicament neuroEvolution was in, however, NeuroEvolution of Augmenting Topologies (NEAT) algorithm was created to successfully address these problems, and achieve efficient evolution of weight and topology.

NEAT initializes its population uniformly without hidden layer, meaning all the input nodes are pointing directly to the output node. Then through point mutation, link mutation and

node mutation, the network starts to grow incrementally. Point mutation changes the weight, link mutation adds new connection without introducing new nodes, and node mutation adds new node and connection while disabling old connection (see Figure 19). This design drastically minimizes the dimensionality of search space, and improves the overall computation efficiency. In terms of crossover, the algorithm introduces a pivotal device – historical marking – to keep track of the structure's chronology and produce clean and effective child topology. Each connection of the network, defined by its input-output pair, is given an innovation number to signify its birth sequence. Innovation number is implemented as a global counter, which gets incremented each time a new connection is created. This design effectively eliminates the problem of competing convention by integrating chronology as part of the structure. Other than that, with historical markings, the parent genes can line up against each other, and by simply comparing the fitness of connections, the system can perform crossover directly without having to carry out complicated analysis on the topology (see Figure 20) [12].

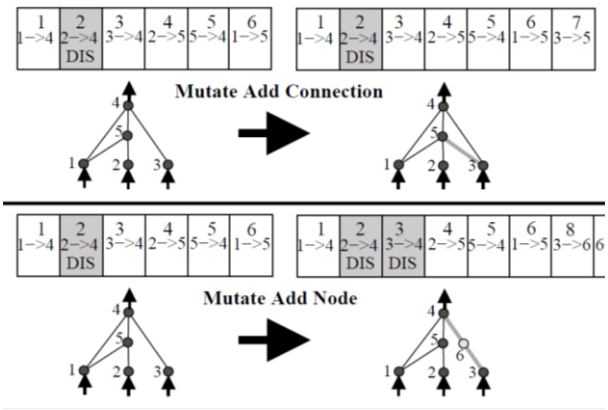


Figure 19: Mutation [11]

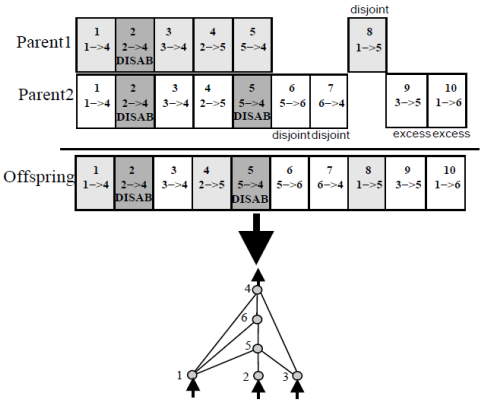


Figure 20: Crossover [11]

With historical marking, speciation is also made possible. The gaps left in between the innovation number line-up is called disjoint or excess (see Figure 14), they are effective tools to measure structural differences. By comparing the number of disjoint D and excess E to a compatibility threshold δ_t, the system is able to speciate the chromosomes.

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

$\overline{W}$ is the average weight differences of the matching genes. Coefficients c_1, $c_2$and c_3 are used to adjust the importance of the three factors [12]. Each new chromosome will be compared to a random representative from the existing species sequentially, and will be assigned to the first

species it matches. This will prevent species overlap. Speciation allows innovation, which in return produces fewer rounds of iteration.

3.2.2. Hyper-NEAT

This project considers Hyper-NEAT to be a participant algorithm for the comparison, for it may replace NEAT when it fails. NEAT lacks the ability to exploit the geometry of the environment and to encode regularity and modularity. It evolves generations of artificial neural networks to address the problem, but neural network alone is oblivious to the geometric information of the input nodes, meaning that it could take a lot more generations and a lot bigger population sizes to learn a function of regularity and modularity between two geometric positions. Due to time limits, it is unrealistic for this project to run all the generations that NEAT requires until achieving proper progress, therefore Hyper-NEAT should step in to accelerate the evolution process. This inefficiency of NEAT is largely due to the direct encoding scheme it has, which is a one-to-one cardinality mapping between genotype and phenotype. Whereas, learning from how human brain engineered trillions of inter-connection parts with only 30 thousand of genes [20], a developmental encoding scheme is created to reproduce this indirect mapping between the encoding components and the output structure, where the outputs can achieve astronomically more complex variation than its components combined.

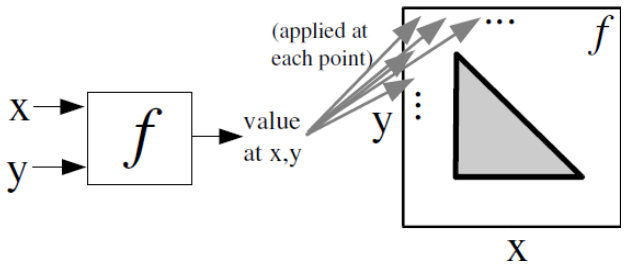


Figure 21: Compositional Pattern Producing Networks

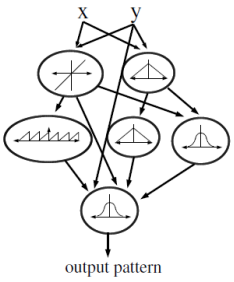


Figure 22: CPPN structure

Hyper-NEAT evolves **C**ompositional **P**attern **P**roducing **N**etworks (CPPN) with NEAT for such encoding. CPPN takes in x, y or x, y, z values that specify a position in a 2d/ 3d space, and output the corresponding intensity at that coordinate (Figure.21). The network aggregates inputs not through summation, but a series of basic functions such as linear, sin, sigmoid, Gaussian to represent identity, symmetry or periodic motif (Figure.22). It is originally used to evolve regular patterns or structures in a 2d/ 3d space, but in Hyper-NEAT algorithm, CPPN

takes in the position of two substrate nodes, and its output determine the weight that connects both nodes (Figure.23).

Hyper-NEAT first queries each potential connection between the input and output substrates, then feed the coordinates of two substrate nodes to CPPN, CPPN outputs the weight of connection and helps ANN with defining all of its weights iteratively. The idea is that by feeding geometric information, CPPN possesses the sensitivity to geometry, or in other words, the ability to see. It flexibly adjusts the light-weighted CPPN structure according to new rounds of fitness values and is able to solve the task in fewer generations.

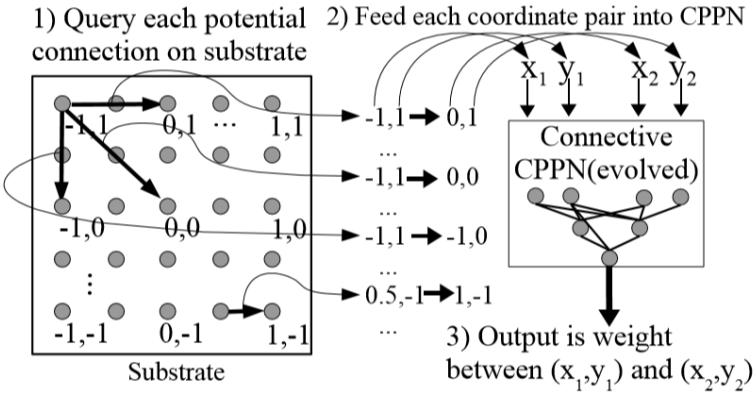


Figure 23: Hyper-NEAT work flow [20]

Take this project as an example to see how ANN weights works for different substrate node values. In the Grid World experiment, Hyper-NEAT only looks at the four nodes corresponding to the four legal actions. Figure 24 and 25 display that the same set of weights should response to different object arrangements on the substrate.

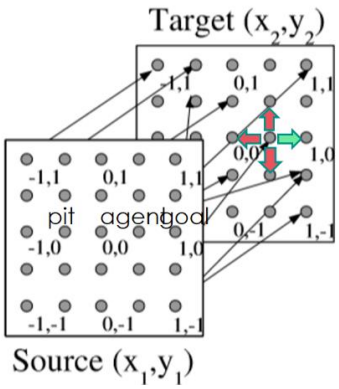


Figure 24: Substrate Computation-1 [20]

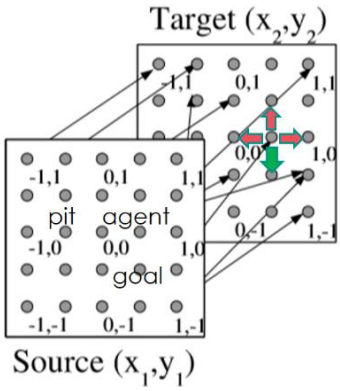


Figure 25: Substrate Computation-2

3.3. Artificial Neural Networks

ANN is a computation model based on neural connections in human brain. Each node of the network imitates the working of a neuron. A real neuron has dendrite that receives electrical signals from neighboured neurons, and axon terminals that send out the signals (see

Figure 26-27). The connection strength between neurons determines the signal transmission strength. An artificial neuron is a computation unit representing the computational manipulation of input values. It implements the same rationale in neural biology, with each node being given an activation function that activates computation process if the combined inputs are strong enough, or the unit outputs nothing. A large number of artificial neurons are collected for complicated function approximation, hence the name "neural network".

Figure 26: Arts of Neurons in Brain-1

Figure 27: Arts of Neurons in Brain-2

Multi-layered perceptron (MLP) is a fundamental model of neural network, which consists of an input layer, a hidden layer and an output layer. A vector of input nodes represents an array of input data, or "features" in machine learning context. These nodes are connected to the second layer, usually a hidden layer, through connections or links. The number of connections can be configured freely, although the most common model is always presented to be fully-connected, since the absence of connection is essentially a link of zero weight. Weight is to indicate the connectivity strength of the link. By multiplying weights with input node values, the inputs are weighted according to the connectivity strengths, and they are aggregated to be the values for the hidden layer nodes. By setting up multiple hidden layers, the computation complexity of the network can be increased.

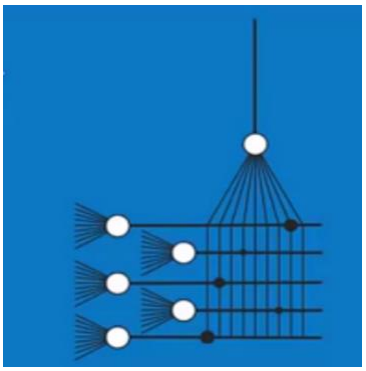


Figure 28: Abstraction of neural connections and strength-1 [25]

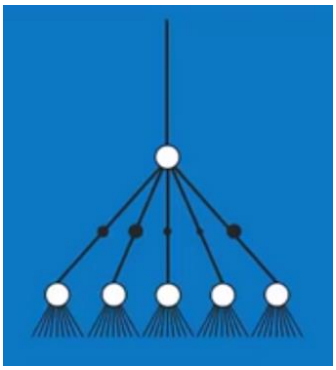


Figure 29: Abstraction of neural connections and strength-2 [25]

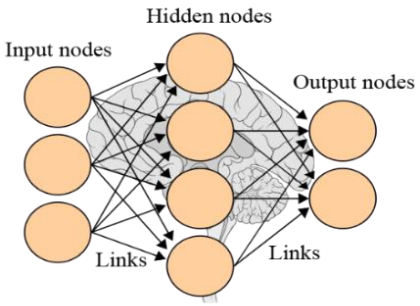


Figure 30: Neural network

By defining the input features, neural network in nature can estimate any functions. This attribute has contributed to its wild popularity in machine learning application. Fields such as voice recognition, face recognition, natural language processing are all making breakthroughs with the help of neural networks [26].

Back-propagation

In the early days of neural network applications, input features were hand-engineered and network weights were untrainable [25]. By introducing back propagation that is essentially gradient descent combined with chain rule, the network can compute the gradients and correct errors all on its own.

Gradient descent

Gradient descent is the process of getting the first order derivative of the loss function and adjusting the weight value accordingly. By looking at the sign of the derivative, the machine is able to determine whether to increase or decrease the weights to minimize the error between target value and actual value. After several iterations, the error value may be deducted to zero, and the weights converge to the optimal set. Hence, the network can predict the value accurately.

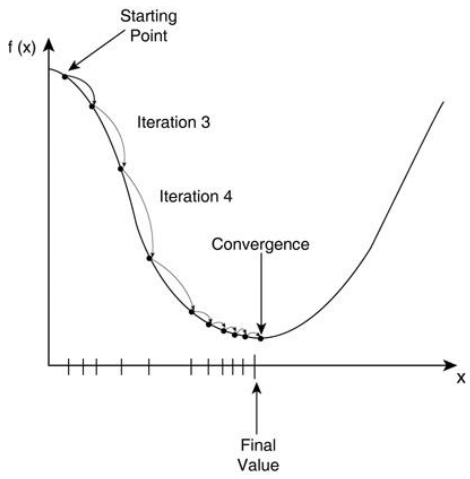


Figure 31: Gradient descent

Chain rule

However, neural network has several layers of nodes and weights, and error function only determines the weights that are directly connected to the output layer. To be able to adjust all the weights of the network, chain rule is applied. It effectively utilises the knowledge that the next layer vector is a dot product of weights and previous vector, in this way, vectors from

all layers are chained, and by applying chain rule, one is able to adjust any set of weights through the known node values and the error.

$$\frac{\partial \text{err}}{\partial weight} = \frac{\partial a}{\partial weight} \times \frac{\partial b}{\partial a} \times \frac{\partial c}{\partial b} \times \ldots \times \frac{\partial y}{\partial x} \times \frac{\partial z}{\partial y} \times \frac{\partial err}{\partial z}$$

3.3.1. Convolutional neural networks

CNN is equipped with filters that imitate the mechanism of neurons on retina to help with recognizing specific patterns. Filters are 2d or 3d volumes of nodes with trainable weights. CNN accepts input as matrices rather than vectors, and performs dot production between the filter weights and the input (See Figure.73 for illustration). The convolution layer is followed by a max pooling layer that further compresses the results. Essentially, CNN is neural network with several layers of filter and max pooling up front, which enables the network to deal with matrices of data and train the weights for pattern recognition capacity. This technique is widely utilised by facial recognition, voice recognition tasks (Figure.32).

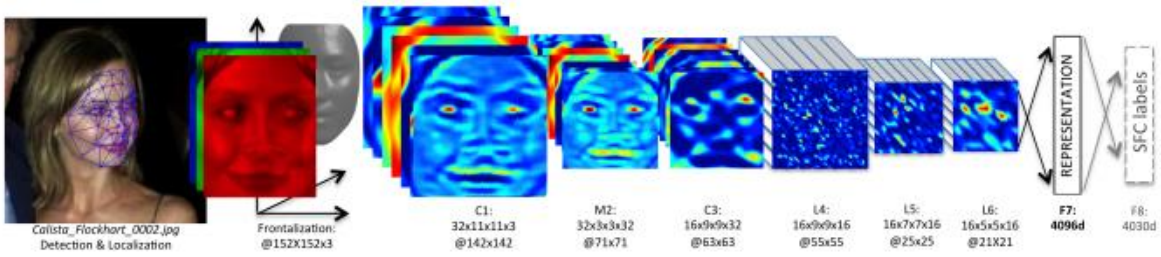


Figure 32: Convolutional neural network for face recognition

Chapter 4

4. Methods

This project runs the algorithms in two distinct environments – Atari and Grid World. They are to compare the processing efficiency of high and low dimensional data. The input for Atari environment is a vector of 84 x 84 x 4 (1E5) dimensions, while for grid world, it is only of size below 1E2. Different from Atari, Grid World is an environment created specifically for this project, therefore its environmental parameters can be adjusted, making it convenient to carry out a control experiment with much richer comparative items than ALE provides.

4.1. Atari games

The simulated Atari games can be categorized as round-based and action-based rewarding systems. Sports games like Pong and Tennis issue score at the end of each round, these scores are then summed up to determine whether the player wins or loses. In other words, a complete sequence of actions in each round is evaluated for a single feedback. The player may keep playing against opponent for a few turns before the round is finished. For action-based games, players receive rewards after each action. Actions lead to consequences in a few frames ahead, and rewards are issued accordingly. For example, for the game Breakout (Figure.34.), by moving the plate below the ball to support it at different angles, the player determines which bricks to be hit by the ball, and each of the crushed bricks yields immediate rewards. Round-based rewarding is sparse while the action-based rewarding is compact.

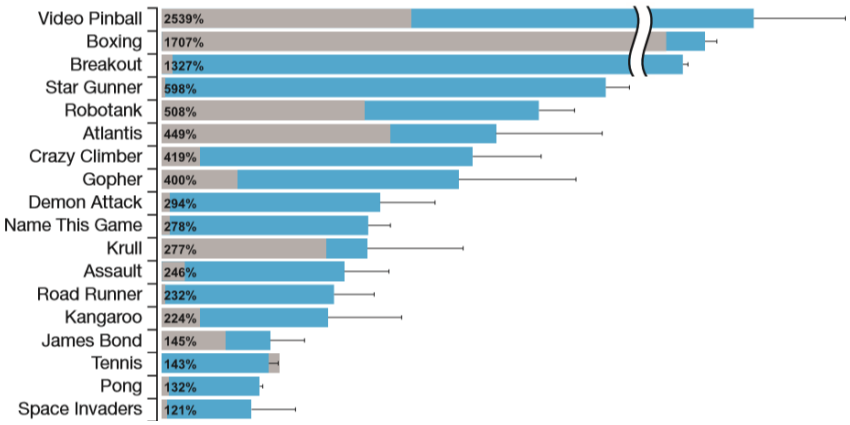


Figure 33: DQN performance [13]

By observing the DQN performances implemented in [13] (Figure.33), the top 10 scores from all the gameplays are from the action-based rewarding games, only at the 16th place appears the sports games that adopt round-based rewarding. In order to observe whether the two rewarding systems influence the algorithms' performance, or more specifically, whether evolutionary algorithms present a different preference, this project chooses one for each category, Breakout and Pong, to be the testing environment (Figure.34 and 35).

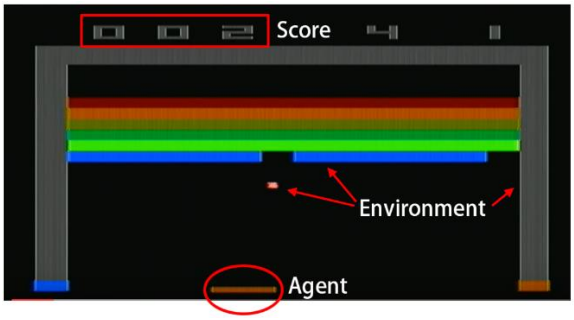


Figure 34: Breakout

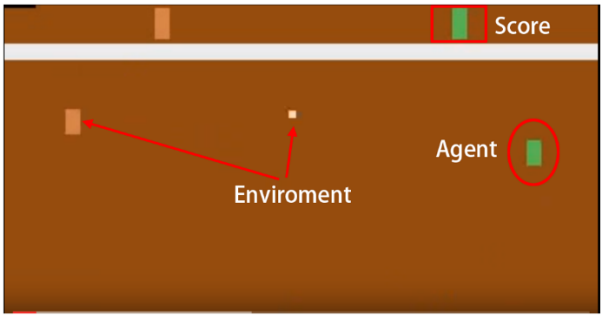


Figure 35: Pong

In addition, they are also to test training efficiency on raw pixel data. Input screen will be the downsized from 210 x 160 RGB to 84 x 84 grayscale. This project is interested in whether the evolved neural network is able to process this amount of data efficiently within the number of generations specified, which is the same to the generation size for evolution. Performances are compared in terms of time and accuracy.

4.2. Grid world games

Atari games are good for inter-algorithm comparison, however, since it's a commercialized game with fixed source code, it's difficult to twig the environment parameters for comparative observation. To enrich the compared items, this project implemented its own version of a grid world game and designed 27 maps that either differs on map size or map pattern/ difficulty. The 27 maps are to disintegrate the whole learning/evolving process into 27 small tasks, and by observing how individual solutions are formed, this project can evaluate the capacity of both algorithms at solving a RL problem.

On these maps, Hyper-NEAT and DQN is tested. At the beginning of each episode, agent is instantiated at random starting location other than wall, goal and pit location. Having been trained by algorithms, the model should move the agent towards goal with the least number of steps possible.

White arrow is goal, stepping on goal will return +10 reward and terminate the game. The agent has four available actions: go up, down, left and right. Moving into wall returns -1 reward and the agent remains at the old position. Each move yields -1 reward to encourage goal reaching with the least number of steps. Stepping on pit will receive -10 reward and episode termination.

4.2.1. Map Design

For maps of size 4x4 to 6x6 (Figure.36-40), each training/ evolution process is set to terminate when test accuracy reaches 100%. Number of testing episode is maximised at map width * map height + 10, larger than the total grid number on map, meaning that at the end of each epoch of training, the testing procedure should have placed the agent on all the possible locations of the map, guaranteed that the testing statistics is accurate.

For pattern #1 and #2, nothing is blocking the way from any location to the goal, which makes it the easiest maps that estimates a linear function. For pattern #3, about 2/3 of the locations need to make one turn to reach goal, and some of the trips may end up in the pit. For pattern #4 and #5, if randomised at location (0,0), the agent should make more than 3 turns to reach the goal, which looks for the most complicated function of all. To finish within 10 steps, it takes 5 turns on pattern #5 to win. The same layout is repeated on size 5x5 and 6x6 to test if map size has any impact on model training.

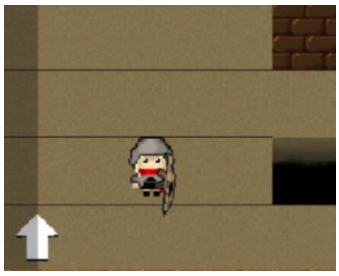


Figure 36: Map pattern 1

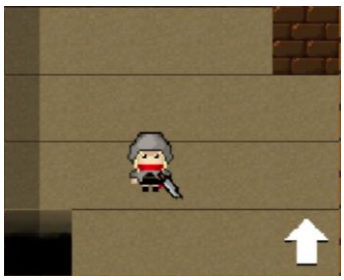


Figure 37: Map pattern 2

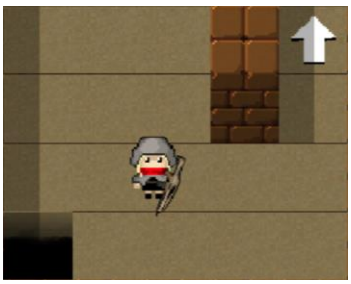


Figure 38: Map pattern 3

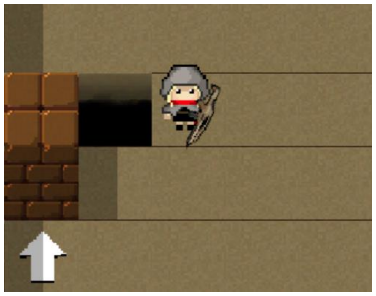


Figure 39: Map pattern 4

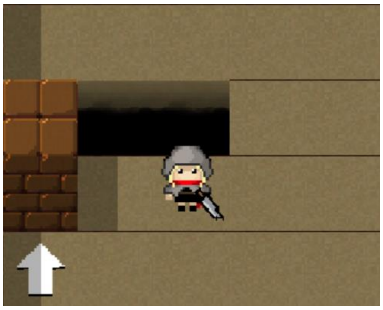


Figure 40: Map pattern 5

For map of size 7x7 to 9x9, much complicated patterns are drawn (Figure.41-43) – a concentric circle map, a stripe map and a vortex map. Due to time limit, models are trained within 50 epochs * 100 steps per epoch, or 50 generations * 100 population size, hence they are not expected to win 100% in all of them. However, 50 epochs are enough to observe the overall trend of the progression, and it's enough to complete the comparison. These designs are to observe how the map patterns and sizes influence the training process, what algorithm is good at which patterns, and why is that. The same layout is also drawn on maps of all three sizes.

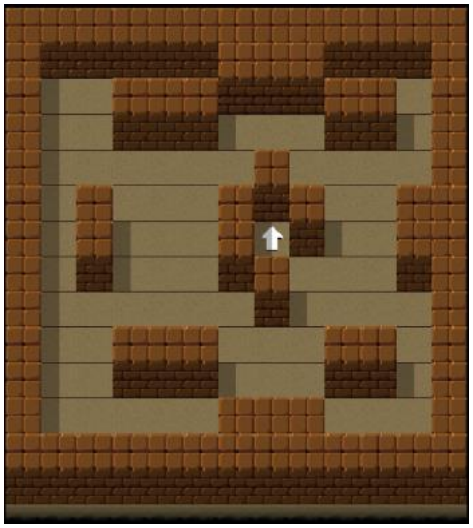


Figure 41: Map pattern 6

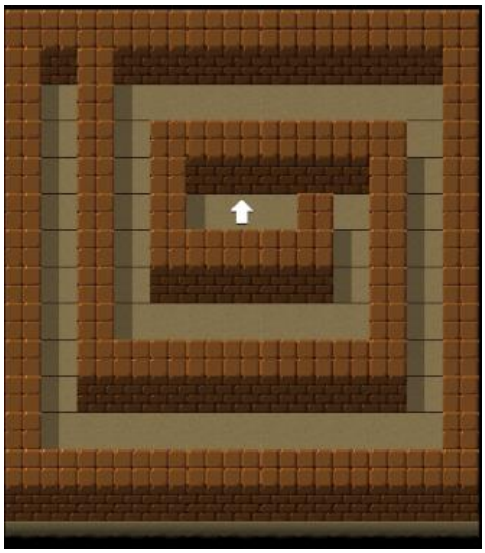


Figure 42: Map pattern 7

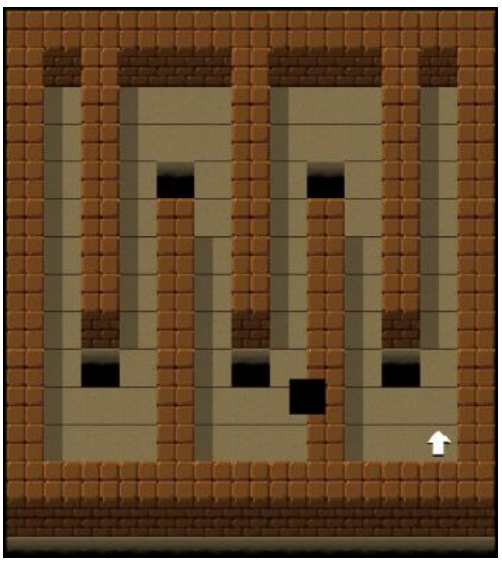


Figure 43: Map pattern 8

4.2.2. Comparison Framework

Performance evaluation is a comprehensive assessment of 4 attributes: accuracy, time, stability and network structure.

Accuracy

Accuracy is determined by the algorithm's ability to correctly find the goal position within the step threshold specified. Considered that the agent will be randomised at the beginning of each test episode, this project uses the win rate to define the accuracy, for example, 10 wins out of 10 tests is deemed as 100% accuracy. Therefore, two factors will be involved in accurately reflecting the algorithm's path finding capacity – test step threshold and test episode number. Test step threshold is mainly to make sure that the trained network will find the goal within minimal possible steps, and also to prevent overfitting from the training process. Test episode number is to guarantee that the solution is able to address the task from all the locations available on map. For example, if the map is of size 10x10 with 78 valid positions (the unoccupied positions that can be accessed by other unoccupied positions), the test episode number should at least be not smaller than 78, so that a 100% accuracy is of higher probability covering all the action sequences originated from all the valid positions. In this project, test number follows the equation:

$$\text{test number} = \text{map height} * \text{map width} - (\text{map height} + \text{map width})$$
$$\text{If test number} < 15, \text{test number} = 15$$

Due to time limits, it is infeasible for this project to go with a number greatly larger than height * width, the equation above strikes a proper balance between time and probability after experimenting with some of the other heuristics. Since map patterns vary greatly, step threshold is defined pattern-wise. Detailed specification is available at table 4.

Table 4: Step threshold specified for each map pattern

Pattern number	*Step Threshold*
1, 2, 3, 4, 5, 8	height + 2 * width
6	3 * height + ceil(width/2)
7	height * ceil(width/2) + ceil(width/2)

Time

Time attribute is to measure the amount of time required to finish epoch number of training or the evolution of generations of networks. Time combined with accuracy attribute determines the algorithm's efficiency.

$$\text{efficiency} = \frac{test\ accuracy}{time}$$

Stability

Stability measures the time and accuracy variance and the learning curve for each map.

Each of the algorithms in nature develops the solution with certain level of randomness – Hyper-NEAT initializes the first generation of networks randomly, and the evolution process is equipped with 0.1% chance of mutation, mutation also occurs in random weight and connection of the network; In terms of DQN, it has ε chance of making a random move and $\frac{1}{legal\ action\ number}$ rate to make either one of legal actions. Stability attribute is to measure to what extend is each algorithm's time and accuracy affected by this randomness, or its resistance to randomness. 8 out of 27 of the maps are trained multiple times to examine the algorithm's resistance to randomness.

Learning curve is to look at how learning progresses in an individual map. By observing whether the accuracy develops stably, this project is able to tell how efficiently and effectively each algorithm establishes new action policy based on its previous knowledge.

Network structure

Network structure looks at the number of layers, nodes and connections for each algorithm's solution. This attribute is to discover which algorithm provides a solution of simpler structure, and whether the structure can be diversified in nature.

Apart from conducting a horizontal comparison among the attributes listed above, results are also analysed vertically to examine each algorithm's reaction to different map sizes and map difficulties. This project is interested in examining whether accuracy, time, stability and network structure varies as the map size or pattern difficulty grows. Horizontal comparisons in further depth are established on top of the vertical comparison results. Full analysis is available in the next section.

Comparison framework is listed as follow:

- Inter-algorithm comparison
    - Efficiency (accuracy/time)
    - Stability
    - Network structure
- Intra-algorithm comparison
    - Different map size

- Efficiency (accuracy/time)
- Network structure

- Different map difficulty/ pattern
  - Efficiency (accuracy/time)
  - Network structure

4.3. Implementation

This section introduces the implementation of codes. Table 5 shows the main logic of each algorithm. Program structure, class details, parameter settings and other necessary information are also illustrated.

Table 5: Main logic for both algorithms

DQN:	Hyper-NEAT/NEAT:
For epoch number **e**: While buffer not full: Make an action (random) Append to the buffer (State, action, reward, new state) For training steps **s**: Make an action (random or greedy) Insert to the buffer (State, action, reward, new state) Train the network on minibatch sampled from the buffer	For each generation **g**: Crossover and mutate (After the first generation) Speciation (After the first generation) For each population **p**: Initialize (first generation only) While not exceed step threshold **t** or episode not terminated (testing): Step the gene for prediction Give each solution a fitness value according to the test results
While not exceed step threshold **t** or episode not terminated (testing): Step the network for prediction Return model	Query the fittest model and set the ANN weights accordingly Return the ANN

Grey rows indicate CPPN evolution, which also describes the implementation of NEAT algorithm since CPPN is evolved by NEAT. For the statistics to be comparable, this project sets the first loop number e and g, as well as the second loop parameter s and p to be equal, meaning that evolutionary algorithm's generation size is equal to DQN epoch, EA population size is equal to DQN training steps.

4.3.1. DQN for Atari

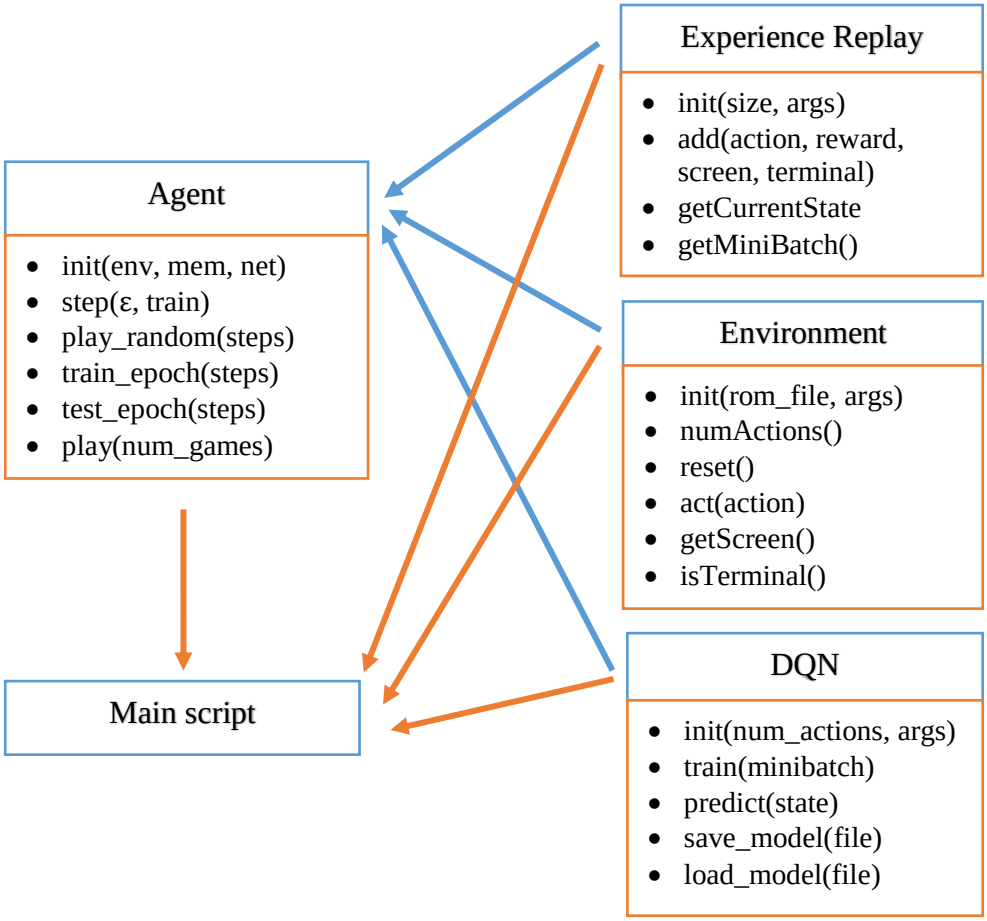


Figure 44: DQN Classes implemented for Atari environment [1]

Environment

Environment class is a wrapper class of the ALE python interface, further simplifying the query and ALE interactions. It directly interacts with the agent class, accepting an action value and returning the corresponding rewards through act() function, with customisation that suits the testing and training purposes. Environment information such as screen data, whether the game terminates and legal action set can be queried through this class. It also accepts bash command for initialization.

Experience Replay

Experience replay class is for the replay memory management. It specifies the size of the replay memory/ buffer and minibatch, stores the "state, action, reward, new state" tuple to the replay memory, and samples a minibatch from the buffer. It interacts with the agent class by receiving tuple and sending out the minibatch.

Deep Q Network

The low-level implementation of networks is coded in the DQN class. Using Neon library, the network can be executed in both CPU and GPU environments. Initialization of the network requires the setting of environment parameters such as number of legal actions, screen dimensions, exploration rate, gamma value and clip error. It implements the training and prediction process, and may store the network structure through save_model().

Agent

Agent class is where integrates the other three components and carries out the steps specified in Table 5. It makes a move, receives corresponding feedback (rewards and game termination information) from the environment object, outputs the SARSA tuple to the experience replay class, receives minibatch, sends the minibatch to the DQN class for model training and cycles the loop for epoch number of iterations. Agent class initialization requires an environment, experience replay and DQN instance.

Main

Although agent class has already implemented the main logic for the task, there is an additional main script for interfacing the bash commands and the component classes. Bash commands are to specify task and environment parameters such as frame skip, learning rate, exploration rate, replay buffer size, train steps etc. The task will go with default if not specified.

Parameter setting

The deep Q neural network is equipped with three convolutional layers and two fully connected layers. Max pool layers are left out to maintain geometric sensitivity. Network implementation follows Table 3. See table 6 for parameter setting and appendix for full configuration details.

Table 6: DQN parameter setting for Atari

Epoch number		**Train steps**		**Discount rate**		**Exploration rate**	
Atari	GW	Atari	GW	Atari	GW	Atari	GW
200	50-100	250,000	100-200	0.99	0.975	1-0.1	1-0.1
Buffer size		**Decay rate**		**Learning rate**		**Minibatch size**	
Atari	GW	Atari	GW	Atari	GW	Atari	GW
1,000,000	180	0.95	$\frac{1}{epoch}$	0.00025	1	32	140

4.3.2. Hyper-NEAT/ NEAT

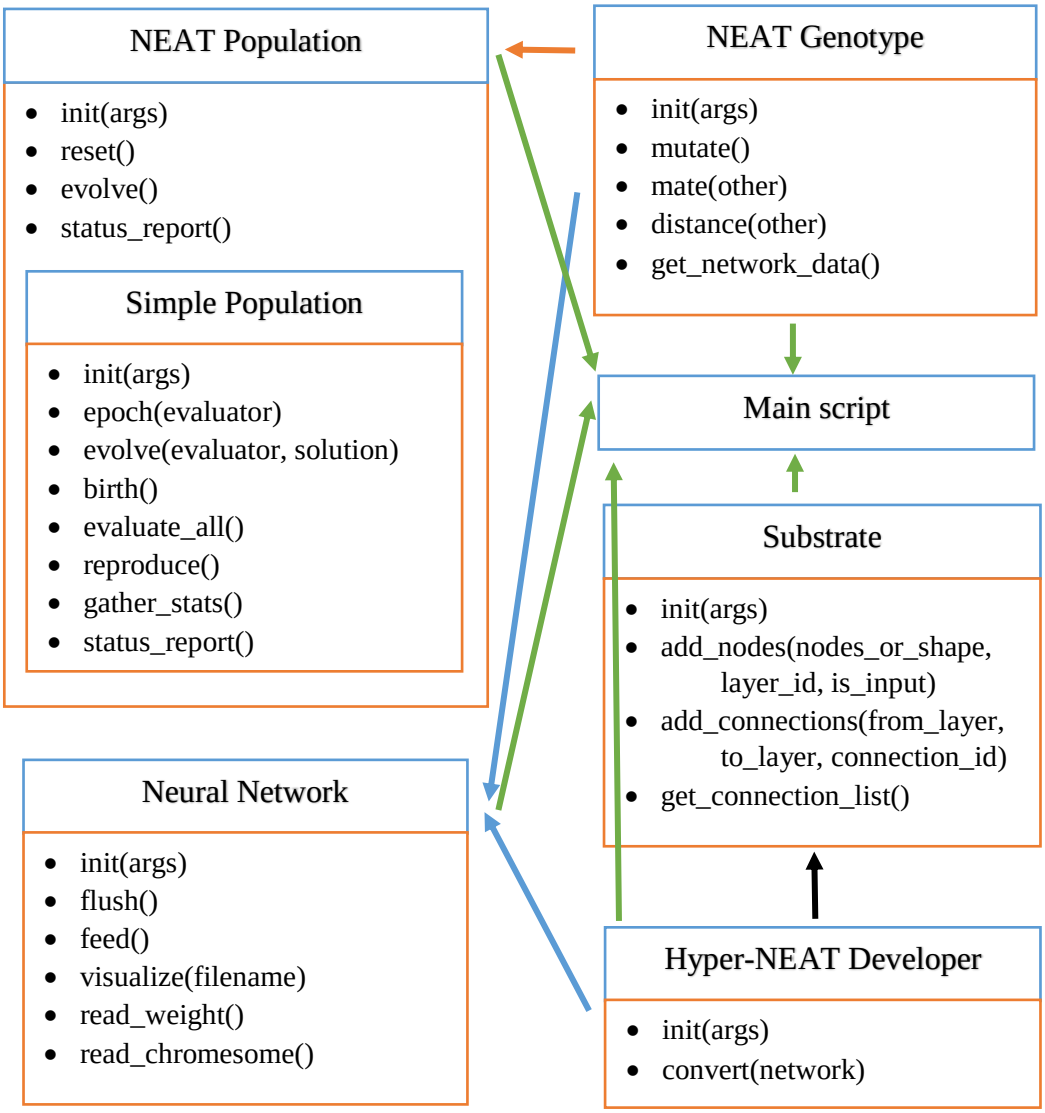


Figure 45: Implementation structure for NEAT and Hyper-NEAT

NEAT Population/ Simple Population

NEAT Population is a super class of Simple Population. Population class oversees the evolution process illustrated in Figure 15. Epoch() function stores a counter for generation, evolve() function is a pipeline that integrates birth(), evaluate_all(), find_best(), reproduce() and gather_state(). Birth() initializes the first generation; evaluate_all()tests each chromosome and defines the fitness value for each (accepts self-defined evaluator and solution function); find_best()outputs the champion chromosom; reproduce()handles crossover and mutation; gather_stat() collects the statistics. This class interacts with NEAT Genotype class, receiving the mutated and mated genes.

NEAT Genotype

This class is for the implementation of NEAT chromosome/ genotype and for the mutation, crossover and speciation process. It accepts parameters such as number of input output node, probability for adding node/connection, probability for mutation, distance excess/disjoint etc. that determine the development of structure.

Neural Network

Neural Network class defines the node activation functions, handles the conversion from NEAT genotypes and weight matrices to computable structures, as well as network visualization. Feed() works as a prediction function, returning the network calculation from an input vector. Flush() resets the activation values. It is implemented as a recurrent network, but through get_network_data() function, the network is converted to a feed-forward or a sandwich structure. It also implements the fully-connected neural net for Hyper-NEAT.

Hyper-NEAT Developer

Developer class takes in a substrate instance and converts the input network to a suitable CPPN.

Substrate

Substrate class configures the substrate for the Hyper-NEAT task. The object can be initialized with a set of node or shape as input. Further configuration is allowed through the function add_nodes() and add_connections().

Main script

Main script is needed for setting up the parameters and for defining the fitness evaluation and solution function. Its interaction with other component classes is mainly to feed the parameter. Evaluator() and solve() functions are transferred to NEAT Genotype class for genotype evaluation and fitness value distribution.

Parameter setting

A number of parameters have been set up for NEAT and Hyper-NEAT task. Below is a portion of the setting. See appendix for the full list.

Table 7: Parameter setting for Atari NEAT/Hyper-NEAT

Generation		**Population**		**Compatibility_Threshold**		**Fitness criteria**	
Atari	GW	Atari	GW	Atari	GW	Atari	GW
200	50-100	250,000	100-200	3.0		Max	

Max_stagnation		Old age		Survial_rate		elitism	
Atari	GW	Atari	GW	Atari	GW	Atari	GW
15	50	30	50	0.2		5	

Execution environment

It could be extremely time-consuming for personal computers without decent GPUs to run machine learning tasks, therefore, to guarantee the execution efficiency, all the experiments from this project were carried out in the AWS instance p2.xlarge. See Figure.46-47 for the instance information.

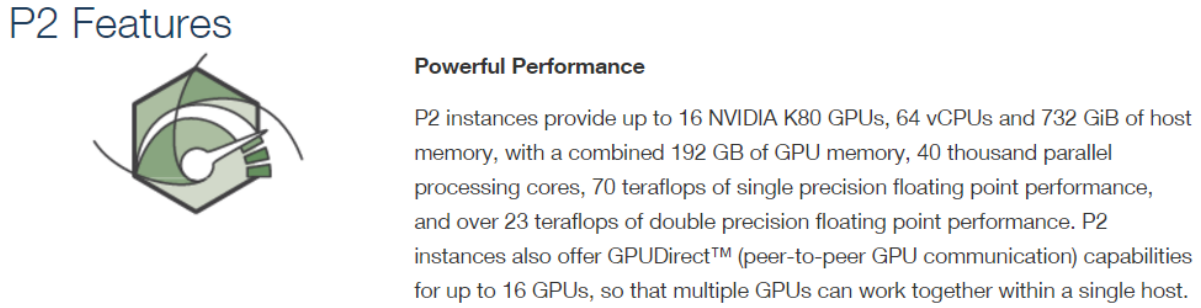
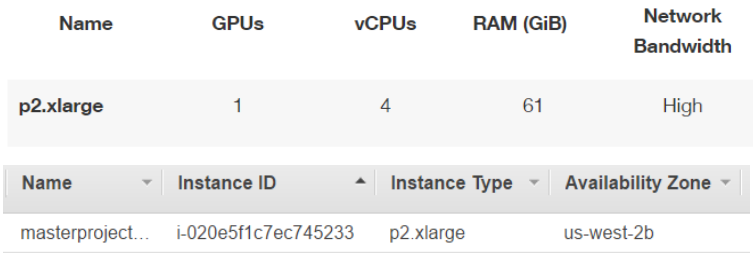


Figure 46: P2 instance performance



Name	GPUs	vCPUs	RAM (GiB)	Network Bandwidth
p2.xlarge	1	4	61	High

Name	Instance ID	Instance Type	Availability Zone
masterproject...	i-020e5f1c7ec745233	p2.xlarge	us-west-2b

Figure 47: Instance information for this project

4.3.3. Grid World

For Grid World task, the class structure for DQN is a bit different. Gird World DQN is compressed to a single class named DQNtask that implements the full pipeline of training procedures. All the task classes interact with a Grid World game instance, commanding the instance to change state and receiving from it the new state. Detailed class information and interactions are presented in Figure 48.

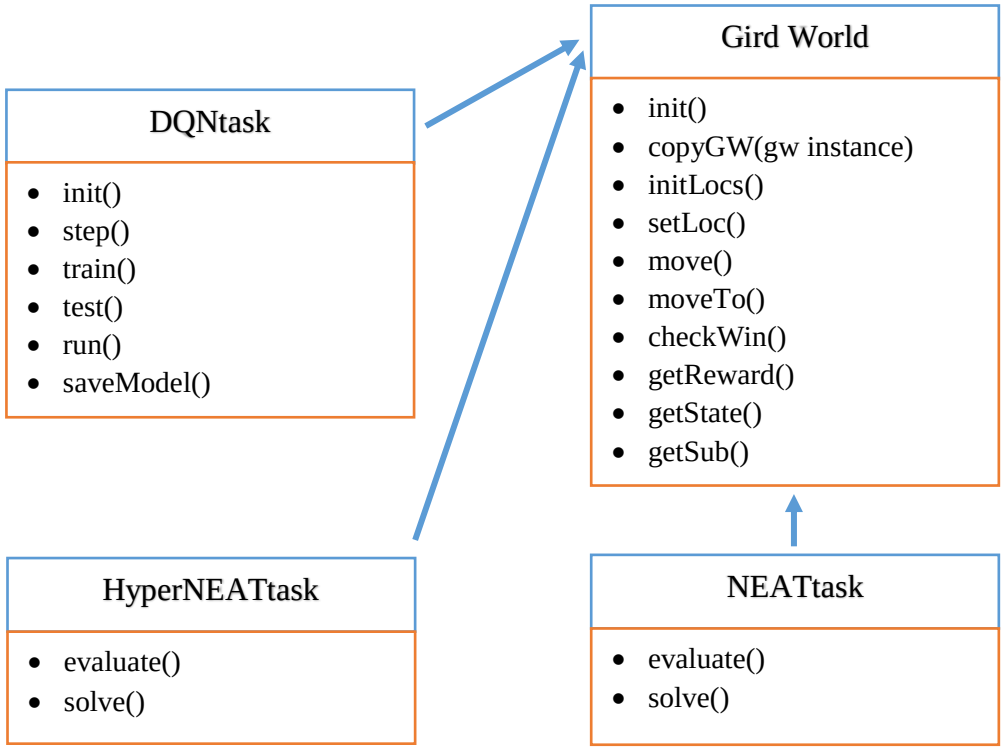


Figure 48: Implementation structure for Grid World tasks

GridWorld

GirdWorld class defines the game object. Instance is initialized through manual configuration or through feeding another GridWorld instance. By specifying the map height and width and the number of object types, the instance may be created with objects randomly arranged. The alternative initialization is to leave the map blank, and through setLoc() function, objects can be placed on the empty grid, which is also how the 27 maps are created.

CopyGW() carries out deep copy of the input instance, which is for storing the current state. There are two move functions, move() and moveTo(), and the former is for DQN task, the latter is for Hyper-NEAT. DQN task outputs the action value, while Hyper-NEAT sends the coordinate of the new state after making the action. GetSub function converts the state which is a numpy object of (width * height * #object types) dimensions (see Figure.49) to the size (width * height) with each position being signed with the object type number (0 for empty grid) (see Figure.50), which is needed for Hyper-NEAT tasks. Finally, there is a series of get functions for environment information query.

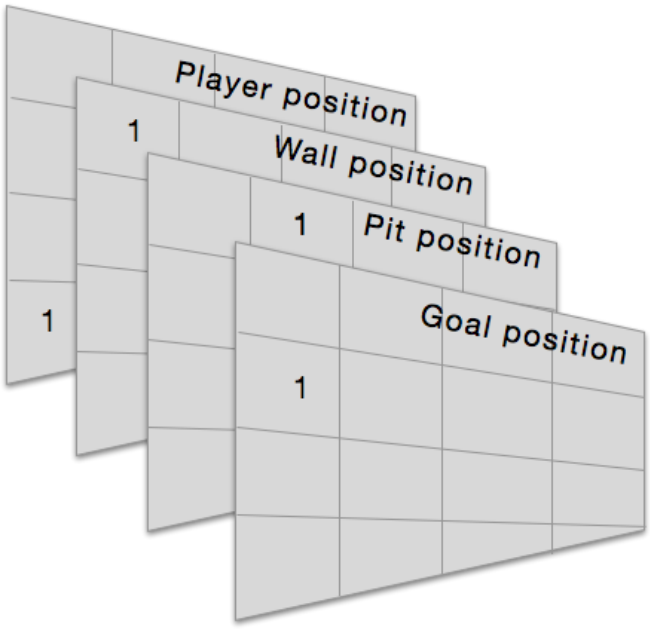


Figure 49: Gird World state of 4x4x4 dimensions [24]

1	0	0	0
0	2	0	0
0	0	3	0
0	0	0	4

Figure 50: Substate of size 4x4

HyperNEATtask/NEATtask

All the task classes for EA specify parameters and runs the program. It creates a NEATGenotype instance, a NEATPopulation instance, and for Hyper-NEAT task, a HyperNEATDeveloper instance. By calling the run function for NEATPopulation object, the task begins its evolution process.

DQNtask

GridWorld environment is much simpler than Atari, therefore, DQN algorithm is compressed to a single class that handles agent, experience replay and neural network. The former environment class has become a part of the GridWorld class. GridWorld DQN is implemented with Tensorflow Keras library that runs on GPU or multiple cores of CPU.

Chapter 5

5. Results and Analysis

This section presents the performance data in the form of line and column graph that emphasis on accuracy, time, stability and network structure. In the second half, comparative analysis based on them are stated.

5.1. Results

Data from Atari and Grid World environments are presented separately. Main discoveries are established on Grid World data, since NEAT algorithm fails at delivering satisfactory performance for the Atari task. Only win rate data is presented, considered that further comparison with one algorithm failing the task is pointless.

5.1.1. Atari

Figure 51 and 52 show the training progress of both algorithms in Atari Breakout and Pong with x axis being epoch number and y axis being rewards received in testing. DQN rewards steadily climbs up within the first 50 epochs and maintains roughly at score 425 for Breakout and 21 for Pong afterwards.

According to the Breakout leader board on steam, top score for Breakout is 576pt, which is also the highest score on record, and the highest score from DQN gameplay ranks the 3rd in steam. The median of the leader board is close to 300pt, therefore DQN indeed achieves super-human level control. Looking at the score distribution, the testing score fluctuates much strongly than that in Pong game, and the performance peaks and drops from time to time, which can be explained that the network prediction and Q value training for Breakout is less accurate than Pong.

On the other hand, NEAT performance stably maintains at the same level as random plays, gaining the lowest score possible. For Breakout, NEAT occasionally receives 1 point per round, but mainly lingers at random level. For Pong, the performance seems to be peaking more frequently, but the maximum score never surpasses that of random plays, and in most cases, it loses all 21 of play, receiving score -21.

Since NEAT fails at delivering decent gameplay for both games, it can be concluded that NEAT is unable to evolve workable solutions in Atari environments, let alone competing against DQN's super-human performance. Hypothesis of explaination for the failure is addressed in the analysis section.

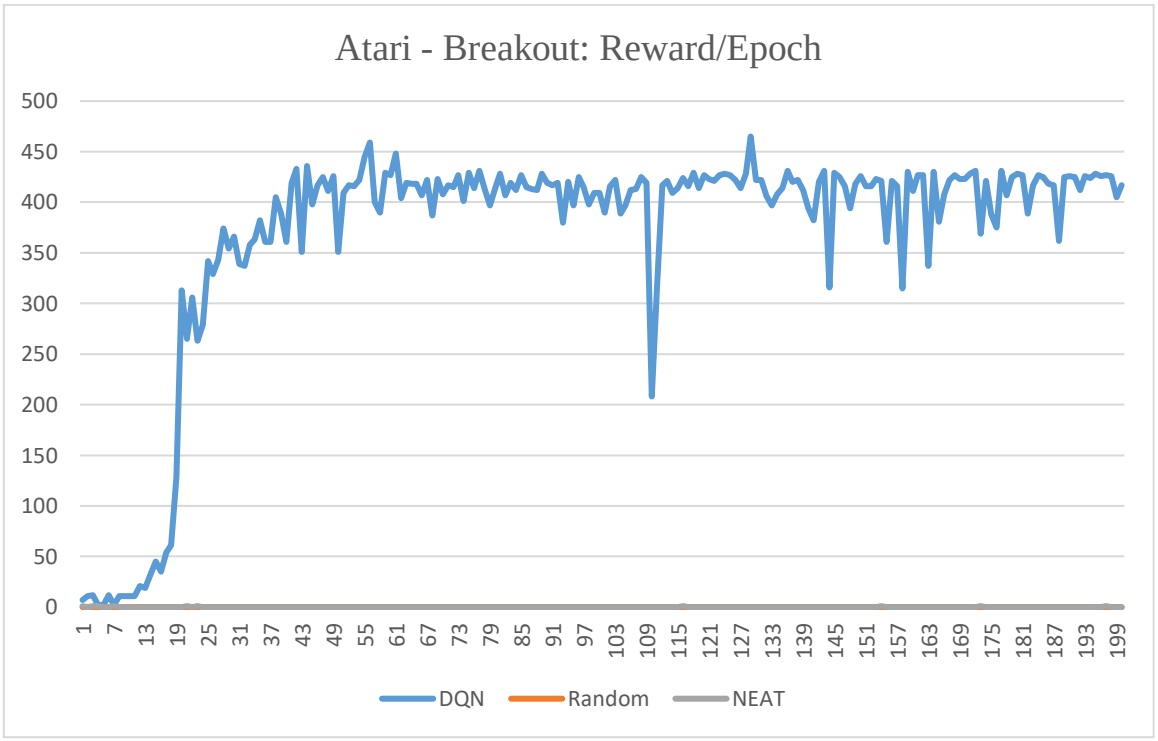


Figure 51: Rewards plotting for Breakout game

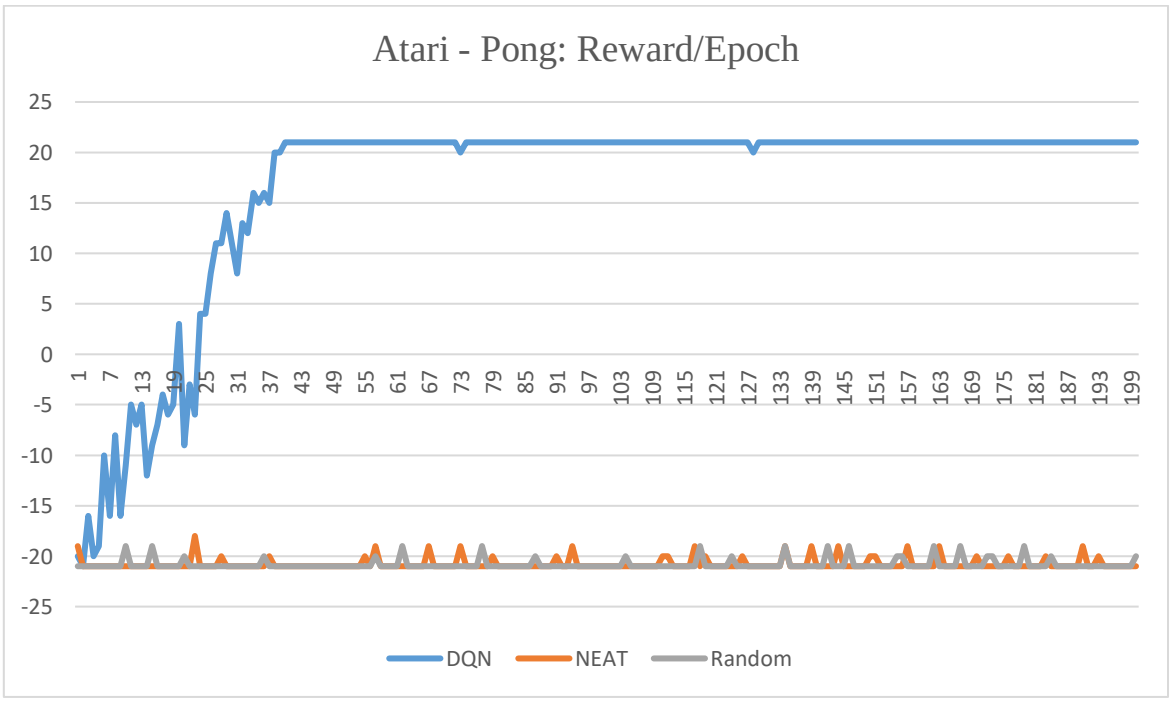


Figure 52: Rewards plotting for Pong game

5.1.2. Grid World

Gird World Data is based on the performance of DQN and Hyper-NEAT. NEAT fails all the stochastic Grid World tasks and most of the deterministic tasks, therefore NEAT results are excluded from analysis. However, the failure statistics will be mentioned in the analysis section while attempting to explain its incompetence.

Overall Time Usage

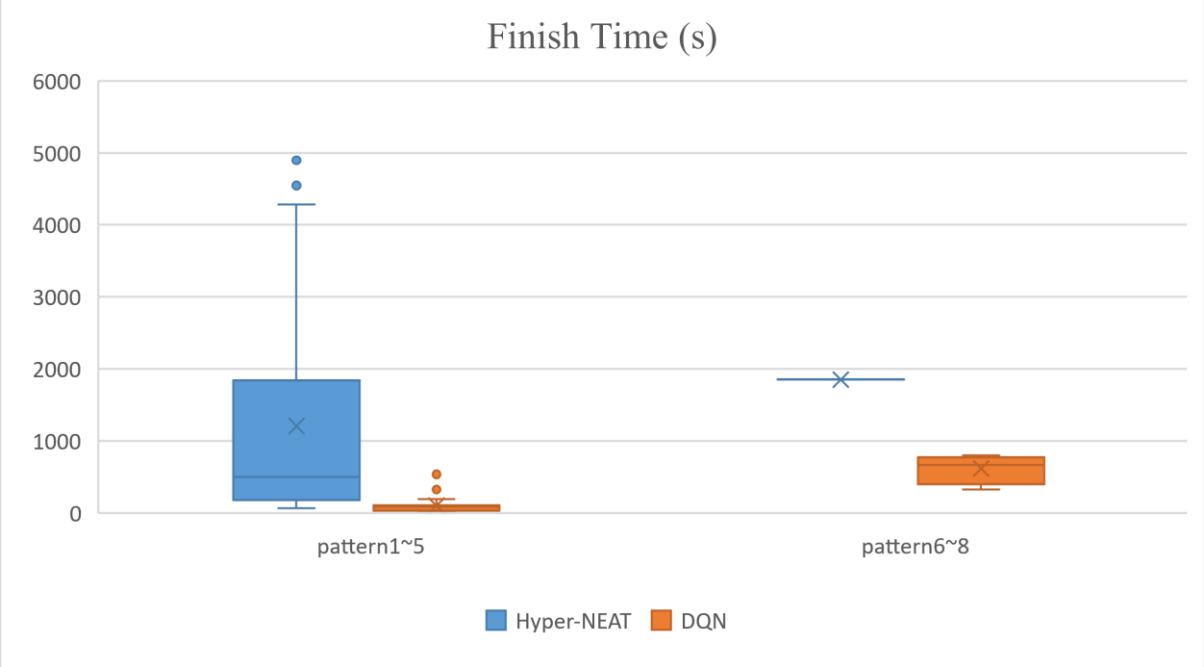


Figure 53: Finish Time for eight patterns

The overall time usage comparison only looks at the solved maps, since training for unfinished maps always take up to the full specified number of epochs, the comparison of which can be done with a pen and paper and is unnecessary for this project. By comparing the finish time distribution, this project examines which algorithm takes more time to train/ evolve a solution, and how much does the number varies.

Results (Figure.53) show that in the worst cases of map pattern 1 to 5, Hyper-NEAT may take up to more than one hour to successfully evolve a fully-worked solution, while DQN takes about 535 seconds, a 10^{th} of that. On average, Hyper-NEAT takes 20 minutes to evolve a solution, which is 19 more minutes than DQN who only takes 99 seconds. DQN time distribution is compact, with 108 seconds of variance, which is less than a 10^{th} of Hyper-NEAT's 1459 seconds. Results for pattern 6 to 8 seems to contradict the observation for pattern 1 to 5, but this is because Hyper-NEAT only solves one map, while DQN solves four. For

pattern 6 to 8, Hyper-NEAT spends 1851 seconds to evolve the fully worked solution. This is still 20 minutes larger than DQN's 614 seconds. From looking at this graph, DQN does take more time to solve a larger and more complicated map pattern. The overall time usage over all eight patterns is 1228 seconds for Hyper-NEAT and 167 seconds for DQN. Both algorithms were running on GPU architecture of the same computation environment. Therefore, it can be concluded that DQN is much more time efficient than Hyper-NEAT.

Size-based Time Usage

Time attribute is then plotted separately per map size to discover the correlation between epoch time and map sizes. From Figure 54, one may observe a polynomial correlation in Hyper-NEAT results. This project fits the values to the degree-2 polynomial and receives the following estimate function:

$$\text{avg} = \text{f(size)} = 7.114size^2 - 6133size + 1.379e04 \;\; (SSE = 1.143e06)$$

This shows that Hyper-NEAT evolution time always grows with the map size, however, DQN does not demonstrate a positive correlation as strict. DQN's average training time for size 5 is less than size 4, and size 9 less than size 8.

Table 8: Size-based time usage.

	DQN		*Hyper-NEAT*	
	Avg	**Std**	**Avg**	**Std**
Size 4	91.66057	144.43	843.6363	1218.073
Size 5	81.67789	43.63266	1020.604	1386.279
Size 6	101.406	16.70635	1806.267	1903.095
Size 7	745.4577	**446.2753**	5754.633	759.1616
Size 8	865.7386	156.6698	10807.98	1426.229
Size 9	818.5313	196.3652	16433.82	**3305.591**
Size 10	**892.6829**	89.22411	**23278.76**	3278.409

The standard deviation of DQN training time peaks at size 7, presenting no apparent correlation with map size either. While the same value for Hyper-NEAT grows with maps of the same pattern set.

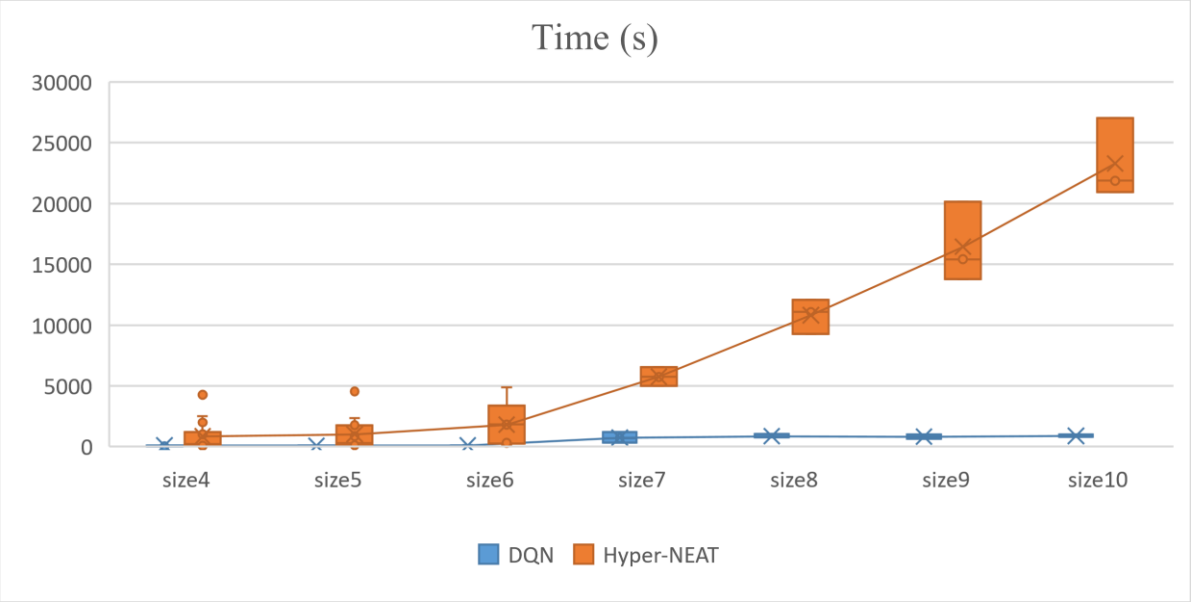


Figure 54: Size-based time usage

Pattern-based Time Usage

The correlation between time usage and map patterns is less obvious, because test step threshold is specified pattern-wise, and test time constitutes largely to the differences of time usage, especially for Hyper-NEAT (time difference = test number * step threshold * population size * generation size). However, pattern 1 to 5 has equal step threshold, and their results show that Hyper-NEAT's average time usage exhibits positive correlation with map difficulty, and the range of deviation increases as pattern complexity grows. However, the same is not true for DQN, whose time usage fluctuates intensely for pattern 3 and 6 with standard deviation 180 and 230.

Table 9: Pattern-based time usage.

	DQN		*Hyper-NEAT*	
	Avg	**Std**	**Avg**	**Std**
Pattern 1	56.40586	32.62314	100.4128	31.54119
Pattern 2	51.20999	39.94566	255.7959	59.87401
Pattern 3	191.3638	180.9592	842.0834	918.6748
Pattern 4	64.41662	22.59323	2438.225	1471.425
Pattern 5	65.81662	24.92242	2466.67	1839.323
Pattern 6	644.7758	230.5918	13491.32	6150.854
Pattern 7	1068.873	97.58749	16059.8	9573.667
Pattern 8	778.1589	67.32832	12655.28	6964.242

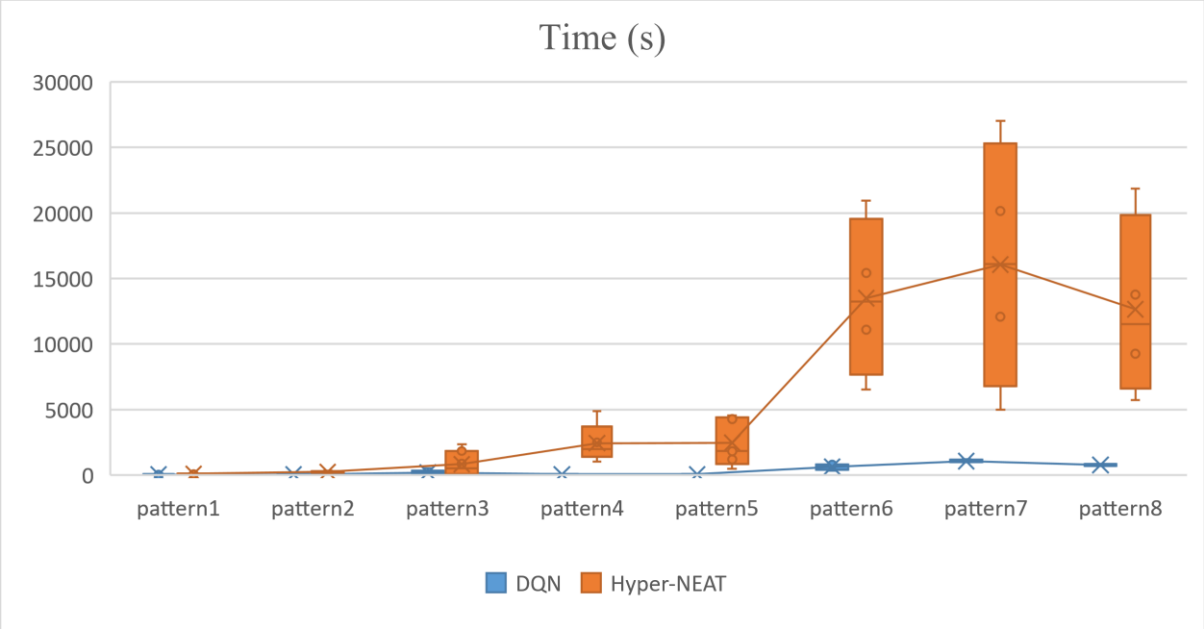


Figure 55: Pattern-based time usage

Overall Accuracy

Both DQN and Hyper-NEAT achieves 100% win rate for maps of pattern 1 to 5. The accuracy of DQN solutions for pattern 6 to 8 averages at 73% with four of the solutions reaching 100% win rate in map 7x7 pattern 1 and 3, 8x8 pattern 1 and 9x9 pattern 1; map of size 10x10 pattern 1 gains 97.5% accuracy, and with further training, it should be able to achieve 100%. DQN average win rate is 17.33% higher than Hyper-NEAT for pattern 6 and 8. Looking at the overall statistics, it is obvious that DQN outperforms Hyper-NEAT regarding to accuracy over all the patterns.

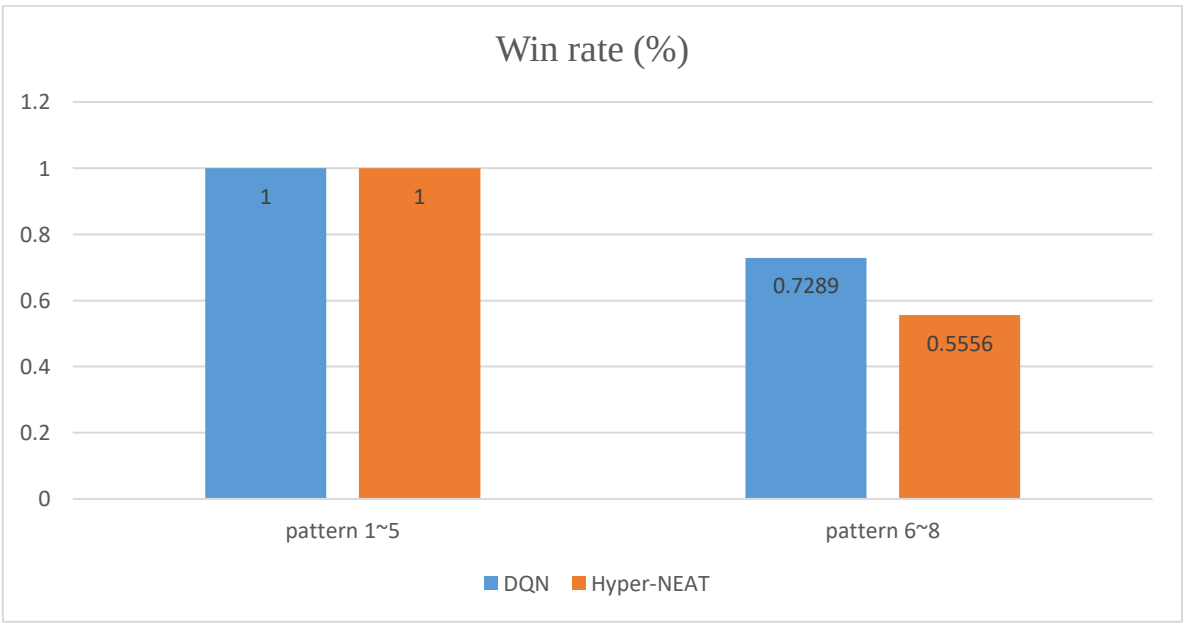


Figure 56: Win rate over eight patterns

Size-based Accuracy

This project looks at the average and standard deviation value for accuracy. Average of accuracy discovers how the algorithm's overall win rate changes as map size grows up, and standard deviation examines to what degree is accuracy determined by the size attribute. Only statistics from map larger than 6x6 is listed in the table below since both have reached full accuracy uniformly (Figure.57).

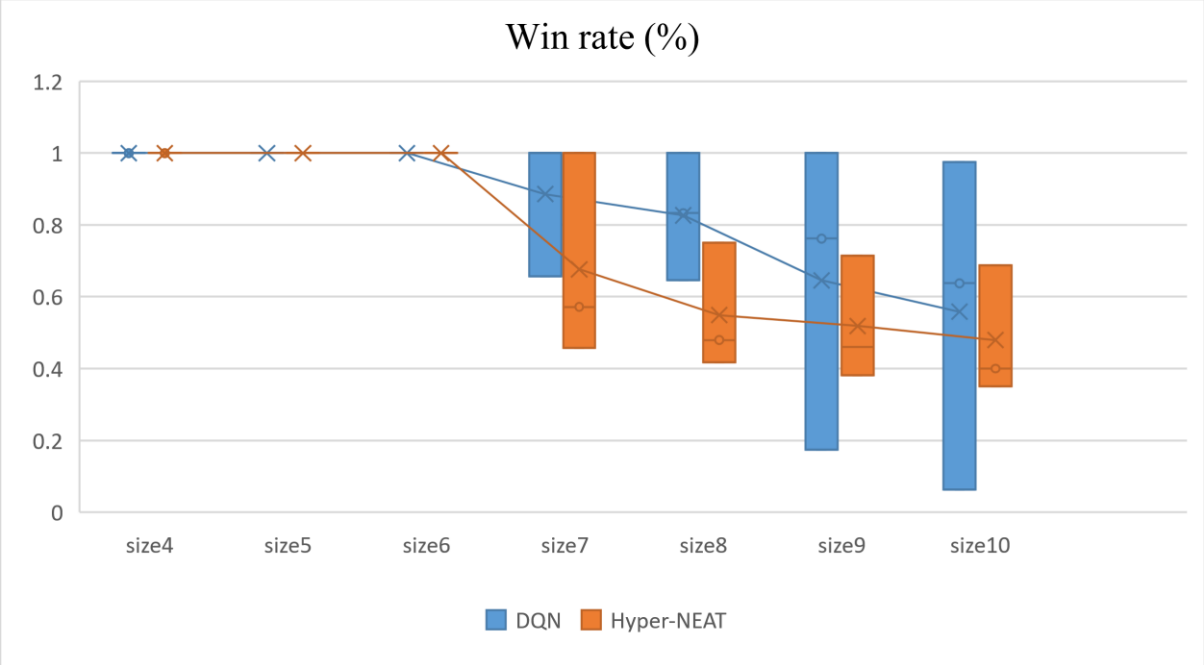


Figure 57: Size-based accuracy

Table 10: Size-based accuracy.

	DQN Win Rate			*Hyper-NEAT Win Rate*		
	Avg	**Std**	**Max**	**Avg**	**Std**	**Max**
Size 7	0.885714	0.197949	1	0.67619	0.286190	1
Size 8	0.826389	0.177185	1	0.548611	0.177185	0.75
Size 9	0.645503	0.424831	1	0.518519	0.174122	0.714286
Size 10	0.558333	0.461373	0.975	0.479167	0.182145	0.6875
Average	0.72899	0.315334		0.555622	0.204911	

Looking at the numbers in table 10, both algorithm's accuracy decreases steadily along the way, at the same time, DQN's win rate always remains higher than Hyper-NEAT. DQN achieves full accuracy in more maps than Hyper-NEAT too. Standard deviation values suggest

that DQN accuracy for maps #22 -27 (size 9 to size 10. See appendix for map code and details) is less effected by size but much more by other attributes, while Hyper-NEAT is determined much strongly by the size attribute.

Pattern-based Accuracy

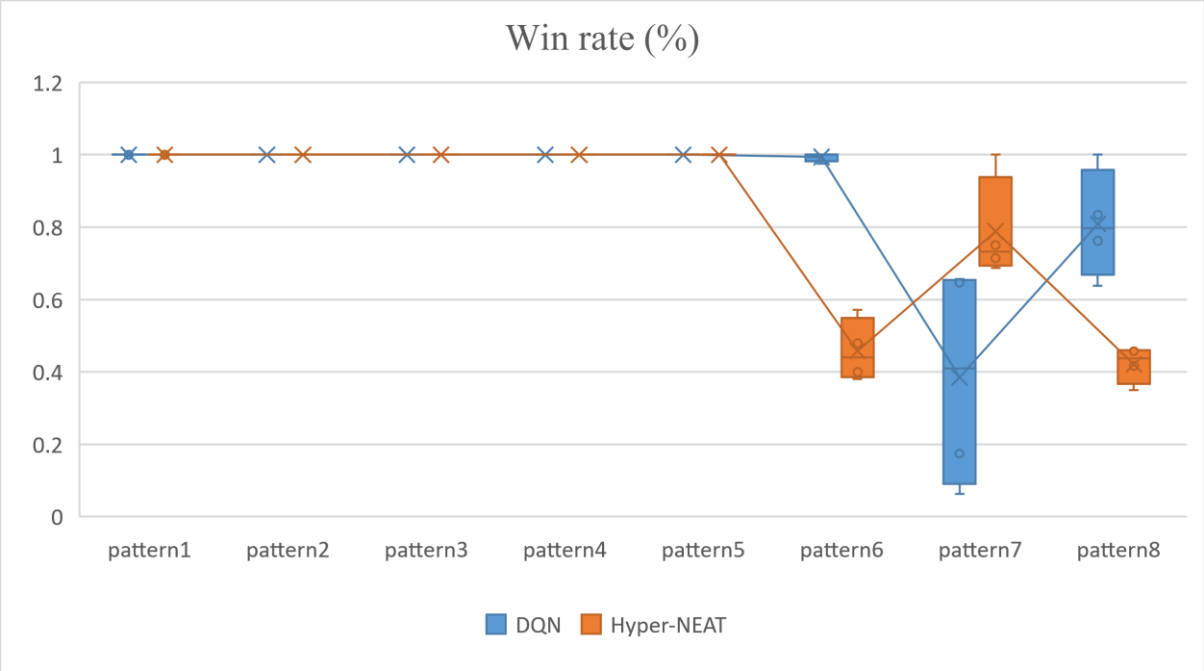


Figure 58: Pattern-based accuracy

Only results for map of pattern 6 and above are listed in details due to the same reason addressed above. Pattern-based standard deviations for DQN and Hyper-NEAT algorithms are 0.1583 and 0.0939, 50% less than size-based variance, indicating that algorithm accuracy is greatly determined by map patterns rather than map sizes. For pattern 7 and 8, other attributes still play a role in explaining DQN's performance, considered that their variances are one magnitude larger than pattern 6.

Table 11: Pattern-based accuracy.

	DQN			*Hyper-NEAT*		
	Avg	**Std**	**Max**	**Avg**	**Std**	**Max**
Pattern 6	0.99375	0.0125	1	0.457887	0.0868	0.571429
Pattern 7	0.38502	0.3111	0.761905	0.787946	0.1437	1
Pattern 8	0.80819	0.1513	1	0.421032	0.0513	0.460317
Average	0.72899	0.1583		0.555622	0.0939	

The Average of DQN accuracy is surpassed by Hyper-NEAT for map pattern 7, which is the only scenario where Hyper-NEAT outperforms, and the reason behind this is discussed in the analysis section.

In summary, DQN is generally better at training solutions of higher accuracy with the only exception pattern 7.

Stability

This project repeats the experiment 3 times for map #1-8 to observe each algorithm's resistance to randomness. Accuracy statistics in table 12 are the same, suggesting that both algorithms always solve the task within epochs specified unaffected by randomness. The maximum time usage for DQN is 535, which is almost a 5^{th} of Hyper-NEAT's maximum time. Judging from the standard deviation value, the range of time usage for Hyper-NEAT is 7 times wider than DQN, and average time usage is almost 6 times more. In summary, DQN demonstrates impressive stability in different runs, and Hyper-NEAT is considerably more vulnerable to the random factor than DQN.

Table 12: Accuracy statistics over eight maps.

	DQN	*Hyper-NEAT*
Max	1	1
Min	1	1

Table 13: Time usage statistics over eight maps.

	DQN	*Hyper-NEAT*
Avg	94.19508	579.5827
Std	115.5992	774.2389
Max	535.1348	2367.507
Min	25.12575	66.82734

In terms of the progression stability in a single training/ evolution process, this project looks at whether the win rate of the next epoch is larger than the current. Therefore, by calculating the percentage of positive and negative win rate differences, this project is able to tell which algorithm utilises the previous knowledge more effectively and progresses steadfastly.

Figure 60 and 61 demonstrates the percentage of difference sign. DQN has 56.28% of the next accuracy value larger than the current, which slightly surpasses Hyper-NEAT by 1%;

6% less Hyper-NEAT epochs are making no progress, but 13% more fails at maintaining last epoch's progress.

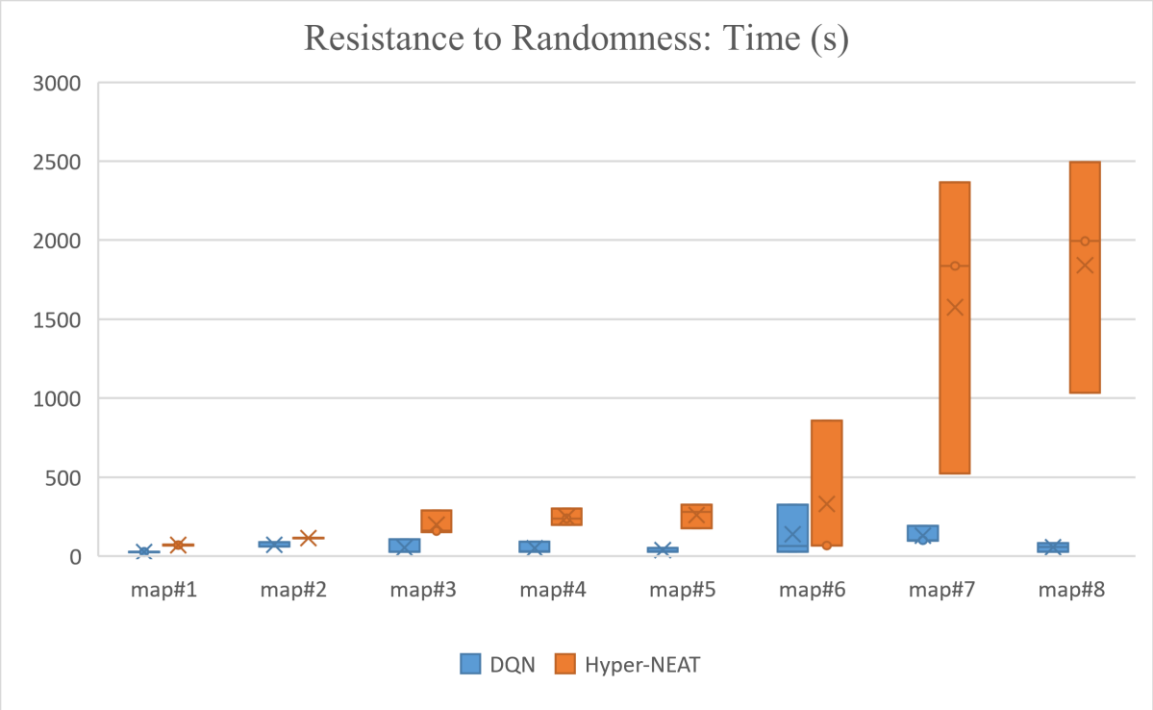


Figure 59: Resistance to Randomness map-specific

In summary, both algorithms are making the same level of positive progress, but Hyper-NEAT regresses more frequently than DQN.

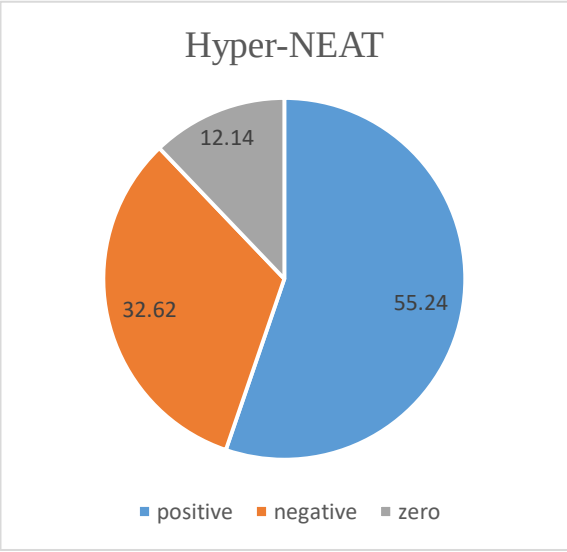


Figure 60: Positive, negative and zero difference rate of Hyper-NEAT

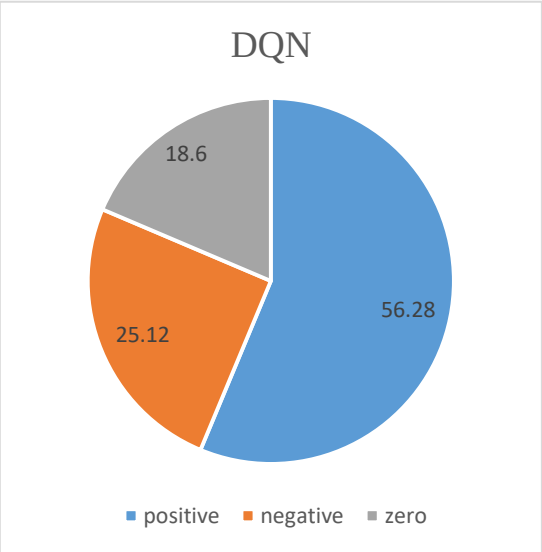


Figure 61: Positive, negative and zero difference rate of DQN

5.2. Analysis

Analysis is built on the results demonstrated above, and is presented under titles of discovered phenomenon with reasoning that attempts to explain each phenomenon.

5.2.1. NEAT Fails at Performing in Atari and Grid World Environment

Require Pre-processing and Object Representation

Figure 51 and 52 as well as the description in the results section have proved that NEAT fails at providing workable solution for Atari game tasks. As described in Methods - Implementation, this project implements NEAT that takes in raw pixel data of 84 x 84 dimensions as input. However, according to [15], which is one of the very few experiments recorded that successfully utilises NEAT for controlling Atari game agent, it requires heavy pre-processing and a substrate of object representation for NEAT to work in this condition. The input dimension is downsized to 21 x 16, and objects were manually labelled. The evolution process is not entirely free from human involvement and requires domain-specific knowledge, therefore, following this would be unfair to DQN and would render the comparative results invalid. However, this information may explain the incompetence of NEAT – the evolution process fails at keeping up with high dimensional input data within small generation size.

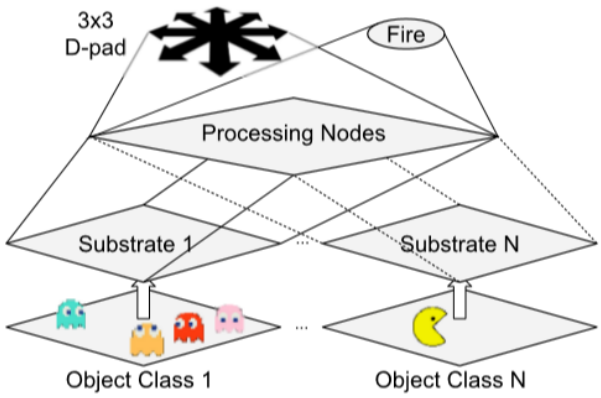


Figure 62: The architecture implemented by [15] that receives object classes as input

The Weaknesses of Direct Encoding

NEAT has also been executed in Grid World environment deterministically and stochastically. Deterministic performance (always initialized at the same spot) shows that NEAT is only able to solve the first two patterns, and for the rest of maps it consistently returns the lowest possible score. Deterministic environment requires NEAT to find the right solution in one epoch, the possibility of which may decrease drastically as the function complexity

grows up, therefore, only the two simplest patterns are solved. Stochastic results exhibit the performance plateau for all patterns (Figure.63).

NEAT is unable to solve the easiest patterns if being randomised at different spots at the beginning of train episodes, suggesting that NEAT cannot integrate its previous path finding experiences if deviating from the initial starting point, because it requires heavy adjustment on the network structure. As illustrated in Background section, NEAT implements direct encoding scheme, hence one generation of evolution is mapping to one small structure change. Therefore, if evolution of function is like building a house, to estimate a 7-degree polynomial is like to build from scratch a skyscraper with tiny changes at a time, which could take thousands or tens of thousands of generations to complete. Figure 64 to 66 shows a few structures over which NEAT stagnated for dozens of generations, however, the final structure it settles on is not guaranteed to be correct, (solution represented by Figure.66. has 0% win rate) which leads to the second problem with direct encoding – the evolution is trivial.

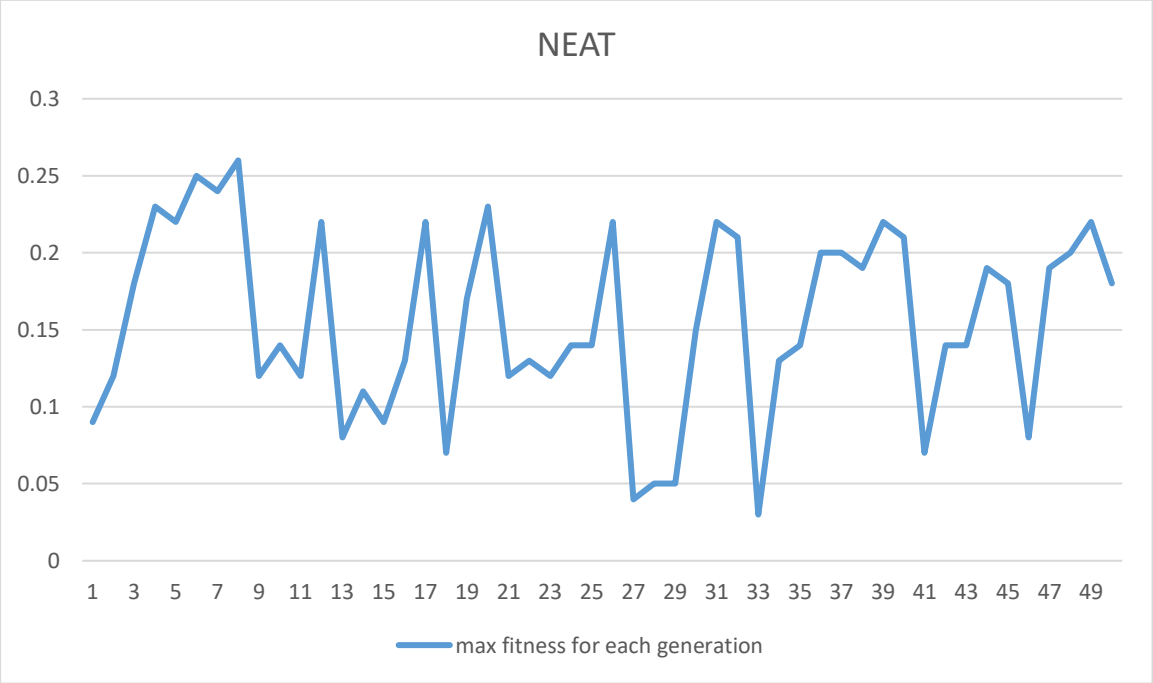


Figure 63: Progression of max fitness value for NEAT in stochastic environment

NEAT, like DQN, thrives on rewards, or fitness growth. Since Grid World environment only set up one position that returns positive rewards, NEAT cannot rely on reward accumulation, and has to make one guess and the guess has to be correct. If the function is a n-degree polynomial (n>10), this is equivalent to asking an organic to evolve to be human in one step, next to impossible, explaining its long generations of futile evolution.

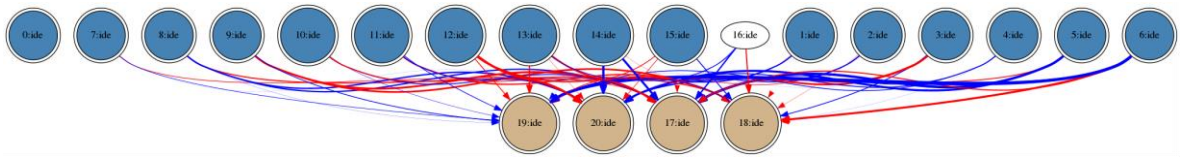


Figure 64: CPPN structure in 10th iterations for map #8

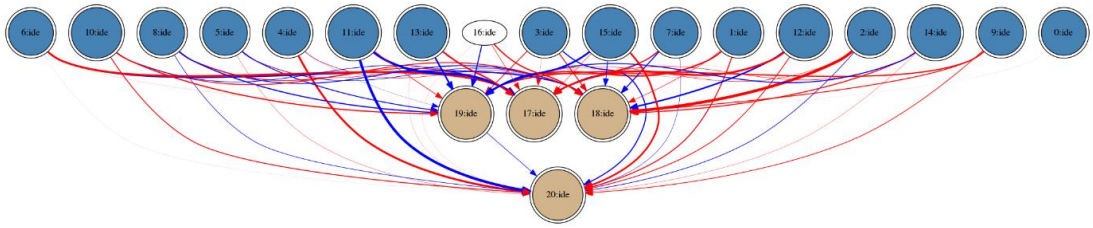


Figure 65: CPPN structure in 30th iterations for map #8

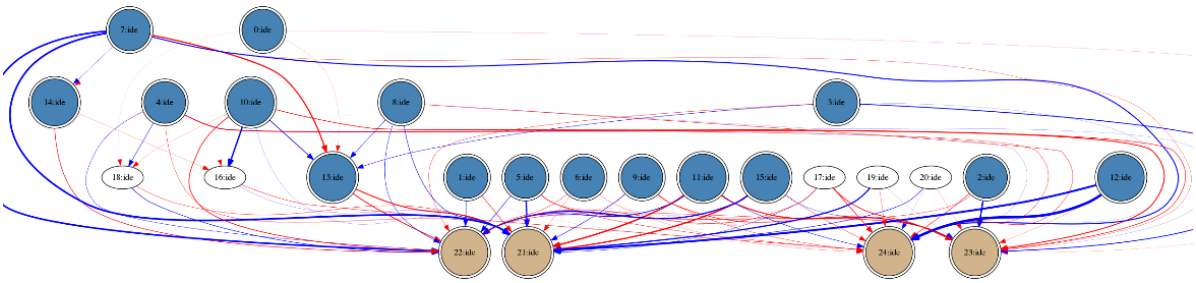


Figure 66: CPPN structure in 50th iterations for map #8

Hyper-NEAT, on the other hand, evolves CPPN that defines weights for a fully-connected ANN that outputs action value. Different from NEAT, one small change in CPPN results in disproportionally larger adjustments for the ANN, hence drastically alter the final output, achieving both regularity and modularity. As shown in Figure 67-69, the same task only takes three generations for Hyper-NEAT with tiny changes on weights. In addition, since CPPN feeds on object coordinates, this modularity is effectively established on environment geometry. Through this, evolution being guided by "eyes" is much more efficient. This visual mechanism is similar to DQN's three layers of convolutional filters, which also help with recognizing objects.

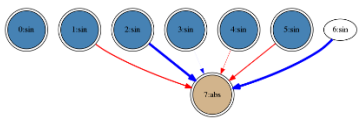


Figure 67: HyperNEAT Evolution at iteration 1

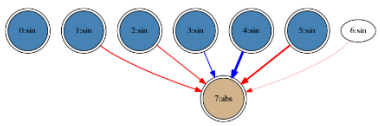


Figure 68: HyperNEAT Evolution at iteration 2

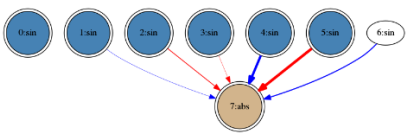


Figure 69: HyperNEAT Evolution at iteration 3

5.2.2. DQN wins at efficiency

Figures 70-75 show the typical distribution of accuracy per epoch. The number of point on the graph indicates the number of epoch/generation the algorithm takes before achieves full accuracy.

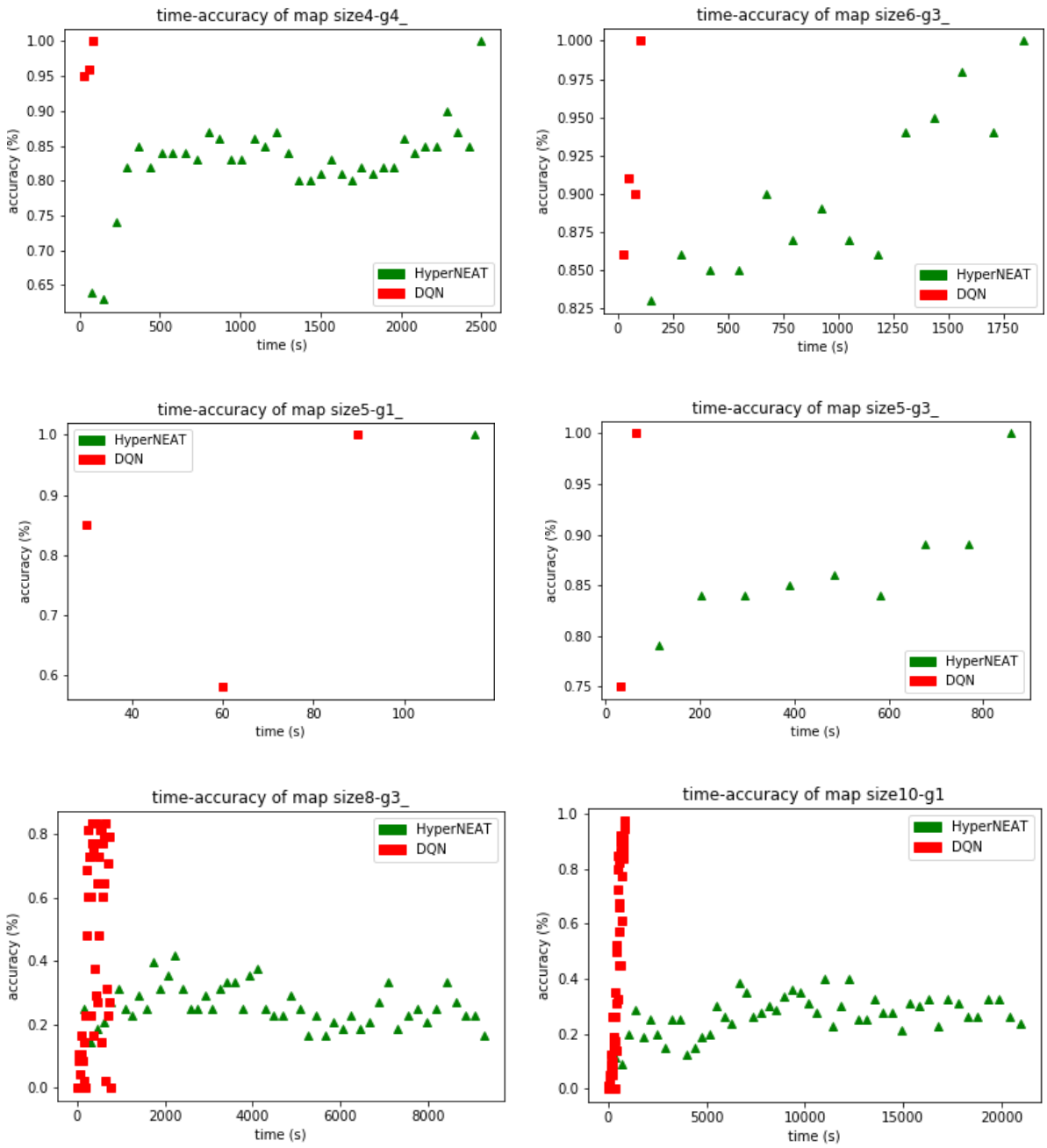


Figure 70, 71, 72, 73, 74, 75: Time-accuracy plotting for map #4, 13, 6, 8, 21, 25 (left to right, top to bottom)

Generally, DQN takes less epochs, and the shape of distribution is slim and tall – slim is due to shorter x distance – smaller time usage, and tall is due to larger y distance – bigger win rate/ accuracy. The plotting together with graphs and tables presented in Results section,

conclusion can be drawn that DQN possesses much higher efficiency at solving the Grid World task.

Smaller Time Usage

Table 5 presents the implementation logic for Hyper-NEAT and DQN. Both programs implement two nested loops, and the two iteration numbers are set to be equal. However, the difference occurs at the evaluation step. DQN only evaluates the solution at the end of the epoch, while Hyper-NEAT executes the same process for every chromosome of the generation for fitness evaluation. Evaluation is in the first loop for DQN, but the inner-most loop for Hyper-NEAT. If being translated in equations:

$$\mathrm{O}(\mathrm{HyperNEAT}) = \mathrm{O}(generation * evaluation * evaluation)$$
$$\mathrm{O}(\mathrm{DQN}) = \mathrm{O}(epoch * train\ step)$$

Looking at table 8, full evolution (50 generations) is at least 10 times longer than full epoch training, proving that network training, which is in the inner-most loop of DQN logic, is less costly than evaluation. Apart from that, Figure 54-55 shows that DQN generally takes less epochs too which further speeds up the training process.

Higher Accuracy

DQN achieving higher accuracy proves that the mathematical framework designed specifically for RL problems is more accurate at solving reinforcement learning problems in shorter time than evolutionary algorithms.

5.2.3. DQN shows higher resistance to randomness

According to table 12 and 13, the only factor effected by randomness is time usage, and performances of both algorithms demonstrate certain level of variance. On average, the variance for DQN is controlled within 2 minutes, presenting much smaller deviation range than Hyper-NEAT that varies at 13 minutes. Looking at Figure 59, this variance is map-specific – it increases as map size and difficulty grows up (map#6, 7 and 8).

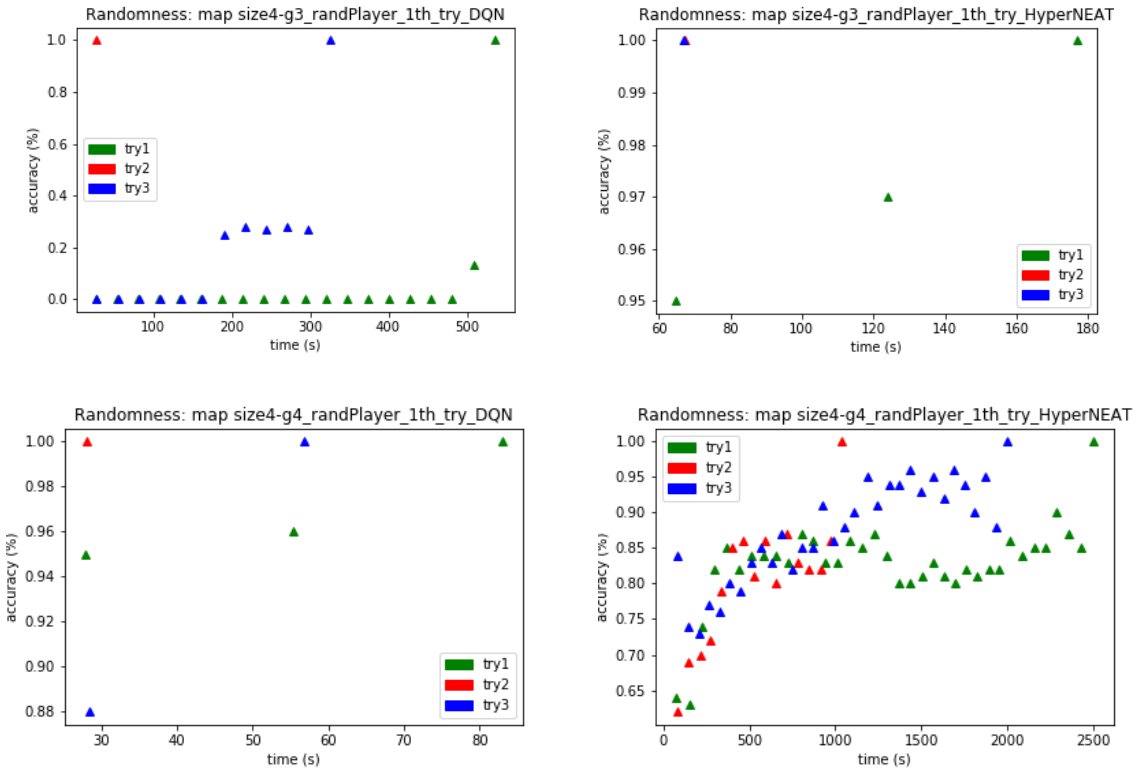


Figure 76, 77, 78, 79 : Accuracy-time plotting over three tries for map #3 and 4 (left to right, top to bottom)

Both algorithms require certain luck element (favourable randomisation) to finish in shorter time. As Figure 77 presents, the first generation of solutions, which are absolute randomisation from scratch, may already contain a 100% accuracy solution, hence the evolution terminates within one generation. Larger population has higher chances of that happening, therefore has higher resistance to randomness, but the trade-off is longer evolution. The same applies to reinforcement learning – take Figure 76 as example, the agent may keep falling into the pit from some spot, or the evaluation episode only places the agent on the untrained Q map sequences, which explains the stagnation at zero for a few epochs. In cases demonstrated in Figure 78, one try's performance may surpass the other even the previous generations were making smaller progress. With luck, the agent may make more optimal decisions at the exploration stage, and get initialised at untrained positions for most of training episodes, and solves the problem for only one epoch, which is the same scenario as the "try2" for map #4. However, the evolution time for each generation is about 6 times longer than a training epoch, explaining the overall time variance.

It is mentioned above that map size and difficulty positively correlate to Hyper-NEAT time usage variance, and Figure 50, 74 and 75 altogether prove that Hyper-NEAT indeed takes

more generations (more plots on the graph) to solve more complicated patterns, which leads to higher chances of large deviation. Since the substrate size is equal to map size, larger map indicates more nodes on the substrate, hence more weights to be computed by the CPPN described by each chromosome, and it would be more difficult for chromosomes to evolve the CPPN that correctly describes that many weights. For DQN, map of larger size and more difficult pattern takes more steps to reach the goal, however, this number is restricted by a threshold, therefore the variance is controlled in a much smaller range than Hyper-NEAT.

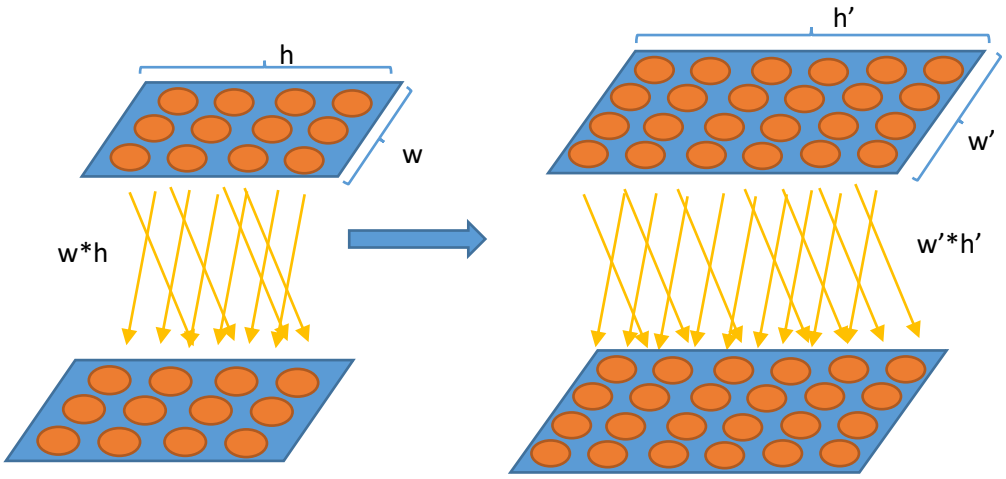


Figure 80: Larger map size leads to more substrate nodes and more weights

5.2.4. Hyper-NEAT shows diversified solution structures

Size

In Grid World environment, DQN structure takes in a vector of substrate nodes (see Implementation - GridWorld section for details), has two fully-connected layers of size 164 and 150 (see Figure.82). Total weight number is

$$\text{weights} = \text{w} * \text{h} * 164 + 164 * 150 + 150 * 4 = 41600\ (if\ w = h = 10)\ .$$

For Hyper-NEAT, the network that participate in prediction has an input layer of size $\text{w} * \text{h}$, and an output layer of size $\text{w} * \text{h}$ (see Figure.83.), total weights number is

$$\text{weights} = \text{w}^2 * \text{h}^2 = 1\text{E}4\ (if\ w = h = 10),$$

which is four times less than DQN weights. As long as the map size is lower than 16.145, Hyper-NEAT ANN weights will always be less than DQN weights ($16.145^2 \times 164 + 164 \times 150 + 150 \times 4 \approx 680{,}000 \approx 16.145^2 \times 16.145^2$). If applies Hyper-NEAT to Atari

games, the network will implement three additional convolutional filters (see Figure.81.), and the weight number becomes

$$\text{weights} = (8 * 8 * 4 * 32) + (4 * 4 * 32 * 64) + (3 * 3 * 32 * 64 * 64) + (7 * 7 * 64 * 518) + (518 * 18) \approx 1\text{E}6.$$

Hyper-NEAT on the other hand has weight number

$$\text{weights} = (84 * 84) * (84 * 84) \approx 1\text{E}7.$$

DQN implements more layers in Atari environment, but the weight number dropped instead and becomes one magnitude smaller than Hyper-NEAT ANN. This is essentially due to the weight-sharing nature of convolutional neural network filters.

In summary, for larger problem size, Hyper-NEAT neural network structure can be much larger than Q learning network, but for smaller and simpler problems, Hyper-NEAT could obtain lighter network weights.

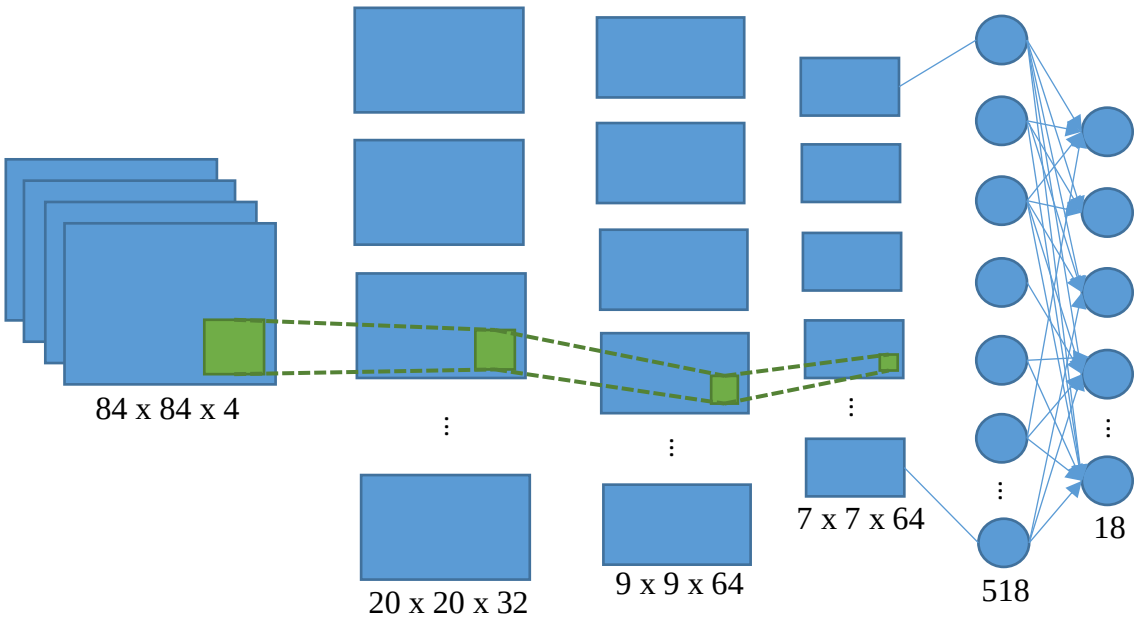


Figure 81: DQN for Atari task

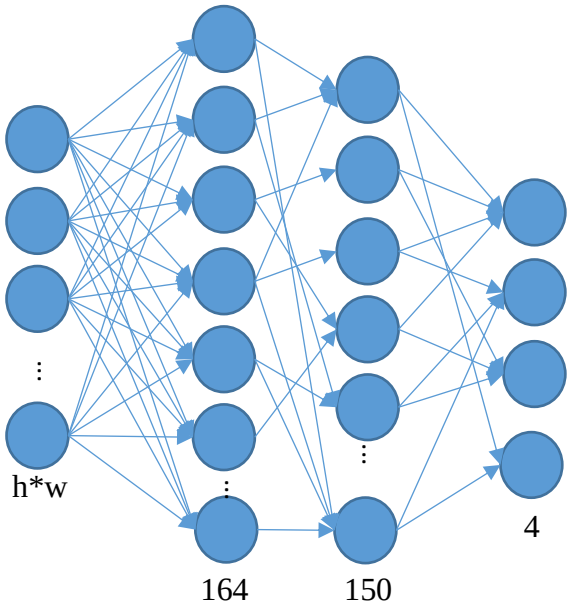


Figure 82: DQN for Grid World task

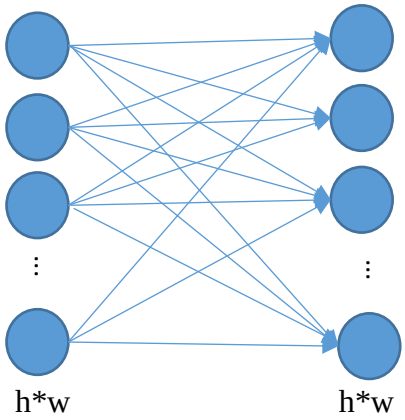
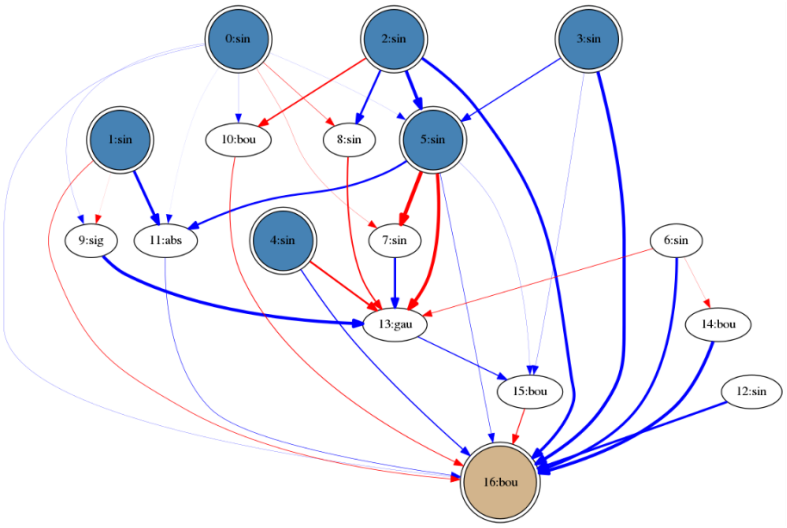


Figure 83: Hyper-NEAT ANN

Diversity

Network structure for DQN and Hyper-NEAT ANN is pre-fixed, but the CPPN structure is evolved from scratch. Multiple CPPNs with diversified network structures that solve the same map can be evolved due to random mutation and crossover. For example, Figure 84 to 86 show three CPPNs that successfully address map #5. These networks possess five to ten hidden nodes, and their input nodes are connected very differently. No obvious similarity can be observed among them.



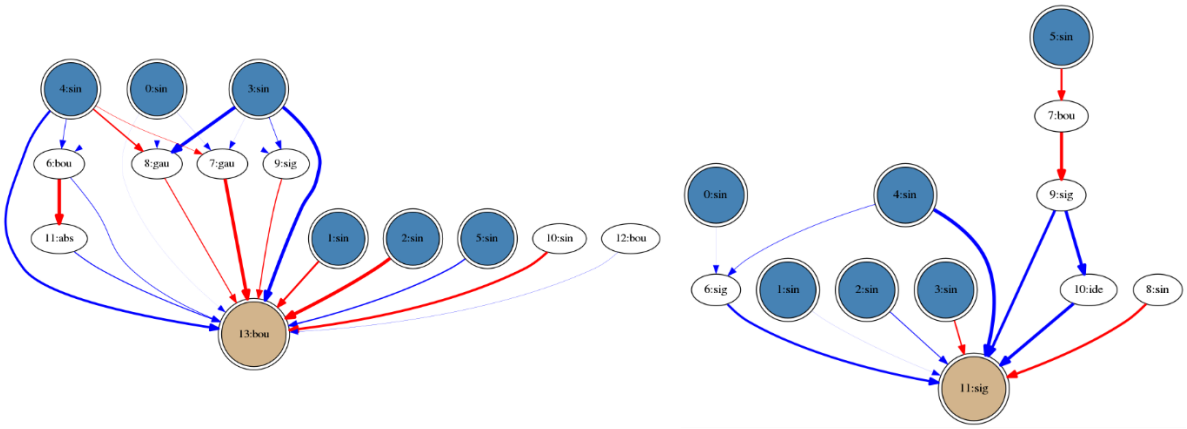


Figure 84, 85, 86: Three CPPNs that fully solve map #5

Different CPPN structures provide different output for ANN weights, and in this way, Hyper-NEAT is able to present a wide range of diversified solutions.

5.2.5. DQN is more sensitive to map difficulty

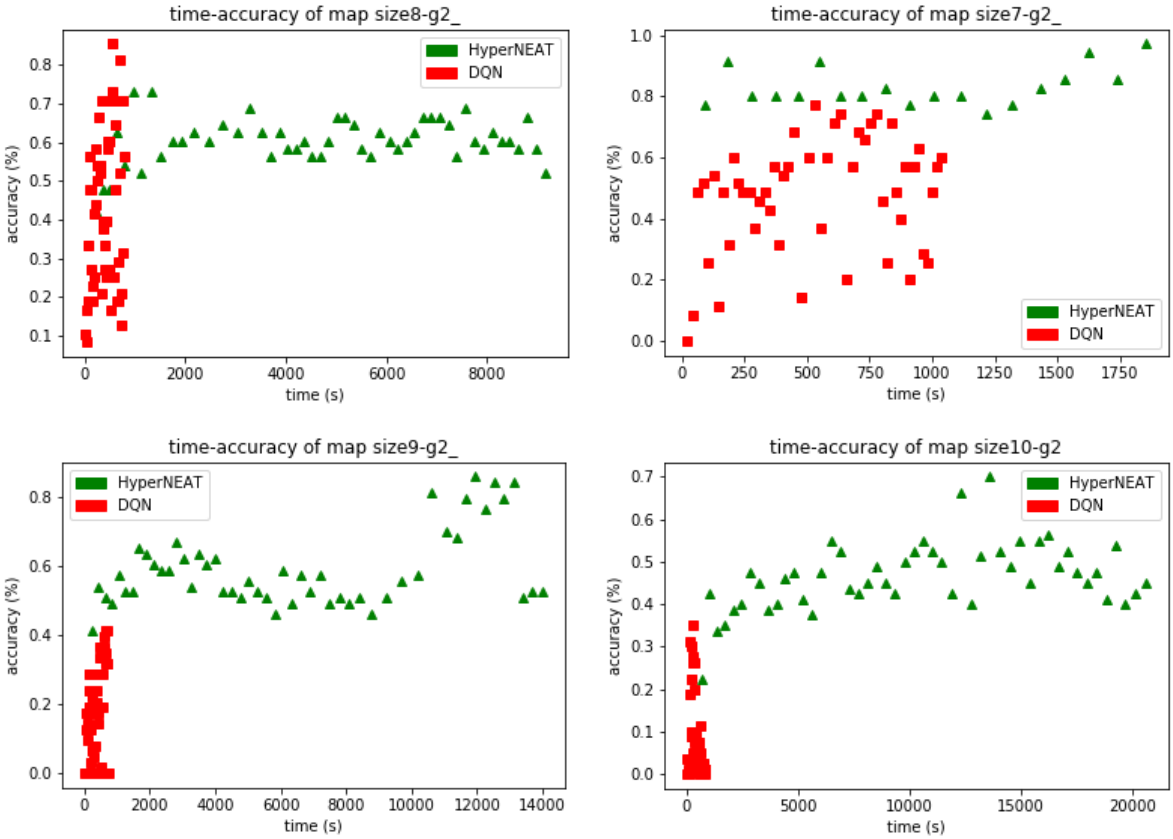


Figure 87, 88, 89: Accuracy-time plot from performances in map #17, #20, #23, #26 (left to right, top to bottom)

According to the pattern-based win rate graph (Figure.58.), maps of pattern 7 is the only maps in which DQN has been outperformed. Looking deeper at Figure 87-89 that show the

accuracy-time plotting over all the epochs in maps of pattern 7 and size 7, 8, 9, and 10, DQN's win rates in map #17, #23 and #26 are all remaining below the average of Hyper-NEAT. Table 11 suggests that Hyper-NEAT wins this pattern not only because DQN is bad at it, also because itself is good at it, since its win rate for this pattern is above the average, while DQN below the average. It is necessary to look closer at the pattern.

```
[[u'P' u'W' u' ' u' ' u' ' u'W' u' ' u' ' u' ' u'W']
 [u' ' u'W' u' ' u' ' u' ' u'W' u' ' u' ' u' ' u'W']
 [u' ' u'W' u' ' u'-' u' ' u'W' u' ' u'-' u' ' u'W']
 [u' ' u'W' u' ' u'W' u' ' u'W' u' ' u'W' u' ' u'W']
 [u' ' u'W' u' ' u'W' u' ' u'W' u' ' u'W' u' ' u'W']
 [u' ' u'W' u' ' u'W' u' ' u'W' u' ' u'W' u' ' u'W']
 [u' ' u'W' u' ' u'W' u' ' u'W' u' ' u'W' u' ' u'W']
 [u' ' u'-' u' ' u'W' u' ' u'-' u' ' u'W' u' ' u'-']
 [u' ' u' ' u' ' u'W' u' ' u' ' u' ' u'W' u' ' u' ']
 [u' ' u' ' u' ' u'W' u' ' u' ' u' ' u'W' u' ' u'+']]
```

Figure 90: Pattern for map #26

Figure 90 show the dispGrid function query for map #26, in which 'W' indicates wall, 'P' the agent, '-' the pit and '+' the goal. The pattern consists of long stripes of aisle that the agent needs to get through to reach the goal. The problem with reinforcement learning for pattern like this is that the agent has to make the right action consecutively for tens of steps, which, if happens at the beginning of training, with 100% exploration rate, is of possibility $(\frac{1}{4})^{55}$ (if initialized far from goal), an event with extremely low possibility. Otherwise, the agent gets stuck and the network is trained with bad-quality experience, which leads to network overfitting with high loss function.

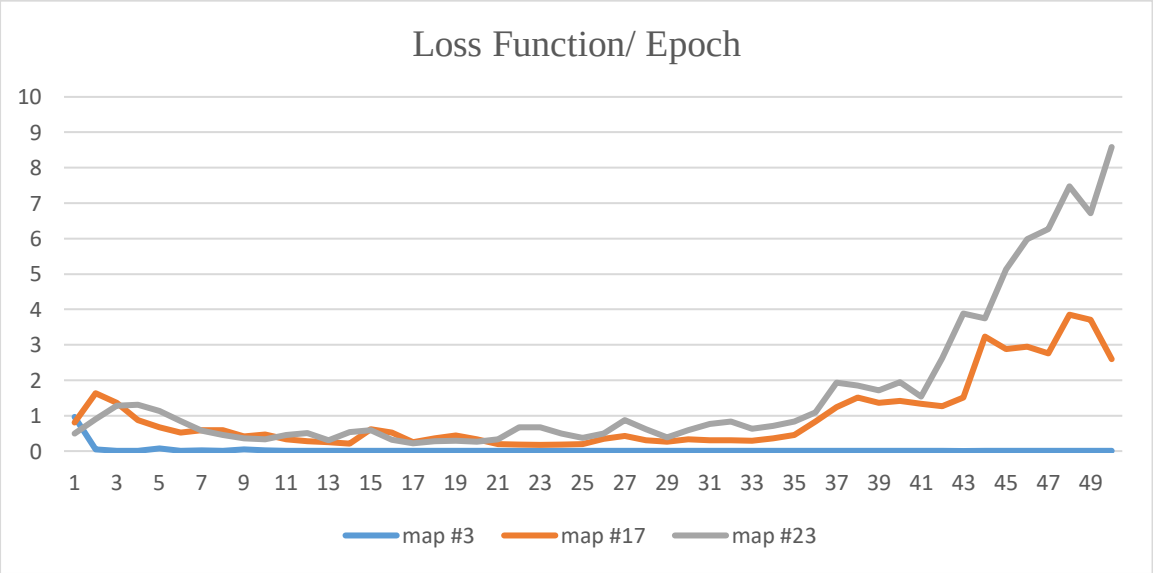


Figure 91: Loss function progression for map #3, 17 and 23

As a matter of fact, the loss function for pattern 7 indeed reaches the highest ever, around 1300 (Figure 92). With map #3 as baseline, Figure 91 shows the loss progression for the other two maps. Although the climbing is not as steep as map #26, they are the only loss values that get increased along the way.

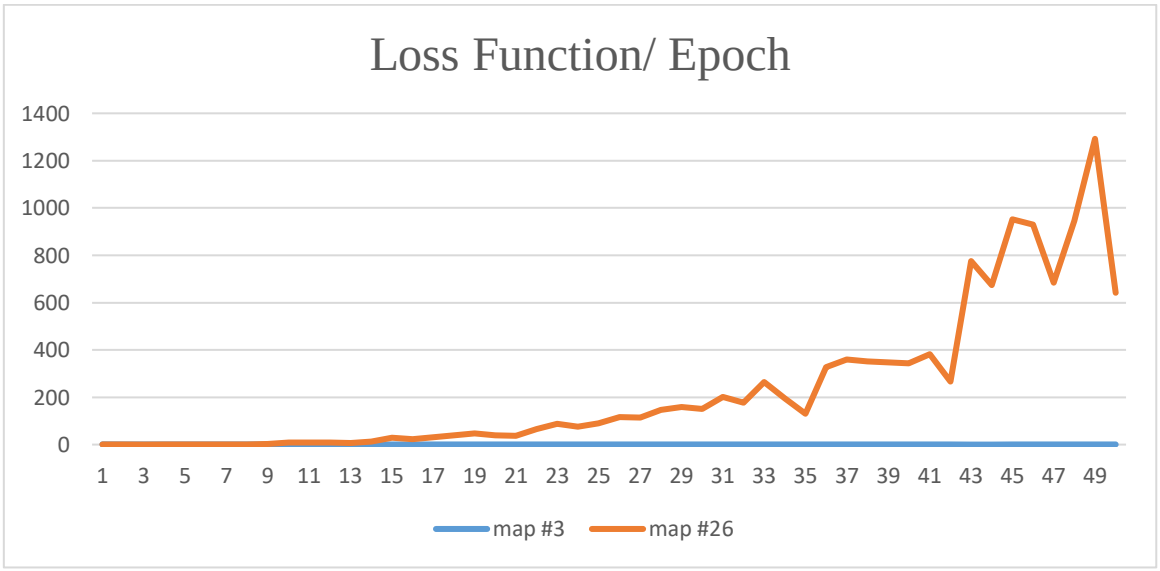


Figure 92: Loss function progression for map #3 and 26

Table 14: Q map for 7x7 pattern 7 iteration 50

↓	Wall	←	←	←	Wall	←
↓	Wall	←	→	←	Wall	←
←	Wall	←	Pit	→	Wall	←
←	Wall	←	Wall	←	Wall	←
←	Pit	↑	Wall	←	Pit	→
←	←	←	Wall	←	←	←
←	←	←	Wall	←	←	Goal

Table 15: Q map for 7x7 pattern 7 iteration 7

↓	Wall	→	→	→	Wall	↓
→	Wall	→	→	→	Wall	↓
→	Wall	→	Pit	↓	Wall	↓
→	Wall	→	Wall	→	Wall	↓
↓	Pit	→	Wall	↓	Pit	→
→	→	→	Wall	→	→	↓
→	→	→	Wall	→	→	Goal

Table 15 shows the Q map for map #17 at iteration 50 (Q map is the action policy for the agent). Apparently, the algorithm was very clueless at this point. However, if trace back to iteration 7 (table 14), the action policy is surprisingly more workable than iteration 50, proving that the network was indeed fed with bad experiences. As exploration rate dropped along the training process, the chance of self-improvement is even lower. Appendix shows the first 10 epochs of Q map, from which one can observe that the Q map quality fluctuates quite intensely

– it may be very close to the optimal at one epoch, and suddenly becomes absolutely wrong at the next. In other words, reinforcement learning quality is heavily dependent on the quality of field experience. However, this type of problem will never occur to Hyper-NEAT, because the development of networks is not built on field trips, but on the evaluation of networks. After the birth of networks, they are immediately put to testing, and the next generation of networks is optimised based on fitness values. Judging from results, Hyper-NEAT methodology is much more ideal for problems similar to pattern 7.

Chapter 6

6. Conclusion

One of the EA this project is concerned with, NEAT, is unable to perform a reinforcement learning task in a stochastic game environment. Its performance is similar to a random play. Hyper-NEAT evolves the function between objects and their locations with geometric awareness that successfully solves grid world tasks, however with less ideal efficiency and stability. DQN presents extraordinary efficiency in most maps, but fails if the exploration space is embedded with too many traps that terminate the game, this may cause the agent to get stuck at a premature state and never be able to move towards the goal state that returns positive rewards. The network topologies returned by Hyper-NEAT can be diversified in nature.

In summary, DQN is good at solving task of less tricky exploration space efficiently and stably. However, where DQN fails, Hyper-NEAT steps in. Hyper-NEAT ignores how far off the goal state is from where it currently is, hence can render promising performance. In terms of network structure, DQN generally returns networks in lighter weight, but Hyper-NEAT generates diversified solutions for the task.

6.1. Improvement of the work

This project accomplishes the promises in abstract, having carried out a basic comparative research, discovered the strengths and weaknesses for both algorithms and attempted to explain the reason behind with data collected. Apparently, there is still space for improvement, and this section lists some of the things that could have been done to improve the quality and validity of the deliveries if more time is at hand.

6.1.1. Larger parameter values

This project executes with 50 epochs and 100 training steps/ populations, which may very likely to be the lowest possible setting to observe and predict the progression trend. The results can be much more revealing if these values are set larger. With longer training/ evolution, some performances may achieve higher accuracy, and this is good for testing the algorithm's limits. Furthermore, larger population size will push up evolutionary algorithm's performance

proportionally, and bigger test number renders more accurate evaluations. However, increasing test number even by 1 results in considerably longer experiment time.

6.1.2. Richer Map Design

This project draws 27 maps with 8 pattern designs, with luck, one of the patterns is Hyper-NEAT's strong-suit. If this pattern was left out, the conclusion would be that Hyper-NEAT is totally useless, which is not true. Therefore, it is imaginable that due to the shortage of pattern supply, the current conclusion could be biased in some way. If this project is able to follow a comprehensive framework of pattern design, its conclusions would be better-supported.

6.1.3. To train a General Solution for Grid World

For now, the trained/evolved solutions are map-specific, however, if with enough time to carry out general solution training/evolution, the results will be more revealing.

6.2. Recommendation for continuations

If to continue this work, improvements above is recommended. In addition, Hyper-NEAT can be implemented in Atari environment to complete the comparison for processing efficiency on high-volume input data.

References

[1] T. Matiisen, "Demystifying Deep Reinforcement Learning", *nervana*, 2015. [Online]. Available: http://demystifying deep reinforcement learning. [Accessed: 13- Mar- 2017].

[2] D. Moriarty, A. Schultz and J. Grefenstette, "Evolutionary Algorithm for Reinforcement Learning", *Journal of Artificial Intelligence Research*, vol. 11, no. 1, pp. 241-276, 1999.

[3] M. Taylor, S. Whiteson and P. Stone, "Comparing evolutionary and temporal difference methods in a reinforcement learning domain", in *the 8th annual conference on Genetic and evolutionary computation*, Seattle, Washington, USA, 2006, pp. 1321-1328.

[4] M. Hausknecht, J. Lehman, R. Miikkulainen and P. Stone, "A Neuroevolution Approach to General Atari Game Playing", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 355-366, 2014.

[5] "Evolutionary computation", *En.wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Evolutionary_computation. [Accessed: 10- Mar- 2017].

[6] "What are the differences between genetic algorithms and genetic programming?", *Stackoverflow.com*, 2016. [Online]. Available: http://stackoverflow.com/questions/3819977/what-are-the-differences-between-genetic-algorithms-and-genetic-programming. [Accessed: 09- Mar- 2017].

[7] "Genetic Algorithms Tutorial", *www.tutorialspoint.com*, 2017. [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/index.htm. [Accessed: 22- Feb- 2017].

[8] D. Shiffman, S. Fry and Z. Marsh, *The Nature of Code*, 1st ed. California, US: D.shiffman, 2012, pp. 390-394.

[9] M. Potter and K. De Jong, "Evolving neural networks with collaborative species", in *Summer Computer Simulation Conference*, Washington, 1995, pp. 340-345.

[10] B. Zhang and H. Muhlenbein, "Evolving Optimal Neural Networks Using Genetic Algorithms with Occam's Razor", *Complex Systems*, vol. 7, no. 3, pp. 199-220, 1993.

[11] K. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies", *Evolutionary Computation*, vol. 10, no. 2, pp. 99-127, 2002.

[12] K. Stanley and R. Miikkulainen, "Efficient Evolution of Neural Network Topologies", in *the 4th Annual Conference on Genetic and Evolutionary Computation*, New York City, 2002, pp. 569-577.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529-533, 2015.

[14] M. Wittkamp, L. Barone and P. Hingston, "Using NEAT for Continuous Adaptation and Teamwork Formation in Pacman", in *2008 IEEE Symposium on Computational Intelligence and Games*, The University of Western Australia, Perth, 2008, pp. 234-242.

[15] E. Grant and B. Zhang, "A Neural-net Approach to Supervised Learning of Pole Balancing," *Proceedings. IEEE International Symposium on Intelligent Control 1989*, Albany, NY, 1989, pp. 123-129.

[16] J. Gauci and K. Stanley, "Autonomous Evolution of Topographic Regularities in Artificial Neural Networks", *Neural Computation*, vol. 22, no. 7, pp. 1860-1898, 2010.

[17] R. Sutton and A. Barto, *Reinforcement learning: An Introduction*, 1st ed. Cambridge, Mass. [u.a.]: MIT Press, 1998.

[18] S. Whiteson, "Evolutionary Computation for Reinforcement Learning", in *Reinforcement Learning*, 1st ed., M. Wiering and M. van Otterlo, Ed. Berlin, Heidelberg: Springer, 2012, pp. 325-355.

[19] D. Silver, "Teaching", *Www0.cs.ucl.ac.uk*, 2017. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html. [Accessed: 11- Mar- 2017].

[20] K. Stanley, "Compositional pattern producing networks: A novel abstraction of development", *Genetic Programming and Evolvable Machines*, vol. 8, no. 2, pp. 131-162, 2007.

[21] K. Stanley, D. D'Ambrosio and J. Gauci, "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks", *Artificial Life*, vol. 15, no. 2, pp. 185-212, 2009.

[22] M. Bellemare, Y. Naddaf, J. Veness and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents", *Journal of Artificial Intelligence Research*, vol. 47, pp. 253-279, 2013.

[23] M. Hausknecht, P. Khandelwal, R. Mikkulainen and P. Stone, "HyperNEAT-GGP: a hyperNEAT-based atari general game player", in *GECCO '12 Proceedings of the 14th annual conference on Genetic and evolutionary computation*, Philadelphia, Pennsylvania, USA, 2012, pp. 217-224.

[24] "Δ ℚuantitative √ourney | Q-learning with Neural Networks", *Outlace.com*, 2017. [Online]. Available: http://outlace.com/rlpart3.html. [Accessed: 11- Apr- 2017].

[25] "How Deep Neural Networks Work", *YouTube*, 2017. [Online]. Available: https://www.youtube.com/watch?v=ILsA4nyG7I0&t=1120s. [Accessed: 04- May- 2017].

[26] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436-444, 2015.

Appendix A Map Code

#1	#2	#3	#4	#5
4x4 pattern 1	4x4 pattern 2	4x4 pattern 3	4x4 pattern 4	4x4 pattern 5
#6	#7	#8	#9	#10
5x5 pattern 1	5x5 pattern 2	5x5 pattern 3	5x5 pattern 4	5x5 pattern 5
#11	#12	#13	#14	#15
6x6 pattern 1	6x6 pattern 2	6x6 pattern 3	6x6 pattern 4	6x6 pattern 5

#16	#17	#18
7x7 pattern 6	7x7 pattern 7	7x7 pattern 8
#19	#20	#21
8x8 pattern 6	8x8 pattern 7	8x8 pattern 8
#22	#23	#24
9x9 pattern 6	9x9 pattern 7	9x9 pattern 8
#25	#26	#27
10x10 pattern 6	10x10 pattern 7	10x10 pattern 8

Appendix B Parameter Setting

NEAT gene:

inputs=2,
outputs=1,
types=['tanh'],
topology=None,
feedforward=True,
max_depth=None,
max_nodes=inf,
response_default=4.924273,
initial_weight_stdev=2.0,
bias_as_node=False,
prob_add_node=0.03,
prob_add_conn=0.3,
prob_mutate_weight=0.8,
prob_reset_weight=0.1,
prob_reenable_conn=0.01,
prob_disable_conn=0.01,
prob_reenable_parent=0.25,
prob_mutate_bias=0.2,
prob_mutate_response=0.0,
prob_mutate_type=0.2,
stdev_mutate_weight=1.5,
stdev_mutate_bias=0.5,
stdev_mutate_response=0.5,
weight_range=(-50., 50.),
distance_excess=1.0,
distance_disjoint=1.0,
distance_weight=0.4.

NEAT population:

compatibility_threshold=3.0,

compatibility_threshold_delta=0.4,

target_species=12,

min_elitism_size=5,

young_age=10,

young_multiplier=1.2,

stagnation_age=15,

old_age=30,

old_multiplier=0.2,

reset_innovations=False,

survival=0.2.

DQN parameters

Environment:

frame_skip = 4

screen_width =84

screen_height =84

replay_size =1000000

history_length =4

learning_rate =0.00025

discount_rate =0.99

batch_size =32

optimizer =rmsprop

decay_rate =0.95

clip_error =1

min_reward =-1

max_reward =1

bufferSize=512

Agent:

exploration_rate_start =1

exploration_rate_end =0.1

exploration_decay_steps =1000000

train_frequency =4

train_repeat =1

target_steps =10000

random_starts =30

train_steps =250000

test_steps =125000

epochs =200

Appendix C Q map

Table 14: Q map for 7x7 pattern 7 iteration 1

↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	Pit	↓	Wall	↓
↓	Wall	↓	Wall	↓	Wall	↓
↓	Pit	↓	Wall	↓	Pit	↓
↓	↓	↓	Wall	↓	↓	↓
↓	↓	↓	Wall	↓	↓	Goal

Table 15: Q map for 7x7 pattern 7 iteration 2

↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	→	↓	Wall	↓
↓	Wall	↓	Pit	↓	Wall	→
↓	Wall	↓	Wall	↓	Wall	↓
↓	Pit	→	Wall	↓	Pit	↓
→	↓	→	Wall	↓	↓	↓
→	↓	→	Wall	↓	→	Goal

Table 14: Q map for 7x7 pattern 7 iteration 3

↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	Pit	↓	Wall	↓
↓	Wall	↓	Wall	↓	Wall	↓
↓	Pit	↓	Wall	↓	Pit	↓
↓	↓	↓	Wall	↓	↓	↓
↓	↓	↓	Wall	↓	↓	Goal

Table 14: Q map for 7x7 pattern 7 iteration 4

↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	Pit	↓	Wall	↓
↓	Wall	↓	Wall	↓	Wall	↓
↓	Pit	↓	Wall	↓	Pit	↓
↓	↓	↓	Wall	↓	↓	↓
↓	↓	↓	Wall	↓	↓	Goal

Table 14: Q map for 7x7 pattern 7 iteration 5

↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	Pit	↓	Wall	↓
↓	Wall	↓	Wall	↓	Wall	↓
↓	Pit	↓	Wall	↓	Pit	↓
↓	↓	↓	Wall	↓	↓	↓
↓	↓	↓	Wall	↓	↓	Goal

Table 14: Q map for 7x7 pattern 7 iteration 6

↑	Wall	←	→	←	Wall	↓
←	Wall	←	←	←	Wall	↓
←	Wall	←	Pit	←	Wall	↓
←	Wall	←	Wall	→	Wall	↓
←	Pit	←	Wall	↓	Pit	↓
←	←	←	Wall	←	↓	↓
←	←	←	Wall	←	→	Goal

Table 14: Q map for 7x7 pattern 7 iteration 7

↓	Wall	↓	↓	↓	Wall	↓
↓	Wall	↓	↓	↓	Wall	↓
←	Wall	←	Pit	↓	Wall	↓
←	Wall	←	Wall	↓	Wall	↓
←	Pit	←	Wall	↓	Pit	↓
←	←	←	Wall	→	↓	↓
←	←	←	Wall	→	→	Goal

Table 14: Q map for 7x7 pattern 7 iteration 8

→	Wall	→	→	→	Wall	↓
→	Wall	→	→	→	Wall	↓
→	Wall	→	Pit	↓	Wall	↓
→	Wall	→	Wall	→	Wall	↓
↓	Pit	→	Wall	↓	Pit	→
→	→	→	Wall	→	→	↓
→	→	→	Wall	→	→	Goal

Table 14: Q map for 7x7 pattern 7 iteration 9

→	Wall	→	→	↓	Wall	↓
→	Wall	→	→	↓	Wall	↓
←	Wall	←	Pit	↓	Wall	↓
←	Wall	←	Wall	↓	Wall	↓
←	Pit	→	Wall	↓	Pit	↓
←	←	←	Wall	→	→	↓
←	←	←	Wall	→	→	Goal

Table 14: Q map for 7x7 pattern 7 iteration 10

→	Wall	→	→	↓	Wall	↓
→	Wall	→	→	↓	Wall	↓
→	Wall	→	Pit	↓	Wall	↓
→	Wall	→	Wall	↓	Wall	↓
↓	Pit	→	Wall	↓	Pit	↓
→	→	→	Wall	→	→	↓
→	→	→	Wall	→	→	Goal