Date of acceptance Grade

Instructor

Applying Reinforcement Learning for Atari Games using NEAT Approach

Haibo Jin

Helsinki May 2, 2014

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author

Haibo Jin

Työn nimi — Arbetets titel — Title

Applying Reinforcement Learning for Atari Games using NEAT Approach

Oppiaine — Läroämne — Subject

Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	May 2, 2014	14 pages + 0 appendices

Tiivistelmä — Referat — Abstract

The paper reviews the idea of applying neuro-evolution for solving reinforcement learning problem, with an emphasis on NEAT approach. An experiment is also implemented by building an AI agent for Atari video games with the NEAT approach. The experimental result and its comparison with other methods implies the effectiveness and reliability of NEAT for solving such a reinforcement learning problem. Finally, some features of NEAT are discussed based on the experimental result.

Avainsanat — Nyckelord — Keywords

NEAT, neuro-evolution, reinforcement learning, general video games playing

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

With the development of modern computer games, artificial intelligence becomes an indispensable part of games. An intelligent and human-like game AI can always increase the fun of the game. On the other hand, an easy game environment can be seen as an abstraction of the real world. Thus, a learning algorithm of AI can be firstly tested and modified in the virtual environment, then applied to the problems of the real world [FaT09].

The Atari 2600 is a video game console which is considered as a successful entertainment device. It consists of 418 original games and supports a second player in many of them. Furthermore, the Atari 2600 has a simple interface for building learning agents. The state of the game can be directly described by its 2D grahpics (a native resolution of 160×210, which makes it complex enough to simulate the real-world). The action space consists of 18 different possible actions (the combination of eight directions of movement and a single button), of which the size is just moderate. Even though the Atari 2600 is no longer popular as a video game, its diversity and extensibility make it adequate for the research of learning agent. The Arcade Learning Environment (ALE) is a free and open-source emulator of Atari 2600 games, which is based on a simple object-oriented framework [BNV12]. The ALE allows researchers to build AI agents for Atari 2600 games without focusing on the details of simulation. Thus, we will use it as the learning environment in the project of this paper.

Several AI agents have been built upon the ALE, using different algorithms, such as SARSA(λ) [BNV12, Nad10] and planning [BNV12, Nad10]. SARSA(λ) is a model-free reinforcement learning approach while planning is a search-based method which does not involve learning.

Though the SARSA(λ) and search-based methods can be applied to build a domain-independent game agent, which can be used over a number of Atari games rather than just one specific game, they still need prior knowledge, especially in the feature construction part. Even when using screen pixels as input, it still requires some preliminary work (e.g. removing the static background and getting the moving targets), which is neither convenient nor general enough for most games. Therefore, we will present a more generic algorithm for building an AI agent in the ALE, and evaluate the performance of the algorithm through an experiment. The algorithm we use in the experiment is based on Neuro-Evolution of Augmenting Topologies

(NEAT), first proposed by Stanley and Miikkulainen in year 2002 [STM02]. NEAT is a neuro-evolution algorithm, evolving topologies of Artificial Neural Networks (ANNs) along with its weights. Different from the algorithms that are based on value function space, neuro-evolution is a reinforcement learning approach that searches policy space by using evolutionary algorithms. With slight modification, NEAT can be applied for developing a general video games playing agent in Atari games, which is able to search in a larger space but with more general inputs.

Hausknecht et al. first applied NEAT and HyperNEAT for general video games playing in the ALE in 2013 [HLM13]. HyperNEAT is a variant of NEAT [SDG09], which uses indirect encoding and performs better than NEAT on high dimensional inputs (e.g. screen pixels) [HLM13]. Even though we use the same NEAT as that of Hausknecht et al. in the experiment, we have different representations of inputs. In other words, Hausknecht et al. used an object based representation as inputs (by detecting the moving objects in the screen) while we use the down sampled screen pixels. More recently, an Atari game playing agent based on deep Q-learning was published, which was also capable of general video games playing [MKS13].

The rest of the article is organized as follows: The basics of reinforcement learning and its two function searching approaches are illustrated in Section 2. In section 3, details of neuro-evolution are discussed as well as two different neuro-evolution based algorithms, namely CNE (Conventional Neuro Evolution), and NEAT. The implementation of AI agent for Atari games is presented in Section 4. Section 5 gives the experimental results of the paper and Section 6 gives the conclusion.

2 Two Approaches for Reinforcement Learning

To solve reinforcement learning problems, there are two main approaches for searching functions: one is to search in value function space and the other is to search in policy space [GMS11]. Among them, temporal difference is a typical method of searching in value function space (though it can also be used in policy space) while evolutionary algorithms representative for the latter [GMS11]. In this section, we will present how temporal difference and evolutionary algorithms can be applied for reinforcement learning respectively, as well as the key difference of the two methods.

2.1 Reinforcement Learning

Reinforcement learning intends to solve sequential decision tasks by conducting iterative interactions with the environment [GRS90]. In machine learning, such a process is typically formulated as Markov decision process. In order to optimize the reward function, an agent will choose the most appropriate action to change its states by observing the current states it received. As a more formal definition, at a time step t, an agent receives a current state s_t and selects an action a_t, then transfers to state s_{t+1} [GMS11].

In reinforcement learning, the goal of the agent is to finally learn a policy π that maps every state s in state set S to a certain action a in action set A. We usually want the policy to be the optimal policy π^*, which is typically defined in such a way that can get the greatest cumulative reward over all the states:

$$\pi^* = argmax_\pi V^\pi(s), (\forall s) \tag{1}$$

in which $V^\pi(s)$ represents the cumulative reward it gets from state s with policy π. To calculate $V^\pi(s)$, there are many ways, one of which is to use a discount rate γ to reduce subsequent rewards over time and then sum it over:

$$V^\pi(s) = \Sigma_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2}$$

where r_{t+i} represents the immediate reward at a certain time step $t+i$, in which i varies from zero to infinity.

While an agent can also be trained by supervised learning, the learning examples are always presented as state-action pairs in this case, associated with a label of "correct" or "incorrect". Because of this, supervised learning can be infeasible when there are not many labels from examples. As for reinforcement learning, it associates the subsequent rewards so that it can be applied to situations where rewards are sparse. Because of the uncertainty of environment, an agent of reinforcement learning needs to gain its knowledge of the environment by exploring. When applying reinforcement learning, one thing to note is that the balance should be made between exploration (of unknown environment) and exploitation (of current knowledge).

2.2 Temporal Difference for Reinforcement Learning

As stated before, temporal difference (TD) is the most common approach for calculating the value function V for reinforcement learning problems. It uses the difference of two consecutive observations to update value functions, where the updating rule can be illustrated as following:

$$V(s_t) = V(s_t) + \alpha(V(s_{t+1}) - V(s_t) + r_t) \tag{3}$$

in which α represents the learning rate and r_t the immediate reward. Thus, following such an updating rule, the values can adjust themselves to agree with their successors and finally agree with the value $V(s_n)$. In other words, when a value $V(s_n)$ receives a reward, the reward will be propagated backwards through the chain of the value [GMS11].

2.3 Evolutionary Algorithm for Reinforcement Learning

Evolutionary algorithm (EA) is an optimization method derived from Darwin's evolution theory, similar to the process of natural selection [GRS90]. EA searches the optimal solution by iteratively updating a population of potential solutions, which are often called genotypes or organisms. During every iteration, new offsprings will be produced based on the previous generation, using some genetic operators, typically known as crossover and mutation. Some producing rules should also be considered, such as the population of a generation, the crossover rate and the mutation rate, etc. A typical EA process is showed in Figure 1.

Even though EA is a general optimization method and can be applied in a variety of domains, the design and specialization of EA is still important when utilized in other problems. One critical design that needs to be considered is the representation, which decides how phenotypes are encoded into genotypes. It has been showed in many studies that the choice of representations affect the effectiveness of EA in a sensitive way.

Although there are a number of choices to represent policies when using EA in reinforcement learning problem, the methods can be categorized into two types: one is to use bit string as representation, and the other evolves neural networks.

Here is an example for illustrating the difference of the two representations. Consider a simple game system with three inputs representing current state and one output

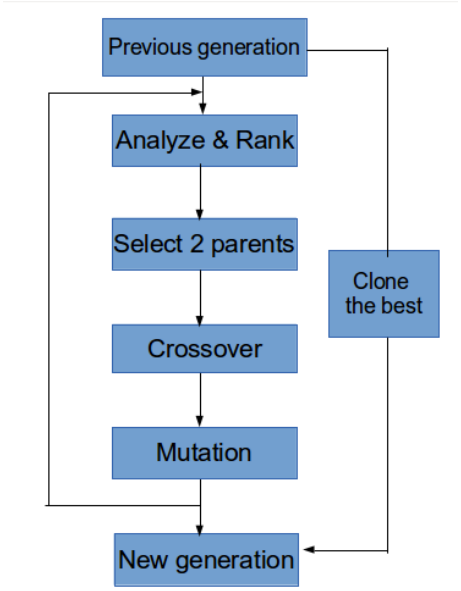

Figure 1: Typical process of evolutionary algorithm

as an action. All the inputs and output are binary values, taking either 0 or 1. Figure 2 shows a bit string representation of the simple system and Figure 3 shows an ANN-based representation. Both of the representations shows such a scenario that for a state featured by 110, the action should be 1.

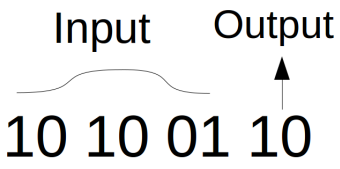

Figure 2: Representation of evolutionary algorithms in bit string form. Every two bits represents a binary value, 10 for value 1 and 01 for value 0. 11 means the value does not matter. The first three values of the bit string are 1, 1, and 0, which means the input is a feature with value 110. The fourth value is 1, representing the choice of action.

3 Neuro-evolution

Evolving artificial neural network with evolutionary algorithms is also known as neuro-evolution. One distinct feature of neuro-evolution algorithms is the ability to

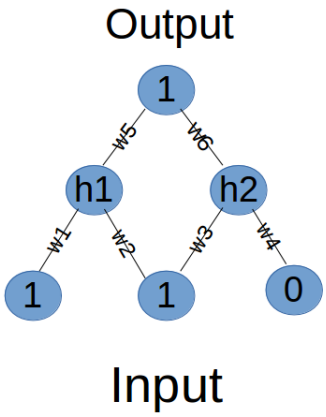

Figure 3: Neural network representation of evolutionary algorithms. The output of the network depends on the value of input and the weights of the network. For this network, when input is 1, 1, and 0, the output is 1.

adapt to an environment even when there exist changes in the environment [Yao99]. In neuro-evolution, there are mainly two ways of evolution: one is to evolve weights and the other is to evolve the topology structure (the connectivity of the neural network) [Yao99].

3.1 CNE

Conventional Neuro-evolution (CNE) is the simplest case of neuro-evolution [HLM13]. The topology of its network is fixed manually and only the weights are evolved during the iteration. Usually, the fixed network has one layer of hidden nodes, fully connected with input nodes and output nodes. Thus, the goal of CNE is to optimize the weights of the network so that the inputs can be mapped to the outputs properly.

However, connection weights is not the only aspect that affects the functionality of neural network. The topology structure of neural network also contributes a lot to the functionality. Despite that a fully connected network can approximate any continuous function in theory, deciding on the topology of the network in advance can take much time of humans. Thus, it is reasonable to evolve both the weights and topologies during evolution, as long as the topologies are legitimately evolved and maintained. Actually, if done right, evolving both the weights and the topologies can improve the performance of neuro-evolution significantly [STM02].

3.2 NEAT

Neuro-Evolution of Augmenting Topologies (NEAT) can evolve both weights and topologies efficiently. Compared to CNE, NEAT is able to add new nodes into the network which gives NEAT the power of handling more complex problems [HLM13]. Additionally, there are three essential techniques used in NEAT that makes NEAT even more reliable and efficient: historical markings, speciation and minimal structure [STM02].

Through historical markings, NEAT can perform crossover in a reasonable way by tracking genes. A gene, also known as the connection between two neurons, is identified by a unique innovation number. Whenever there is a new gene appears (through the mutation operator), the innovation number will be increased and assigned to that gene. Thus, same genes will have same innovation numbers, which makes it easy to do crossover between two networks with different topology structures. Figure 4 shows two networks with different topologies. Figure 5 presents the details of how the two networks doing crossover as well as the structure of their offspring.

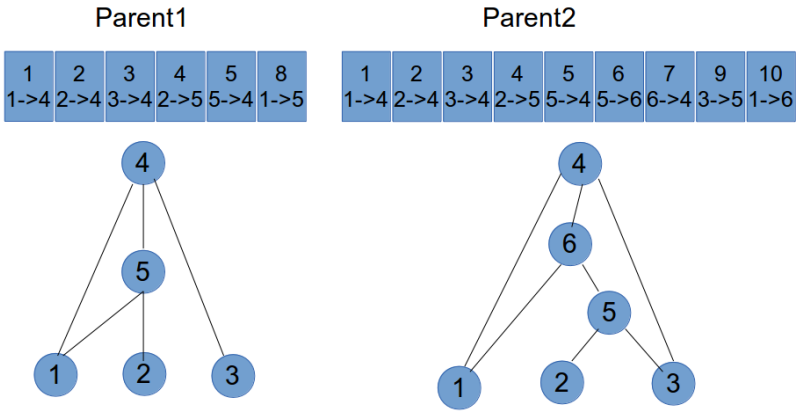

Figure 4: The two networks for crossover. The markings over the networks can be seen as a part of the genes, recording the topology of the network, namely the connections of nodes and their innovation numbers. As showed in the figure, same connections have the same innovation numbers.

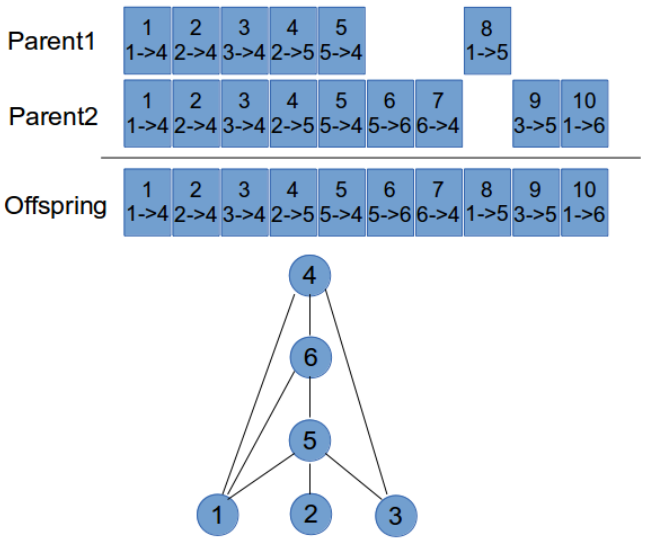

Figure 5: For the genes of the two parents, there must be some that can match up upon innovation numbers and some not. When doing crossover, for those match up genes, they will be randomly inherited by offspring from one of the parents. For those genes that do not match up, they will be inherited from one of the parents that owns better fitness score. In this case, the two parents have the same fitness, then both of their genes are inherited to the offspring.

NEAT can protect innovation organisms (an organism is a neural network, i.e. a candidate solution for AI agent) through speciation. The idea is to divide the generation into groups, in which a group can be seen as a species. The organisms of a species usually own similar structures so that every organism is competing with other similar organisms within the species instead of the whole generation. By doing this, some organisms with innovative structures that perform poorly can be protected. After several generations, these innovations may perform better or even the best.

A typical neuro-evolution method evolving both weights and topologies usually starts with an initial population of random topologies for the purpose of introducing diversity [STM02]. In contrast, NEAT starts with an initial population of minimal structures. More specifically, the initial structure only consists of input nodes and output nodes (maybe one bias node), without any hidden nodes. Since the structure starts and grows minimally, the dimensionality of the searching space can be reduced. Thus, NEAT evolves faster than the typical neuro-evolution methods.

4 Experiment

The goal of the experiment is to apply NEAT for building an AI agent of Atari games. The agent should be able to play several Atari games, which is known as general video games playing. However, due to the limitation of time and device, the experiment is simplified on some details and only tested on one game: space invaders. Space invaders is a shooting game that aims to defeat a wave of aliens with a laser cannon to get as many points as possible.

4.1 Architecture

The process of the agent can be divided into three stages: evaluation, speciation and reproduction. The basic workflow is presented in Figure 6.

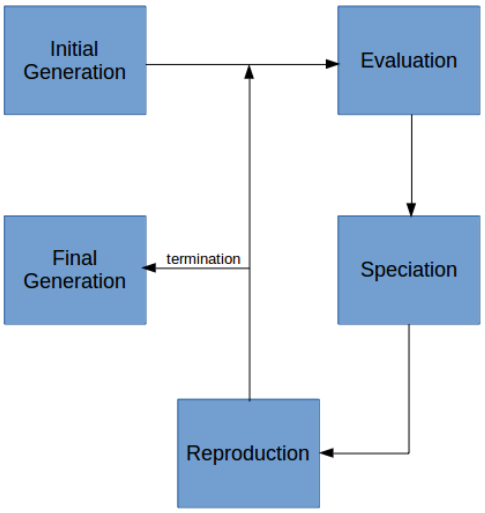

Figure 6: The agent starts from an initial generation. Then it enters into a loop which consists of three stages, namely evaluation, speciation and reproduction. When doing evaluation, every organism of the generation is evaluated by giving a fitness score. Then the generation will be split into several species based on a compatible testing rule. So the organisms with similar structures will be in the same species. For each species, it should calculate the amount of offspring to generate, then use the reproduction rules to get new organisms for the next loop.

4.1.1 Evaluation

For one organism, the evaluation is based on one episode. When one episode ends, the total score of the game represents the fitness of the corresponding organism.

In order to shorten the time of training, the feature extraction algorithm used on the screen shot is quite simple. The extraction is used on the lower half of the image, which is divided into 10×10 grids. The image is firstly transformed to a gray scale image so that every pixel can be represented as an integer (0 to 255). For each small grid, if its average pixel value is larger than the global average pixel value, it will be denoted as 1. Otherwise, it becomes 0. By doing this to all the 100 grids, the image can be transformed to a binary vector with length 100.

Since the length of input, a binary vector, is 100 and the total number of possible actions is 18, the neural network of each organism contains 100 input nodes, 1 bias node and 18 output nodes. The number of hidden nodes of the organisms are different, depending on the mutation of each network, which will be explained in section 4.1.3.

Figure 7 shows the process of one iteration about how an organism is evaluated.

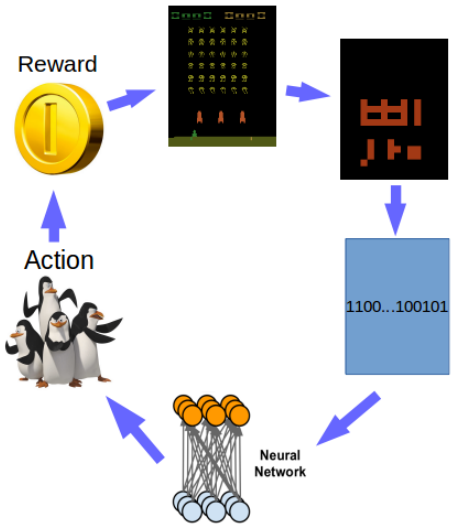

Figure 7: In each iteration, the screen shot of the current game state is firstly converted to a 10×10 binary grids. The grids are then transformed to a vector, as the input of the neural network. When the value of input is propagated to the output nodes, one of the 18 actions will be activated and executed. Then the laser cannon may die, or get a point, or nothing happens, all of which reflect the reward of the selected action.

4.1.2 Speciation

When doing speciation, for every species of the current generation (there is only one species initially), the best organism of a species is set to be the representative of the species. Then the rest others need to do a compatible testing with each representative one by one. Whenever a organism is compatible with a representative, the organism is added to the corresponding species. If none of the representatives is compatible, a new species will be built for the organism. Finally, all the organisms are divided into groups and the structures of networks are similar within the same group. The formula for compatible testing is listed as following:

$$\delta = c_1 D + c_2 W \tag{4}$$

where δ is the compatible distance of two organisms, D represents the number of mismatch genes and W represents the average weight differences of matching genes. c_1 and c_2 are coefficients that used to balance the weights of D and W.

In order to avoid any species taking over the entire population, explicit fitness sharing is used to solve such a problem. That is, the fitness of each organism is adjusted based on the species it belongs to. The adjusted fitness of organism O_i is calculated as follows:

$$f_i^{'} = \frac{f_i}{F_i} \tag{5}$$

where $f_i^{'}$ is adjusted fitness, f_i is original fitness and F_i is the sum of fitness of all the organisms that in the same species of O_i. With explicit fitness sharing, a species cannot become too big even many of its organisms have good performance [STM02]. Assume N is the population of the whole generation, and F is the sum of the adjusted fitness of all the organisms, then the amount of offspring that a species needs to generate is calculated as follows:

$$N_i = N \times \frac{F_i}{F} \tag{6}$$

4.1.3 Reproduction

Reproduction means to generate offspring based on the current organisms, through some genetic operators, such as crossover and mutation. In the experiment, the

reproduction is conducted within a species. For a certain species S_i, firstly, a proportion of S_i with worst performance will be eliminated. Then, the best ones will be cloned and directly added to the offspring. Additionally, some offspring come from crossover and some from mutation. Consequently, offspring with number of N_i will be generated.

The method of crossover is mentioned in section 3.2. As for mutation, there are three ways. One is to mutate the weights. For every weight of the network, there is p_w percent chance to mutate the weights, in which there is p_{w1} percent chance of uniformly perturb the weight and p_{w2} percent chance of randomly assign a new value. Another way is adding a new connection to the network with p_c percent chance and adding a new hidden node with p_n percent probability.

4.2 Parameter Settings

The population size and generation times are both set to 10, which is quite small for a neuro-evolution algorithm. This is for the purpose of saving training time, and it still works, as we can see in the next section. The coefficients of compatible testing is $c_1 = 1.0$ and $c_2 = 0.4$. The threshold of compatible testing, δ is 15. In reproduction, 20% of the organisms will be eliminated, then the best two are added to the next generation directly. 40% of the rest offspring are from crossover and the others come from mutation. In mutation, $p_w = 0.8$ for weights mutation, in which $p_{w1} = 0.9$ and $p_{w2} = 0.1$. The probability of adding a new link is $p_c = 0.15$ and $p_n = 0.1$ for adding a new node. The sigmoid function used in the neural network is $\varphi(x) = \frac{1}{1+e^{-4.9x}}$.

4.3 Result and Comparison

Based on five running times, the average score of one episode game of space invaders is 526, as listed in Table 1, in which the implementation of the experiment is denoted as pixel-based NEAT.

NEAT pixel	Random	SARSA object	NEAT object	Human
526	157	250	1481	42905

Table 1: The first item shows the average score of pixel-based NEAT. It also lists the scores of some other methods [HLM13].

We can also see the comparison between pixel-based NEAT and some other approaches. It is worth mentioning that the implementation of the pixel-based NEAT is simplified and not well tuned. Under such conditions, the result of pixel-based NEAT is obviously better than Random, which means the agent of pixel-based NEAT has learned to choose suitable policies. SARSA (State-Action-Reward-State-Action) is a widely used reinforcement learning algorithm that has been used in a large number of problems [Nad10]. Compared to SARSA, pixel-based NEAT still performs a better score significantly, which means the pixel-based NEAT is effective for building an AI agent of video games. The fourth item in Table 1 gives the score of an object-based NEAT implemented by Hausknecht et al. [HLM13]. In addition to some implementation details and training times, the difference between pixel-based and object-based NEAT is the feature representations. The pixel-based NEAT uses a feature of down sampled screen pixels as representation while the object-based NEAT uses the detected objects as features. The performance of object-based NEAT significantly surpasses pixel-based NEAT, which implies that a more precise features can improve the performance of the agent. As we can see, the best recorded human score is 42905. Unsurprisingly, there is still a big gap between reinforcement learning algorithms and humans.

5 Conclusion

In this paper, we review the basics of reinforcement learning and neuro-evolution. By applying a neuro-evolution method NEAT on building an AI agent of Atari games, we confirm the effectiveness of NEAT for solving reinforcement learning problem. Though it is effective, there are some issues need to consider when it is used: 1.The parameters of NEAT is sensitive, therefore, well-tuned parameters can significantly improve the performance of NEAT; 2.Due to the direct encoding of NEAT, a compact and extracted feature is more suitable for NEAT rather than a raw and redundant feature.

References

BNV12 Bellemare, M., Naddaf, Y., Veness, J. and Bowling, M., The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708.

FaT09 Fang, Y. and Ting, I., Applying reinforcement learning for game ai in a tank-battle game. *Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on*, Dec 2009, pages 1031–1034.

GMS11 Grefenstette, J., Moriarty, D. and Schultz, A., Evolutionary algorithms for reinforcement learning. *CoRR*, abs/1106.0221.

GRS90 Grefenstette, J., Ramsey, C. and Schultz, A., Learning sequential decision rules using simulation models and competition. *Machine Learning*, 1990, pages 355–381.

HLM13 Hausknecht, M., Lehman, J., Miikkulainen, R. and Stone, P., A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

MKS13 Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

Nad10 Naddaf, Y., *Game-Independent AI Agents for Playing Atari 2600 Console Games*. Masters, University of Alberta, 2010.

SDG09 Stanley, K., D'Ambrosio, D. and Gauci, J., A hypercube-based encoding for evolving large-scale neural networks. *Artif. Life*, 15,2(2009), pages 185–212.

STM02 Stanley, K. and Miikkulainen, R., Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10,2(2002), pages 99–127.

Yao99 Yao, X., Evolving artificial neural networks. *Proceedings of the IEEE*, 87,9(1999), pages 1423–1447.