Policy Iteration through evolutionary function approximation

Yunian Pan
Department of Electrical Engineering
yp1170@nyu.edu

Abstract

This report presents the substances of the paper "Evolutionary Function Approximation for Reinforcement Learning", along with some simulations using the related algorithm. In the context that temporal difference methods are theoretically grounded and empirically effective methods for addressing reinforcement learning problems, we have been moving towards a era of deep learning that in most real-world reinforcement learning tasks, TD methods require a function approximator to represent the value function. However, using function approximators requires manually making crucial representational decisions. The related papers investigates evolutionary function approximation, a novel approach to automatically selecting function approximator representations that enable efficient individual learning. The method, which naturally arises from genetic algorithm schemes, evolves individuals that are better able to learn.

Some implemented instantiations of evolutionary function approximation which combines NEAT, a neuroevolutionary optimization technique, with Q-learning, a popular TD method was presented in paper [1], The resulting NEAT+Q algorithm automatically discovers effective representations for neural network function approximators. The original paper also presents on-line evolutionary computation, which improves the on-line performance of evolutionary computation by borrowing selection mechanisms used in TD methods to choose individual actions and using them in evolutionary computation to select policies for evaluation.

Also some online searching scheme has been proposed in this paper based on epsilon greedy method or Boltzmann distribution, which improves the on-line performance of evolutionary computation by borrowing selection mechanisms used in TD methods to choose individual actions and using them in evolutionary computation to select policies for evaluation. Two empirical experiments were evaluated and discussed in this paper, it also presents additional tests that offer insight into what factors can make neural network function approximation difficult in practice.

1 Introduction

Reinforcement learning problems are the subset of machine learning tasks in which the agent never sees examples of correct behavior. Instead, it receives only positive and negative rewards for the actions it tries. Since many practical, real world problems (such as robot control, game playing, and system optimization) fall in this category, developing effective reinforcement learning algorithms is critical to the progress of artificial intelligence.

The most common approach to reinforcement learning relies on the concept of value functions, which indicate, for a particular policy, the long-term value of a given state or state-action pair. Temporal difference methods (TD(λ)) (Sutton, 1988), which combine principles of dynamic programming with statistical sampling, use the immediate rewards received by the agent to incrementally

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

improve both the agents policy and the estimated value function for that policy. Hence, TD methods enable an agent to learn during its lifetime i.e. from its individual experience interacting with the environment.

When the problem scale is not large, the value function can be represented as a table of state-action pairs, but for large scale systems or for problems that has infinite dimension of state space or action space, it remains problematic to only use a "table", instead the function approximator which represents the mapping from state-action pairs to values via a more concise, parameterized function and uses supervised learning methods to set its parameters, plays an important role where we can omit the step of enumering over the state-action space. Many different methods of function approximation have been used successfully, including CMACs, radial basis functions, and neural networks.

However, using function approximators requires making crucial representational decisions (e.g. the number of hidden units and ini- tial weights of a neural network. Poor design choices can result in estimates that diverge from the optimal value function and agents that perform poorly. Even for reinforcement learning algorithms with guaranteed convergence, achieving high performance in practice requires finding an appropriate representation for the function approximator.

As Lagoudakis and Parr observe,[1]] The crucial factor for a successful ap- proximate algorithm is the choice of the parametric approximation architecture(s) and the choice of the projection (parameter adjustment) method. (Lagoudakis and Parr, 2003, p. 1111) Nonetheless, representational choices are typically made manually, based only on the designers intuition.

Therefore, in order to automate the search for effective representations by employing sophisticated optimization techniques, we focus on using evolutionary methods because of their demonstrated ability to discover effective representations, so that our proposition can cover the 2 things: 1) an evolutionary algorithm capable of optimizing representations from a class of functions; 2) a TD method that uses elements of that class for function approximation.

Neural network is a good choice as hey have great experimental value, Nonlinear function approximators are often the most challenging to use, hence, success for evolutionary function approximation with neural networks is good reason to hope for success with linear methods too; and they have great potential in that they can represent value functions that the linear method cannot(given the same basis functions). Therefore NeuroEvolution of Augmenting Topologies(NEAT)[2]] has been chosen as the main framework. The resulting algorithm, called NEAT+Q, uses NEAT to evolve topologies and initial weights of neural networks that are better able to learn, via backpropagation, to represent the value estimates provided by Q-learning.

Section 2 will present the background including Q-learning and deep Q-learning, genetic algorithms and NEAT framework and NEAT algorithm for reinforcement learning and some basic related experiments; Section 3 will present the new emerging method that incorporates the previous algorithm together, along with the experiments and discussion. Section 4 will have some discussion over the results obtained from the experiments.

2 Background

This section mainly contains two parts, one is for the development of Q-learning, Deep Q learning, and some variance of them; The another will discuss the machinery and framework and using NEAT to do reinforcement learning, specifically, we will start from the general framework of genetic algorithm and with a little bit discussion on how it works, then introduce a gene encoding scheme that NEAT relies on.

2.1 Q-learning

As a model-free algorithm, Q-learning trains the agent what action to take under what circumstances, is a well-established, canonical method that has also enjoyed empirical success, In the tabular case, the algorithm is defined by the following update rule, applied each time the agent transitions from state s to state s':

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \min_{a'} Q(s',a')) \tag{1}$$

where $\alpha \in [0,1]$ is a learning rate parameter, $\gamma \in [0,1]$ is a discount factor, and r is the immediate reward the agent receives upon taking action a.

This equation was initially derived from Bellman equation regarding Markov Decision Process, recall the original operator of value function:

$$(TJ)(i) := \min_{u \epsilon U(i)} \sum_{j=1}^{n} p_{ij}(u)[g(i,u,j) + \alpha J(j)] \tag{2}$$

define: $Q(s,a) = \sum_{j=1}^{n} p_{ij}(u)[g(i,u,j) + \alpha J(j)]$, we can omit the probability transition, and through Robins-Monro algorithm we solve the Q-learning problem and end up with the update rule 1.

2.2 Q-Learning with NN

Particularly when combined with neural network function approximators, algorithm 1 describes the Q-learning algorithm using a neural network to approximate the value function. The inputs to the network describe the agents current state; the outputs, one for each action, represent the agents current estimate of the value of the associated state-action pairs. The initial weights of the network are drawn from a Gaussian distribution with mean 0 and standard deviation σ (line 5). The EVAL-NET function returns the activation on the networks outputs after the given inputs are fed to the network and propagated forward. Since the network uses a sigmoid activation function, these values will all be in $[0, 1]$ and hence are rescaled according to a parameter c. The parameter λ the sample roll-out cost decay, which is for the convenience of doing $TD(\lambda)$ extension, which we usually set to 0 or 1, respectively representing the value iteration method and the policy evaluation method.

Algorithm 1: Q-Learning

Input: S : set of all states; A : set of all actions; σ : standard deviation of initial weights; c : output scale; α : learning rate; γ : discount factor; λ : eligibility decay rate; ϵ_{td} : exploration rate; e : total number of episodes;

Initialize: $N \leftarrow$ INIT-NET(S, A, σ) ;

for $i \leftarrow 1$ *to* e **do**
  $s, s' \leftarrow$ null, INIT-STATE(S);
  while *Terminal-state?(s)* **do**
    $Q[] \leftarrow c\times$EVAL-NET(N, s');
    With-prob(ϵ_{td}) $a' \leftarrow$ RANDOM(A);
    else: $a' \leftarrow \arg\max Q[j]$;
    if $s \neq null$ **then**
      BACKPROP$(N, s, a, (r + \gamma \max_j Q[j])/c, \alpha, \gamma, \lambda)$;
    else
      $s, a \leftarrow s', a'$;
      $r, s' \leftarrow$ TAKE-ACTION(a')
    end
  end
end

2.3 NEAT

2.3.1 genetic algorithm

Genetic algorithm(GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms(EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.[3] Figure 1 describes a simple case of doing one iteration gene evolution, the binary codes in the left side represent the population of solutions to the objective problem, while the right side illustrates the process of doing crossover, the mutation step left is just randomly change the 1's into 0's. The convergence is guaranteed under the condition that we choose the best species at every generation.

A standard evolution framework can be summarized as:

(1) Initialize population

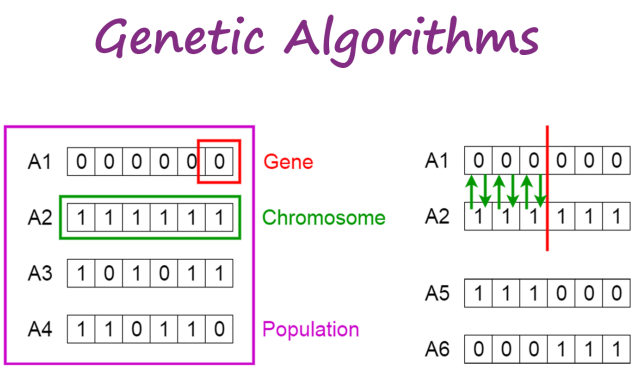

Figure 1: from Google Image

(2) Evolve from $1, \ldots, n_{generation}$:

a) Select from population according to fitness

b) Generate offspring through crossover and mutation

c) Replace population with offspring.

The flow chart 2 describes the scheme: Inspired by the framework, one might think about apply it to

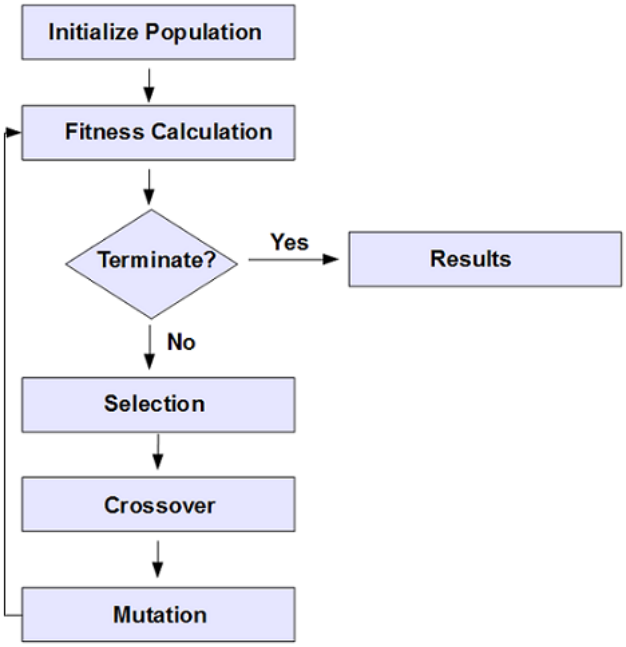

Figure 2: from Google Image

finding the best structure of a neural network, i.e. we initialize a population of neural networks, then proceed the nature selection based on the fitness of the networks and try to breed the next generation with the winners, iteratively increasing the fitness of each generation until the resulting population satisfies the performance criterion.

2.3.2 Another TWEANN encoding

In order to develop the neural evolution algorithm, we need a standard method to encode the neural networks as genotypes so that we can do crossover and mutation operation based on the genotypes provided. There are many evolutionary systems of neural networks while Topology and Weight Evolving Artificial Neural Networks(TWEANN) encoding is the best approach to address the question of how to encode networks using an efficient genetic representation. Absorbing the properties of many TWEANN systems, NEAT encodes the neural networks in following ways:

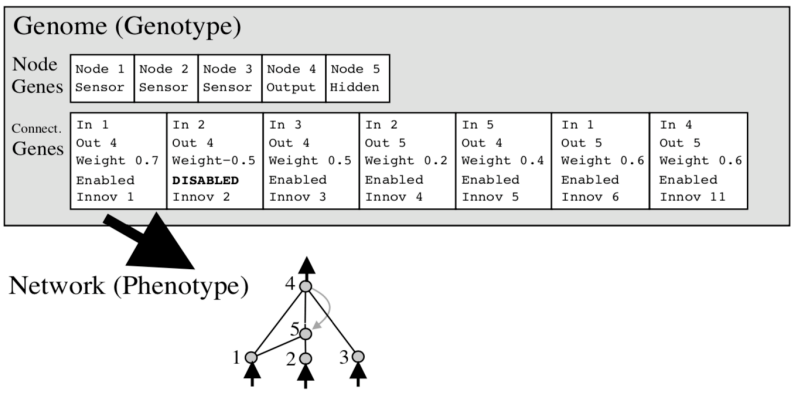

Figure 3: genotype and phenotype from [2]

Genomes in NEAT are linear representations of network connectivity.

The figure 3 illustrates a genotype to phenotype mapping example. A genotype is depicted that produces the shown phenotype. There are 3 input nodes, one hidden, and one output node, and seven connection definitions, one of which is recurrent. The second gene is disabled, so the connection that it specifies (between nodes 2 and 4) is not expressed in the phenotype.

The figure 4 illustrates Matching up genomes for different network topologies using innovation numbers. Parent 1 and Parent 2's innovation numbers (shown at the top of each gene) tell us which genes match up with which. Even without any topological analysis, a new structure that combines the overlapping parts of the two parents as well as their different parts can be created. Matching genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent. In this case, equal fitnesses are assumed, so the disjoint and excess genes are also inherited randomly. The disabled genes may become enabled again in future generations: theres a preset chance that an inherited gene is disabled if it is disabled in either parent.

Figure 5 illustrates the two types of structural mutation in NEAT. Both types, adding/deleting a connection and adding/deleting a node, are illustrated with the connection genes of a network shown above their phenotypes. The top number in each genome is the innovation number of that gene. The innovation numbers are historical markers that identify the original historical ancestor of each gene. New genes are assigned new increasingly higher numbers. To be noted, NEAT begins with a uniform population of simple networks with no hidden nodes and inputs connected directly to outputs.

2.3.3 NEAT algorithm

Without the aid of TD methods, namely Q-learning, NEAT can tacle reinforcement learning in that NEAT does not attempt to learn a value function but tries to find good policies directly by training action selectors, which map states to the action the agent should take in that state, as the same as we do in policy evaluation reinforcement learning, which uses the global optimizing techniques that directly search the space of potential policies.

Algorithm 2 contains a high-level description of the NEAT algorithm applied to an episodic reinforcement learning problem. It differs from the original algorithm in that for every step of choosing network for policy evaluation, the network is not chosen from directly the fixed order of popula-

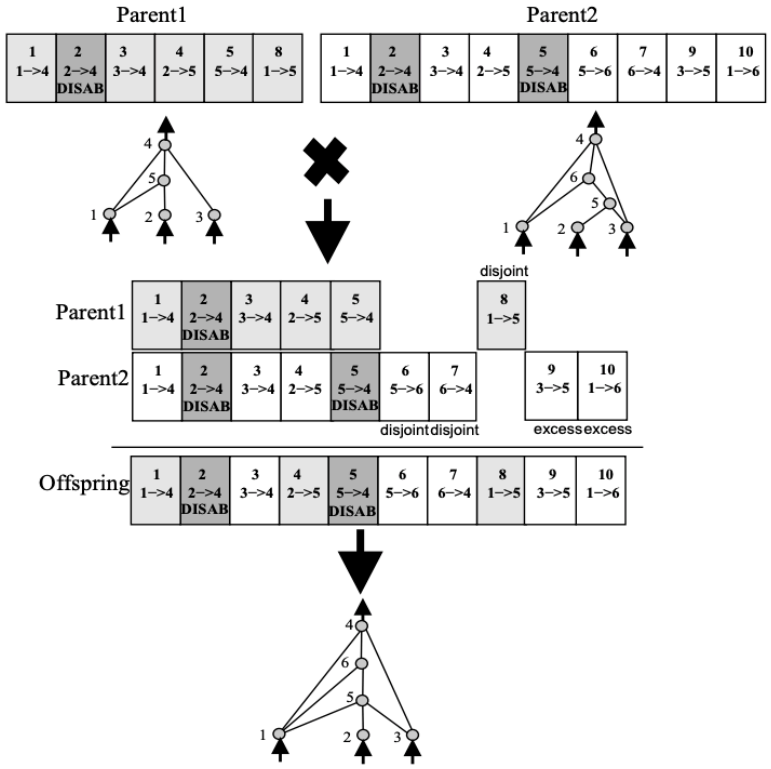

Figure 4: crossover from [2]

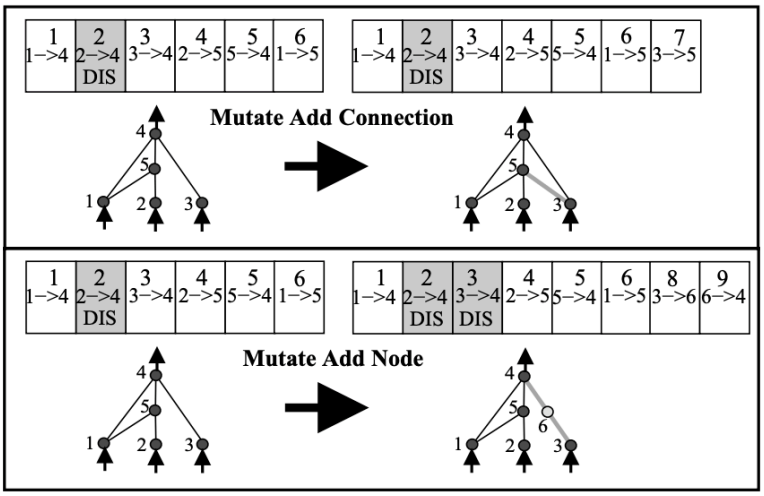

Figure 5: mutation from [2]

tion vector, but randomly selected. This change does not significantly alter NEATs behavior but facilitates to excel online by balancing exploration and exploitation within and across generations.

Algorithm 2: NEAT(S, A, p, m_n, m_l, g, e)

Input: S:set of all states, A:set of all actions, p:population size , m_n: rate of adding node, m_l: rate of adding link, g: generations, e: episodes;
Initialize: $P[] \leftarrow$ Init-Populations(S, A, p) ;
for $i \leftarrow 1$ *to* g **do**
  for $j \leftarrow 1$ *to* e **do**
    $N, s, s' \leftarrow$ Random(P), null, INIT-STATE(S);
    while *Terminal-state?(s)* **do**
      $Q[] \leftarrow$EVAL-NET(N, s');
      $a' \leftarrow \arg\max Q[j]$; $s, a \leftarrow s', a'$; $r, s' \leftarrow$ TAKE-ACTION(a');
      $N.fitness \leftarrow N.fitness + r$
    end
    $N.episodes \leftarrow N.episodes + 1$
  end
  $P' \leftarrow$ new array of size p;
  for $j \leftarrow 1$ *to* p **do**
    $P'[] \leftarrow$ Breed-Net($P[]$);
    with-probability m_n: ADD-Node-Mutation($P'[j]$);
    with-probability m_l: ADD-link-Mutation($P'[j]$)
  end
  $P[] \leftarrow P'[]$
end

2.3.4 speciation

In most cases, adding new structure to a network initially reduces its fitness. However, NEAT speciates the population, so that individuals compete primarily within their own niches rather than with the population at large. Hence, topological innovations are protected and have time to optimize their structure before competing with other niches in the population.

Historical markings make it possible for the system to divide the population into species based on topological similarity. The distance δ between two network encodings is a simple linear combination of the number of excess (E) and disjoint (D) genes, as well as the average weight differences of matching genes (W):

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \bar{W} \tag{3}$$

where the coefficients c_1, c_2 and c_3 adjust the importance of the three factors, and the factor N, the number of genes in the larger genome, normalizes for genome size.

2.4 NEAT+Q algorithm

Since all that is required to make NEAT optimize value functions instead of action selectors is a reinterpretation of its output values, the output of the neural network is alreay identical to what the Q value function represents. Therefore, if the weights of the networks NEAT evolves are updated

during their fitness evaluations using Q-learning and backpropagation, they will effectively evolve value functions instead of action selectors.

Algorithm 3: NEAT+Q(S, A, p, m_n, m_l, g, e)

Input: S, A, p, m_n, m_l, g, e, α, λ, γ, ϵ
Initialize: $P[] \leftarrow$ Init-Populations(S, A, p) ;
for $i \leftarrow 1$ *to* g **do**
  for $j \leftarrow 1$ *to* e **do**
    $N, s, s' \leftarrow$ Random(P), null, INIT-STATE(S);
    while *Terminal-state?(s)* **do**
      $Q[] \leftarrow$EVAL-NET(N, s');
      With-prob(ϵ_{td}) $a' \leftarrow RANDOM(A)$;
      else: $a' \leftarrow \arg\max Q[j]$;
      if $s \neq null$ **then**
        $BACKPROP(N, s, a, (r + \gamma \max_j Q[j])/c, \alpha, \gamma, \lambda)$
      end
      $s, a \leftarrow s', a'$;
      $r, s' \leftarrow$ TAKE-ACTION(a');
      $N.fitness \leftarrow N.fitness + r$
    end
    $N.episodes \leftarrow N.episodes + 1$
  end
  $P' \leftarrow$ new array of size p;
  for $j \leftarrow 1$ *to* p **do**
    $P'[] \leftarrow$ Breed-Net$(P[])$;
    with-probability m_n: ADD-Node-Mutation$(P'[j])$;
    with-probability m_l: ADD-link-Mutation$(P'[j])$
  end
end

To be noted there are two kinds of implementations of NEAT+Q, one is Lamarkian in which the weights are inherited by the next generation, one is Darwinian in which the weights are not. Results will show that there's no advantage in each of them.

2.5 ϵ greedy selection and Boltzmann selection

The algorithm has variant versions in that during the online choosing step we can use either ϵ greedy selection or Boltzmann selection, as we know that in the TD method the step forward process is done by balancing the exploration and exploitation tradeoff using the two selection methods, in particular the probability distribution is constructed using the Q value function:

$$\Pr(\cdot|s) = \frac{e^{Q(s,\cdot)/\tau}}{\sum_{a \in A} e^{Q(s,a)/\tau}}$$

It's remarkable that we can borrow the Boltzmann selection and the ϵ greedy selection in the exploration and exploitation tradeoff acrossing the generations, only by replacing the Q function with fitness function:

$$\Pr(\cdot) = \frac{e^{S(\cdot)/\tau}}{\sum_{q \in P} e^{S(q)/\tau}}$$

where $S(q)$ is the average fitness of the policy q represented by the neural network, τ is the Boltzmann temperature, as τ goes to 0, the selection becomes greedy, otherwise completely random.

Thus regarding the online step variantion, take the Boltzmann selection for example, should be:

Algorithm 4: Boltzman Selection(P, τ)

Input: P : population, τ: softmax temperature
if $\exists N \in P \mid N.episodes = 0$ **then**
  return N
else
  $total \leftarrow \sum_{N \in P} e^{N.average/\tau}$;
  for $N \in P$ **do**
    with-prob ($\frac{e^{N.average/\tau}}{total}$) return N else $total \leftarrow total - e^{N.average/\tau}$
  end
end

3 Experiments results

3.1 Mountain car task and server job scheduling task

In the mountain car task depicted in Figure 6, an agent strives to drive a car to the top of a steep mountain. The car cannot simply accelerate forward because its engine is not powerful enough to overcome gravity. Instead, the agent must learn to drive backwards up the hill behind it, thus building up sufficient inertia to ascend to the goal before running out of speed.

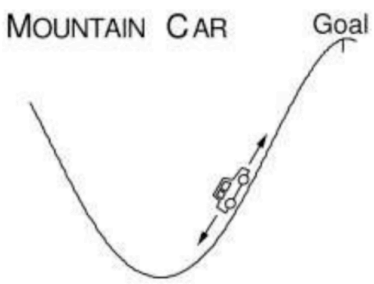

Figure 6: figure from [1]

In the server job scheduling task, which is used to assess whether the methods can scale to a much more complex problem, a server, such as a websites application server or database, must determine in what order to process the jobs currently waiting in its queue. Its goal is to maximize the aggregate utility of all the jobs it processes. A utility function for each job type maps the jobs completion time to the utility derived by the user. The problem of server job scheduling becomes challenging when these utility functions are nonlinear and/or the server must process multiple types of jobs.

3.2 Comparisons

Several comparisons regarding the comparisons between different version of algorithms are listed in this section:

3.2.1 Manually and evolutionary

In both domains of tasks, evolutionary function approximation significantly improves performance over manually designed networks, as shown in 7

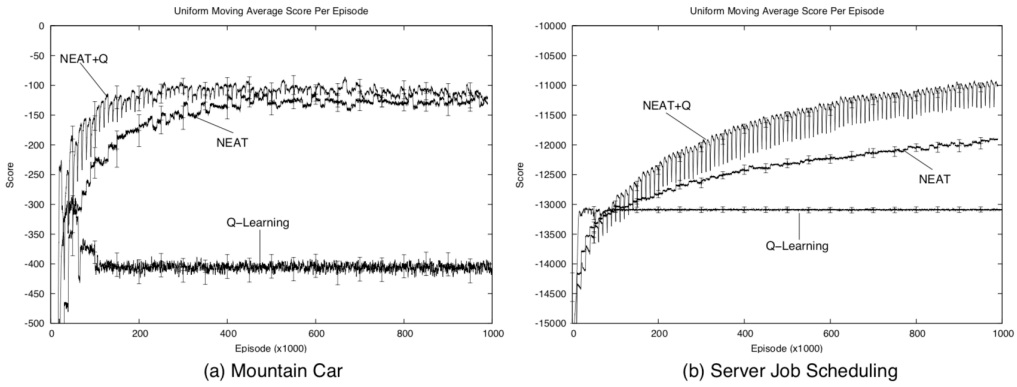

Figure 7: performance of the three algorithms

3.2.2 softmax and off-line

As for on-line v.s. off-line, the softmax scheme significantly outperformsthe off-line algorithm as shown in 8, original paper also explored the comparison between ϵ greedy and softmax, surprisingly they perform just closely to each other.

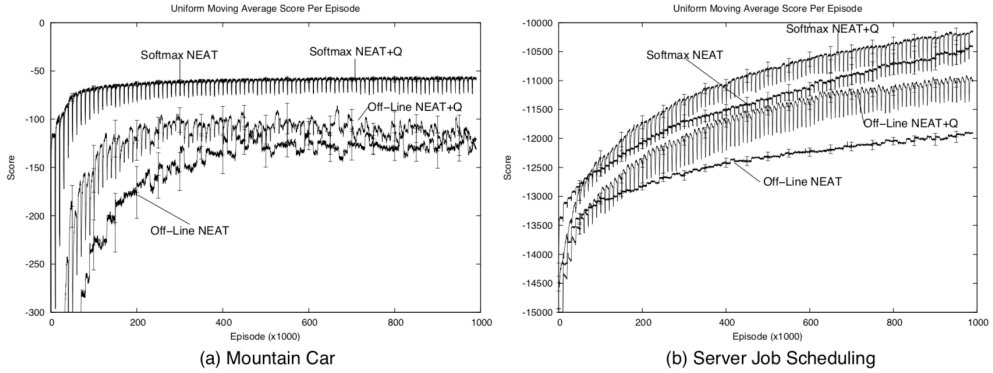

Figure 8: performance of Softmax and off-line NEAT+Q

3.2.3 Lamarkian and Darwinian

Since evolutionary function approximation can be implemented in either a Darwinian or Lamarckian fashion, the two approaches empirically in both the mountain car and server job scheduling domains were compared. However, it is not clear that this approach is superior even though it more closely matches biological systems, as shown in 9

3.2.4 annealing and not annealing

Two scenarios, one where the learning rate is annealed to zero after 100 episodes, just as in training, the other where it is not annealed at all were compared in original paper, it's remarkable that under the condition without annealing the learning rate, the online scheme actually performs worse, while with annealing the learning rate will eliminate this phenomenon.

4 Simulations and comparison

There are some of the results demonstrated with an emphasis on NEAT algorithm.

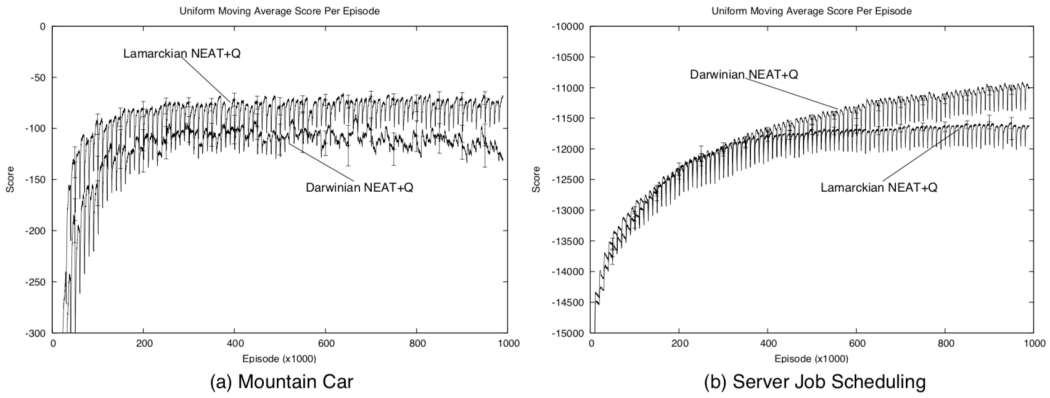

Figure 9: performance of Darwinian and Lamarckian NEAT+Q

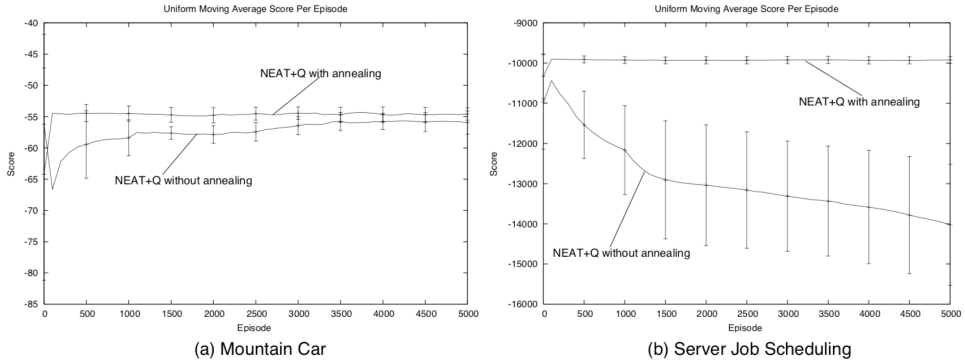

Figure 10

4.1 Cartpole

In order to illustrate how simple the structure that NEAT automatically searched is, the first environment we tested is the cartpole, also known as the inverted pendulum, the observation space is consist of 4 variables, corresponding to 4 dimensional inputs, and the action space consisting of 2 elements. we simply set the initial neural network to be with 4 inputs and 2 outputs, with no hidden layers.

As 11 shows, we are able to find the best individuals from the population within 2 generations, partially because neural network has been achieved great success and simplicity of the environment model. The final winner's structure is as shown in 12, is surprisingly simple as red lines indicate that the connectivity is enabled.

Table 1 shows some basic configuration that was used for NEAT.

Table 1: parameter setting

population size	mutation rate	Node add/delete	Connection add/delete	activation	elitism
100	0.1	0.9/0.2	0.9/0.1	relu	2

As a comparison, if we change the hidden layer into 1, the learning curve becomes 13, which means the manually determined structure could slow down the automate search.

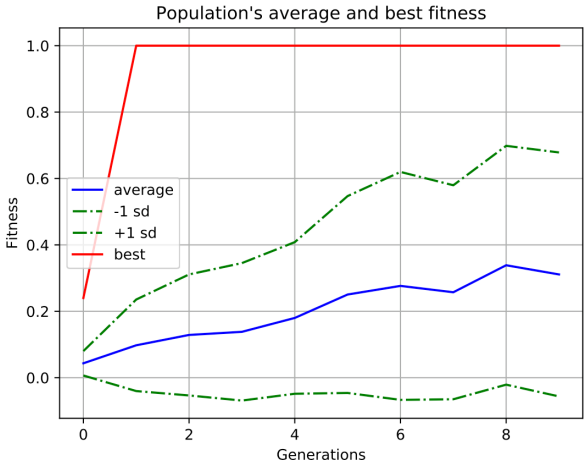

Figure 11: Learning curve of cart pole

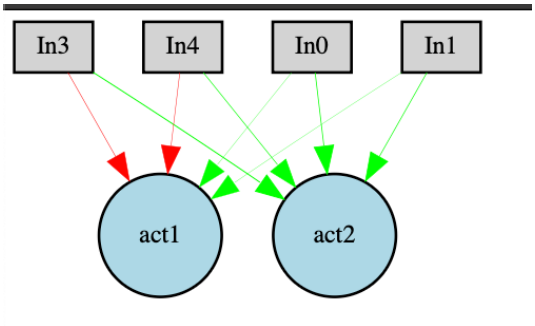

Figure 12: Final conectivity of Cart pole

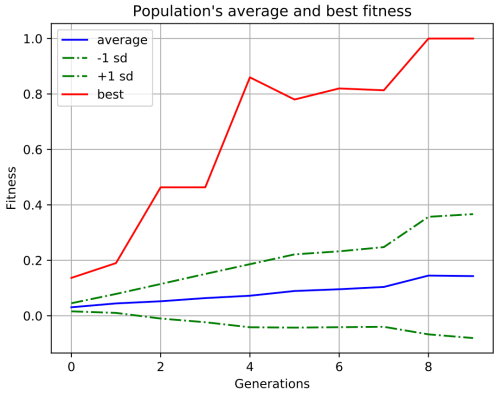

Figure 13: learning curve of cart pole with 1 hidden layer initialized

4.2 Atari game: Pac-man

An environment with larger complexity called Pac Man has also been tested using the NEAT algorithm, in this environment, the observation space is 128 dimensional, the action space is 9 dimensional, therefore we set the initial structure to have 128 inputs with 9 dimensional sigmoid outputs, and 8 hidden layers, table 2 shows the basic configuration of neat algorithm:

Table 2: parameter setting

population size	mutation rate	Node add/delete	Connection add/delete	activation	elitism
150	0.1	0.9/0.2	0.25/0.1	relu	2

After 1000 generations of training, each generation contains variant episodes within which my action in the environment will be finished, learning curve is just shown in Figure 14.

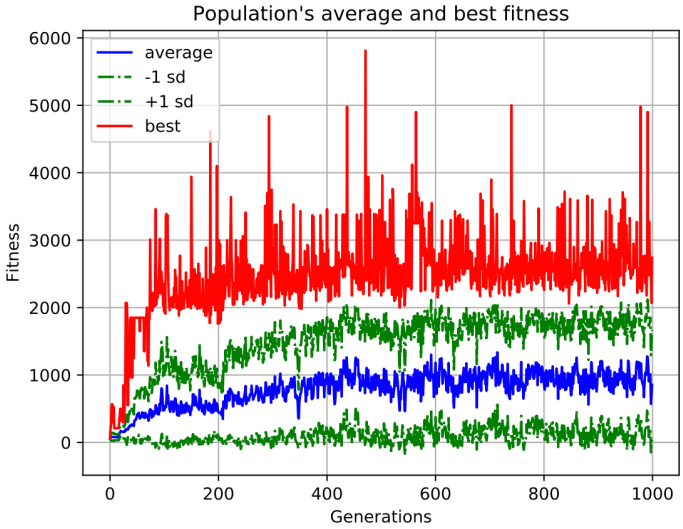

Figure 14: Learning curve of Pac-man

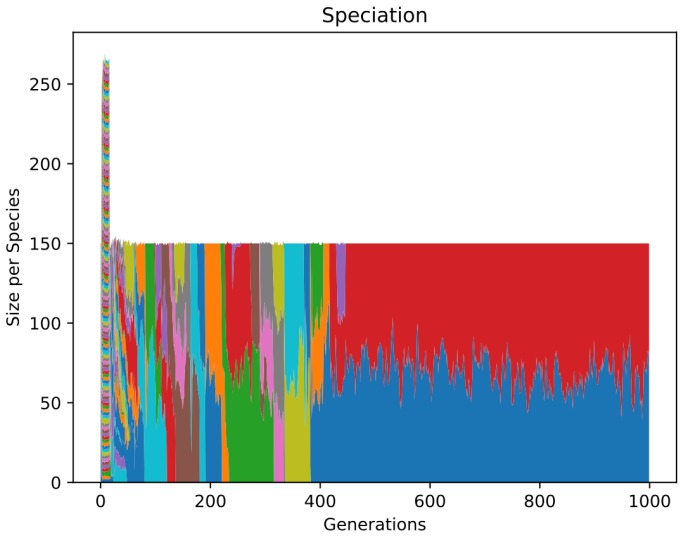

Figure 15: Evolution diagram of Pac-man

Limited by the scale of graph, the connectivity is not shown here, instead the species' evolution is shown, from which we can observe that there are initially very dense species, as the generation goes, the genetic distance between species will be closer and the genomes keep competing with each other within their genetic niches, finally the 2 elite species which survived the best in the environment will be chosen left.

4.3 compare to Q-learning

In both cases the NEAT algorithm significantly outperform DQN. As we manually determine the network structure, take cart pole for e.g., the network defined contain 3 hidden layers with 8, 16, 8 nodes respectively, the speed of training was remarkablely slow compare to NEAT(requires above 5000 episodes while NEAT only required 10 generations); As for in the Atari game case, both algorithms are unable to training a action selector that tacle the game, while NEAT perform better: end up with 210 reward and DQN ended up with 70 reward. The reason for this phenomenon might be because of the on-stationary environment, as in the Atari game, the ghost can be conceived as some randomness that the states doesn't consider, yet it's rather difficult to enumerate the states that absorbing the randomness.

5 Conclusion

Having evidences listed before, we can summarize that :

- NEAT outperform Q-learning in episodes, Generally, NEAT+Q can perform better as shown in the mountain car and server job scheduling domain;
- NEAT explore the function representation automatically, similar to exploring a better policy, the episodes of ficticious play is equivalent to what we do during policy evaluation, only different in that the policy is hidden beneath the represented function of neural network, and we have multiple choice when it comes to policy update.
- It's safe to anticipate that some other Methods(DDQN and Duel QN) can be combined with NEAT, but in order to in corporate with those tools we have to add properties to the existing NEAT system so that it can perform backpropagation.
- although the training process might work in some deterministic systems, it is still challenging to tacle non-stationary environment as the action selector only take the current states into consideration.

References

[1] Whiteson, Shimon, and Peter Stone. "Evolutionary function approximation for reinforcement learning." Journal of Machine Learning Research 7.May (2006): 877-917.

[2] Stanley, Kenneth O., and Risto Miikkulainen. "Efficient reinforcement learning through evolving neural network topologies." Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation. Morgan Kaufmann Publishers Inc., 2002.

[3] Mitchell, Melanie. An introduction to genetic algorithms. MIT press, 1998.