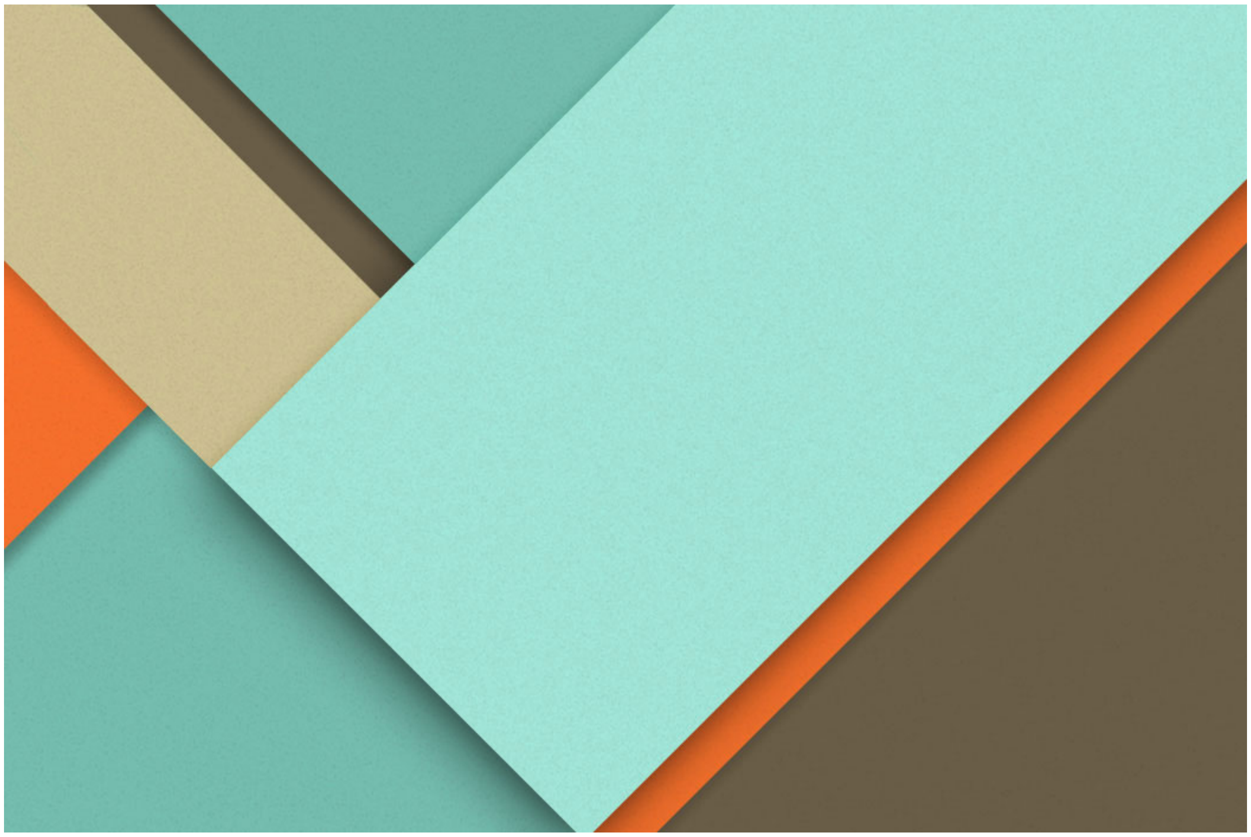


NeuroEvolution of Augmenting Topologies (NEAT)

03.24.2017

Lee, Suckgeun

Overview

NeuroEvolution of Augmenting Topologies (NEAT) is a reinforcement learning algorithm (RL) using Neural Network (NN) and Genetic Algorithm (GA) developed by Ken Stanley at 2002. NEAT first creates many NNs randomly, then use GA to find the optimal solution. It is known that NEAT performs well on simple tasks such as Pole Balancing problem; however, NEAT is also known for its poor performance at some complicated problems. In this report, I will implement NEAT from scratch, then test it on two relatively simple tasks and one complex task. For simple tasks, cart-pole problem [Barto83] and mountain-car problem [Moore90] are used; and for complex problem, SpaceInvaders from atari2600 game is used. The final goal of this report is to test if NEAT is capable of solving complex problem such as game playing tests. Every test will use OpenAI gym, and the NEAT algorithm I implemented is used, not other open source NEAT.

Problem Statement

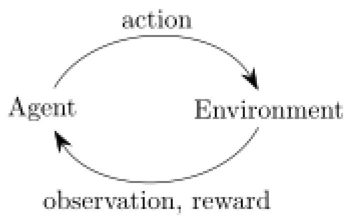
There are many different versions of NEAT, but in this paper, I will use original NEAT the paper (Stanley 2002) describes. I will implement NEAT from scratch, and test it if NEAT could solve cart-pole problem and mountain-car problem. Details of cart pole problem and mountain-car problem are described in “Cart-pole test” section and “Mountain-car test” section.

The final goal of this report is testing if NEAT could learn how to play SpaceInvaders (OpenAI gym environment) using RAM of the Atari machine. Playing Atari2600 games using only RAM of the Atari machine is known for its difficulty. So far, none of existing RL algorithms successfully solves the problem. It is expected that NEAT also fails on this problem, but to see how it fails will be interesting. Details of SpaceInvaders test is described in “SpaceInvaders (Atari 2600) test section.”

OpenAI environment

In this report, OpenAI gym will be used as a test environment.

OpenAI uses the classic "agent-environment loop". Each timestep, the agent chooses an action, and the environment returns an observation and a reward.



<https://gym.openai.com/docs>

When game starts, environment returns observation and reward to the agent. The agents get the information, then choose appropriate action, then returns the action to the environment. This loop continues until desired reward is acquired.

Metrics

Since all test environments this report uses are OpenAI gym environments, OpenAI metric system is used, such that NEAT can be compared with other algorithms on OpenAI score board. Below is some terminologies that this report uses.

1. Episode: a series of time steps (from initial to end point of the test environment)
2. Reward: reward of “each time step”
3. Total reward: total reward of “One Episode”
4. Time to solve: how long does it take to solve a problem

NEAT will continue to learn until it satisfies conditions of “solving” problem. For example, “solving” cart-pole problem is defined as getting average total reward of 195.0 over 100 consecutive trials. Definitions of each test’s “solving problem” are introduced in each test section.

When NEAT finally solves the problem, NEAT’s performance is measured by each episode’s total rewards, number of episodes, and time to solve. So NEAT’s performance is measured by

1. How many times does NEAT have to experience the environment (number of episodes)
2. How long does NEAT takes to solve problem (time to solve)
3. How correct the NEAT solve the problem (total reward)

In this way, NEAT is tested not only its correctness, but also how fast it learns. The test environment will be OpenAI scoreboard such that it can be compared with other algorithms in the same environment.

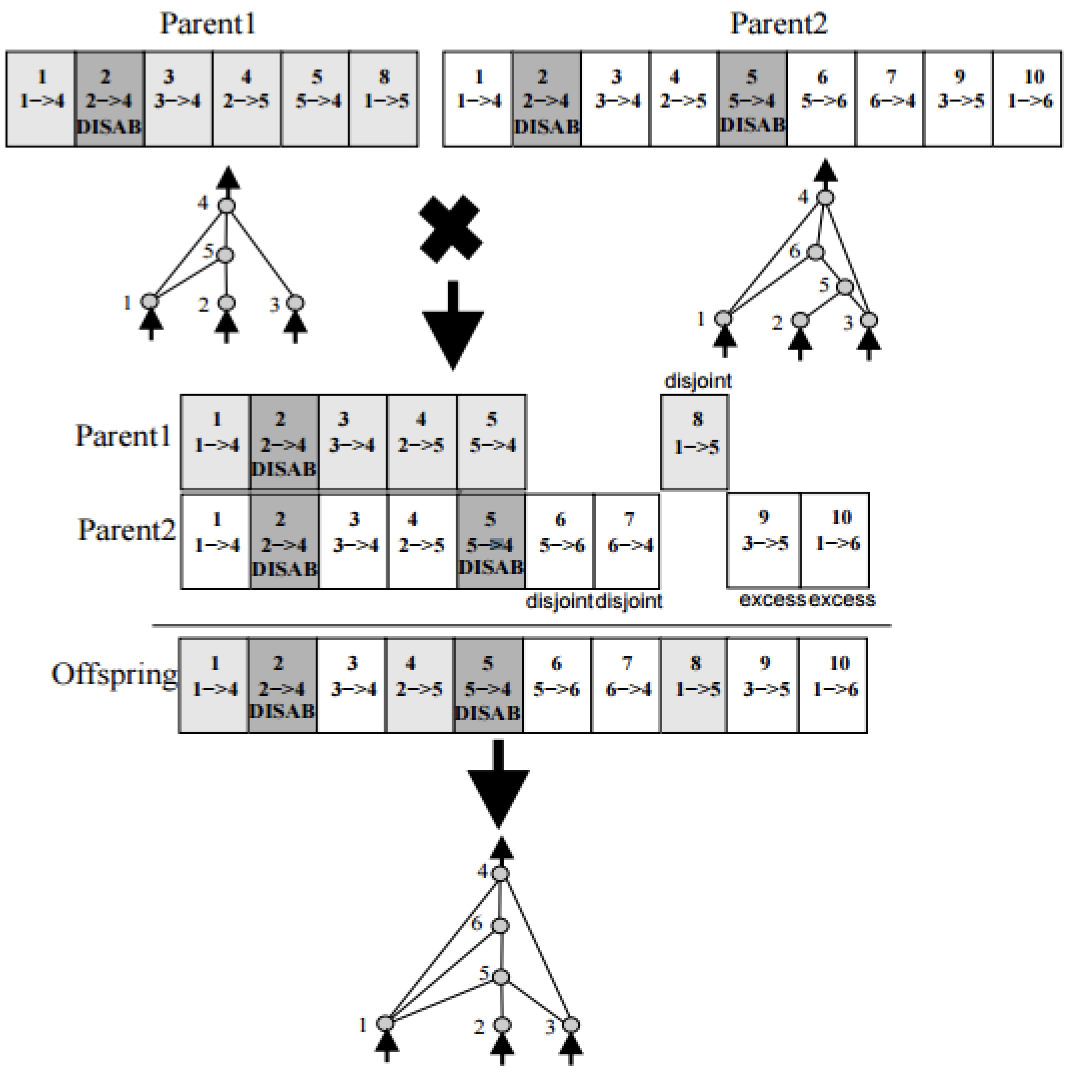
Introduction to NEAT algorithm

NEAT is developed by Ken Stanley at 2002. Unlike other popular NN algorithms, NEAT does not use backpropagation to find it's weights. Also, NEAT starts from very simple structure then slowly evolves itself to more complex structure. The evolving NNs is accomplished by using Genetic Algorithm(GA), and some brilliant tactics are introduced to effectively evolve both structure and weight of NN. The overview of NEAT algorithm is as follows.

1. Create multiple NNs with all input and output nodes are connected (first generation).
2. Run simulation for all NNs, then score(fitness) every NNs.
3. Adjust scores(fitness) of each NNs using “explicit fitness sharing(explained later)”
4. Drops inferior NNs
5. Crossover NNs (Creating next generation)
6. Mutate weights, connections, and nodes of next generation to change network structure.
7. Go to 2. until desired result is acquired.

The above list looks like just another regular GA algorithm, but what differentiates NEAT from other NeuroEvolution (NE) algorithms and GAs is how NEAT creates and manages its next generation. Since NEAT changes both weight and network structures, it is hard to crossover(making a child using two parents) with parents with two totally different structures. Also, when it changes its number of nodes and connections by mutation(changing structure with some probability), there is high possibility that the mutated new NN performs poorly, thus fails to survive against the old and more tuned NNs. To solve the problems, NEAT introduces “Historical Marking” and “Speciation”.

When two NNs with different structures crossover, “historical marking” is used to check where and how to crossover. The NEAT paper introduces this process as follows.



<from NEAT paper>

Figure 4: Matching up genomes for different network topologies using innovation numbers. Although Parent 1 and Parent 2 look different, their 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: there's a preset chance that an inherited gene is disabled if it is disabled in either parent.

NEAT also introduces “Speciation” to protect newly created structure. When NEAT mutates its network structures, there is high possibility that new structures are premature compared to the already tuned structures. To protect new structures and encourage to evolve NNs, NEAT groups NNs into different species using “explicit fitness sharing.” Each NNs are evaluated its structure and assigned to a similar species. NNs compete each other within the species such that it gives new structure enough time to tune itself and survive. Details of explicit fitness sharing is explained well on NEAT paper.

By using these “historical marking” and “speciation”, NEAT effectively changes its structure throughout the generation.

Parameters of NEAT

NEAT has quite a lot of parameters. Below is the explanation of each parameters.

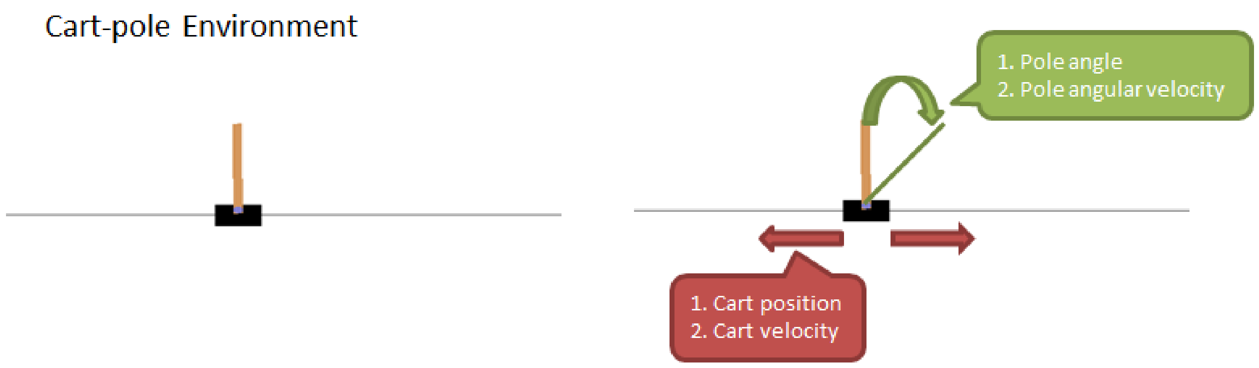
1. n_input = number of inputs of each neural networks
2. n_output = number of outputs of each neural networks
3. n_nn = number of total neural networks to generate
4. activ_func = activation function to use. (default: sigmoid function)
5. bias = bias node. NEAT has only one bias node, and it always outputs same number.
6. c1, c2, c3, cmp_thr = parameters for “explicit fitness sharing”
7. drop_rate = when creating next generation, drop the poor performing NNs with this rate. (default: 80%)
8. weight_mutate_rate = how much weight should be changed when mutating weight
9. n_champion_keeping = decide how many best performing NNs to keep unchanged to the next generation (default: 10)
10. pc = probability of crossover. For this probability, crossover is executed. Otherwise, NN is just copied to next generation. (default: 75%)
11. pc_interspecies = probability of inter species crossover. (default: 0.1%)
12. pm_node_small = probability of node mutation with small species. (default: 3%)
13. pm_node_big = probability of node mutation with big species. (default: 30%)
14. pm_connect_small = probability of connection mutation with small species. (default:5%)
15. pm_connect_big = probability of connection mutation with big species. (default:70%)
16. pm_weight = probability of weight mutation (default: 80%)
17. pm_weight_ramdom = probability of randomly mutating weight (default: 10%)
18. pm_disable = probability of disabling a connection (default: 40%)
19. pm_enable = probability of enabling connection (default: 20%)
20. weight_max = maximum weight
21. weight_min = minimum weight

Among these parameters, number of input, number of output, number of NNs, bias, c1, c2, c3, weight mutation rate, weight max, and weight min will be adjusted for each test. Other parameters will use default values. The default values are from either NEAT paper or experiments.

Cart-pole test

Here, I will test my implementation of NEAT using cart-pole problem.

Data Exploration and Visualization



Cart-pole test uses OpenAI environment as its input data. In the environment, there is a cart that moves along the horizontal axis, and a pole is attached to it. At every time step, one can observe cart position, cart velocity, pole angle, and pole angular velocity. Those four observations are used as input. At any state, there are two actions the cart can take; go right or left. When the pole loses its balance, or the cart is off the screen, one episode finishes. The goal of this test is maximize the total reward by balancing the pole.

Solving Condition

Cart-pole defines "solving" problem as getting average reward of 195.0 over 100 consecutive trials.

NEAT parameters

The parameters used for this test are as follows.

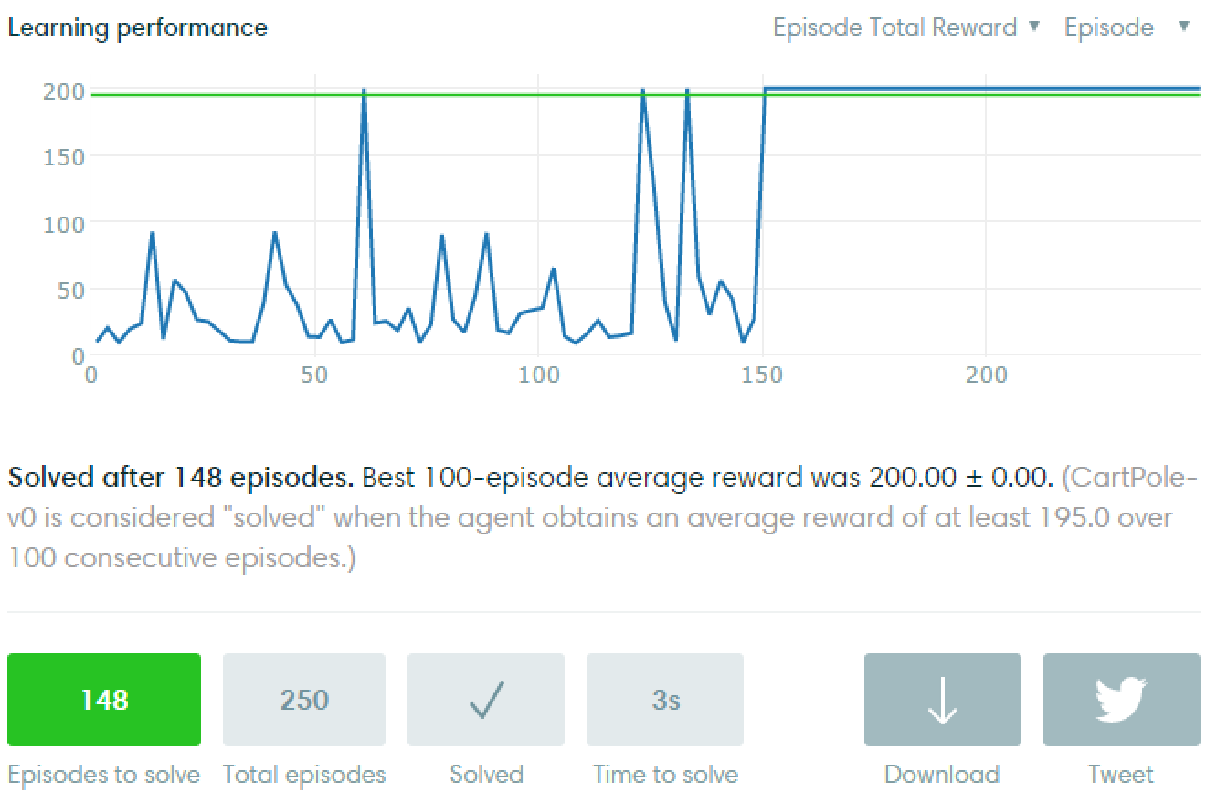
1. number of inputs = 4
2. number of outputs = 2
3. number of total NNs = 50
4. c1 =2, c2 = 2, c3 = 1

5. bias = 1
6. weight_max = 3
7. weight_min = -3
8. Weight_mutate_rate = 0.005

c1, c2, and c3 are chosen such that speciation creates sufficient species, and weight max, min, and mutate rate are chosen by experiments.

Benchmark

Below is the test result.



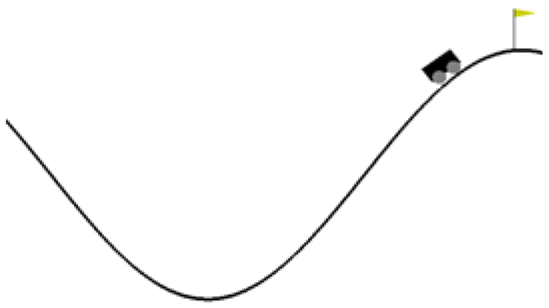
Solved after 148 episodes. Best 100-episode average reward was 200.00 ± 0.00. (CartPole-v0 is considered "solved" when the agent obtains an average reward of at least 195.0 over 100 consecutive episodes.)

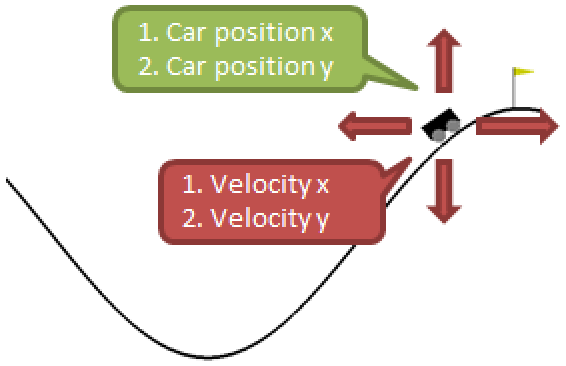
148	250	✓	3s		
Episodes to solve	Total episodes	Solved	Time to solve	Download	Tweet

<https://gym.openai.com/evaluations/eval_4JYXqlYAQYGKQU0lRZriQ>

NEAT is able to solve the problem in 148 episode, in 3 seconds. It is not significant compared to the scores on OpenAI scoreboard; however considering there are lots of algorithms instantly solves the problem in the scoreboard, it is doubtful the scores are real. The fact that it could solve the problem in 3 seconds is fast enough to say NEAT performs very well on this particular problem.

Mountain-car test





Here, I will test my implementation of NEAT using mountain-car problem.

Data Exploration and Visualization

Mountain-car test uses OpenAI environment as its input data. Wikipedia explains Mountain-car environment as follows.

> Mountain Car, a standard testing domain in reinforcement learning, is a problem in which an under-powered car must drive up a steep hill. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill. The domain has been used as a test bed in various reinforcement learning papers.

At every time step, one can observe the car's position (x, y) and the velocity of the car (v_x, v_y). At any state, there are two actions the car can take; go right or go left. At every time step, negative reward is given, and when the car reaches at the top of the mountain where the flag is at, the game ends. So the goal of the agent is to get to the flag as soon as possible not to get the negative reward.

Solving Condition

Mountain-car defines "solving" as getting average reward of -110.0 over 100 consecutive trials.

NEAT Parameters

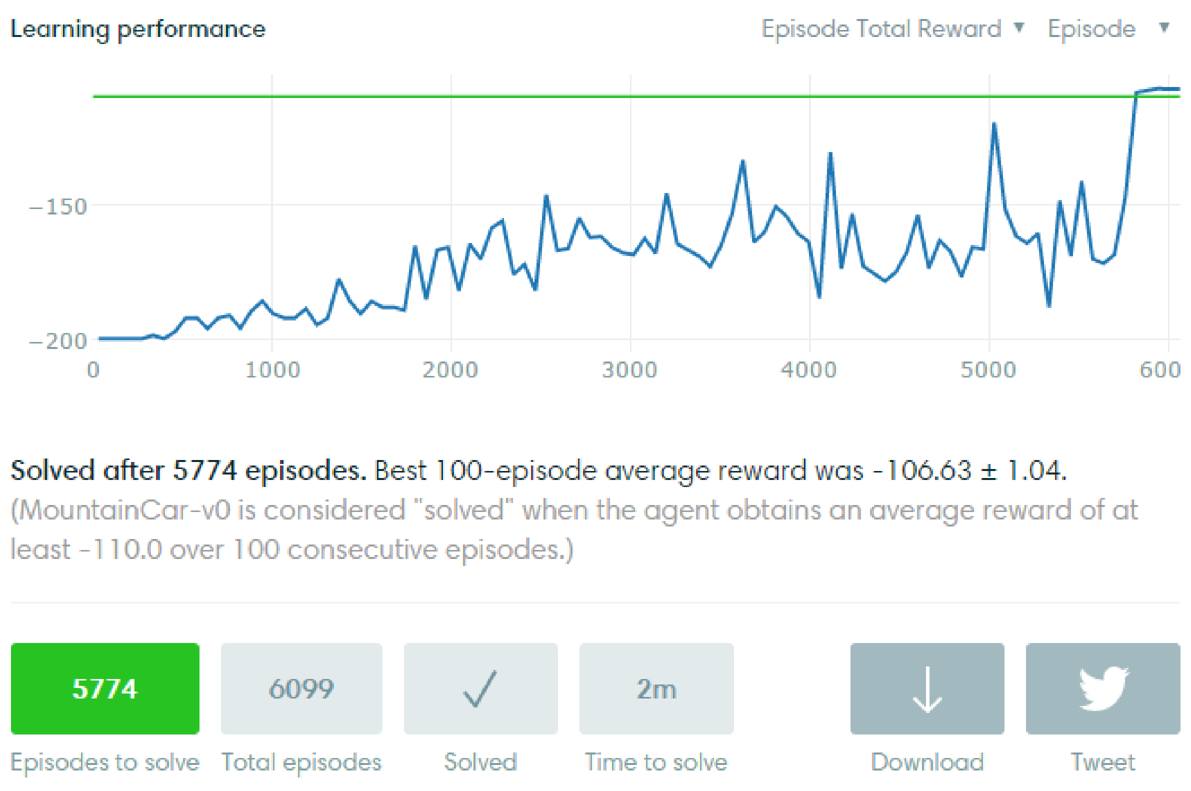
The parameters used for this test are as follows.

1. Number of inputs = 4
2. Number of outputs = 2
3. Number of total NNs = 150
4. c1 =2, c2 = 2, c3 = 1
5. bias = 1
6. Weight_max = 10
7. Weight_min = -10
8. Weight_mutate_rate = 0.1

c1, c2, and c3 are chosen such that speciation creates sufficient species, and weight max, min, and mutate rate are chosen by experiments.

Total 30 generations are generated as test, and each generation's score is taken from its best individual NN. When the car fails to finish the task in 300 time steps, it finishes the trial and starts the next one. Below is the test result.

Benchmark



Solved after 5774 episodes. Best 100-episode average reward was -106.63 ± 1.04.
(MountainCar-v0 is considered "solved" when the agent obtains an average reward of at least -110.0 over 100 consecutive episodes.)

<https://gym.openai.com/evaluations/eval_4eyhoLOMTeql45I3iXdMYA>

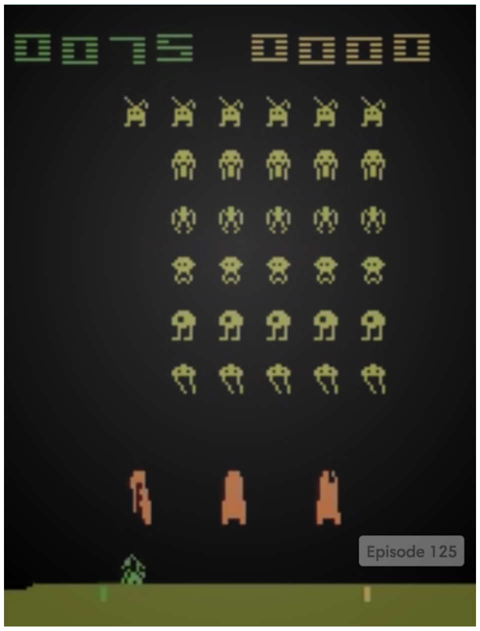
Name	**Episodes to solve**	**Time to solve**
dhfromkorea's algorithm	584.0	2 min
tsetimmy's algorithm	686.0	36m
lguye's algorithm	1299.0	2m
suckgeun's algorithm (NEAT)	1499.0	42s
tsetimmy's algorithm	2329.0	1h
syllogismos's algorithm	2540.0	3m
suckgeun's algorithm (NEAT)	3746.0 2	96s
suckgeun's algorithm (NEAT)	5774.0	2m

<NEAT comparison with other algorithms>

Above is the table of top algorithms in OpenAI (03/29/2017). Compared to other algorithms, NEAT tend to use more episodes, but the time to solve is significantly lower. The reason NEAT uses more episodes is that NEAT generates 150 neural networks as part of GA algorithm, and each neural networks takes couple of episodes to train. Why NEAT uses short time to train is because GA's characteristic. Since NEAT generates many neural networks and tunes them through GA, there is almost no computation on weight tuning. That is why NEAT is fast compared to other algorithms.

Also, notice that sometimes NEAT could find the solution very fast, but sometimes it takes more time. That is also characteristic of GA. When GA first generates NNs, if there is some NNs closer to the answer of problem, it takes much less time. However, if there are no NNs with possible solution, NEAT takes more time to find solution.

SpaceInvaders (Atari 2600) test



<SpaceInvaders>

Here, I will test my implementation of NEAT using mountain-car problem.

Data Exploration and Visualization

SpaceInvaders test uses OpenAI environment as its input data.

OpenAI describes the SpaceInvaders as follows.

> Maximize your score in the Atari 2600 game SpaceInvaders. In this environment, the observation is the RAM of the Atari machine, consisting of (only!) 128 bytes. Each action is repeatedly performed for a duration of kk frames, where kk is uniformly sampled from {2,3,4}.

On this test, the classic game called SpaceInvaders from Atari2600 in OpenAI gym will be used as testing complex environment. The goal of SpaceInvaders is to destroy as many invaders as possible while avoiding the enemy fires. As input data, RAM of the Atari machine will be used. There are 128 bytes in the RAM of Atari machine, and each byte is represented with numbers in range 0 to 255. About outputs, there are total 6 possible actions; do nothing, fire missile, move to left, move to right, move to left while firing a missile, and move to right while firing a missile. Destroying one invader gives 15 score.

Solving Condition

Unlike previous two tests, this test will not have “solving” defined since OpenAI does not define it. Instead of “solving”, the final score will be compared with random. Also, this part does not use OpenAI metrics because of lack of computation power.

NEAT Parameters

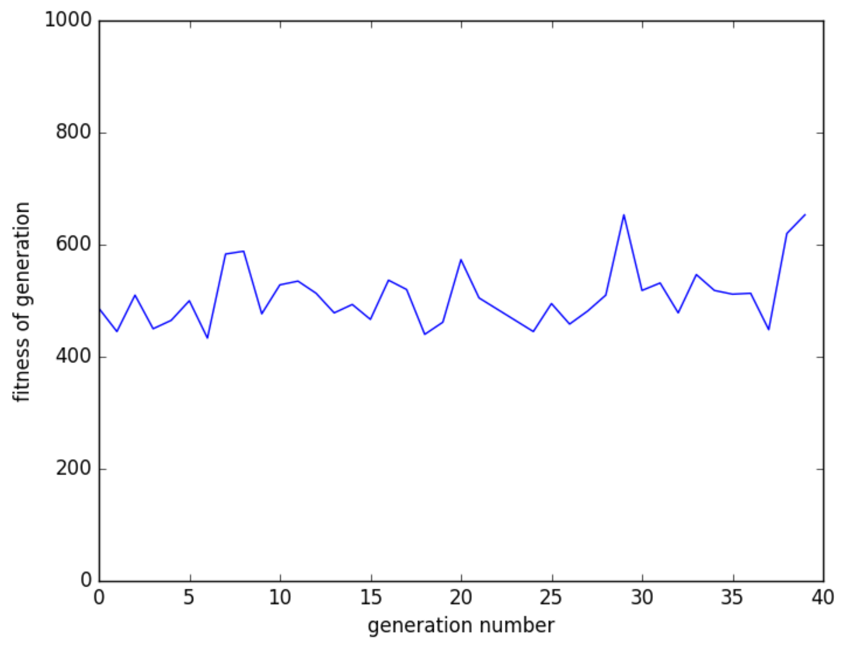
The parameters used for this test are as follows.

1. Number of inputs = 128
2. Number of outputs = 6
3. Number of total NNs = 150
4. c1 =300, c2 = 300, c3 = 150
5. Bias = 1
6. Weight_max = 3
7. Weight_min = -3
8. Weight_mutate_rate = 0.1

c1, c2, and c3 are chosen such that speciation creates sufficient species, and weight max, min, and mutate rate are chosen by experiments.

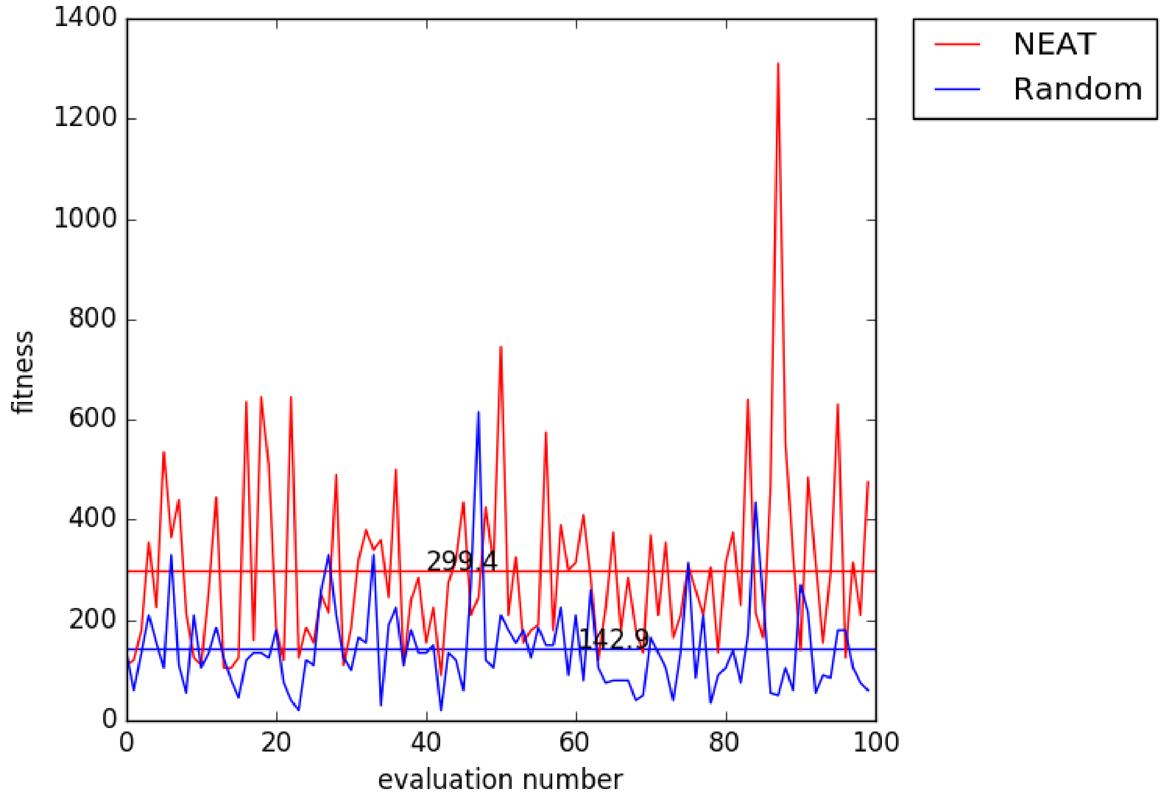
Benchmark

Total 40 generations are generated as test, and each generation’s score is taken from its best individual NN. Below is the test result.



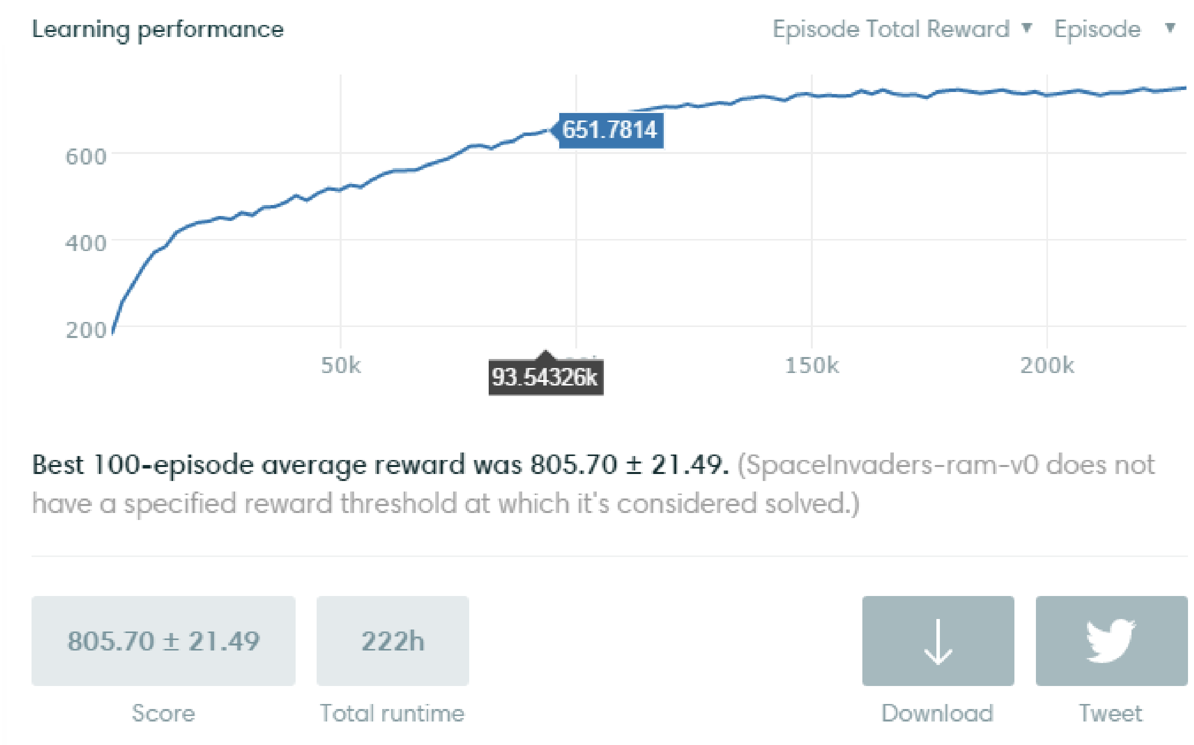
As shown in the graph, through the 40 generation, there is no significant performance improvement. Note that the recorded fitness is the best performed fitness, thus the actual fitness should be much lower. The graph shows that NEAT is not learning well.

The best scored NN is recorded and ran 100 consecutive trials. Below graph shows comparison between NEAT agent and random action gameplay.



The NEAT agent is getting average of 299.4 scores, and the random action game play is getting 142.9 scores. NEAT is learning to play the game, but the performance is not so great.

Compared with the best score on OpenAI scoreboard, NEAT performs significantly worse.



Best 100-episode average reward was 805.70 ± 21.49. (SpaceInvaders-ram-v0 does not have a specified reward threshold at which it's considered solved.)

<ceobillionaire's algorithm from OpenAI score board>

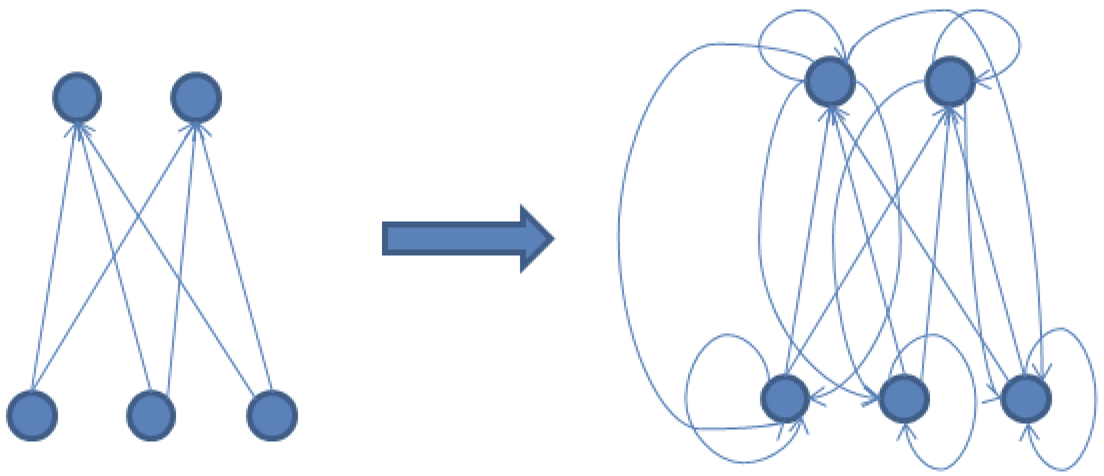
Conclusion

I have tested NEAT with two simple tests and one complex test. The two simple test, NEAT has performed very well compared to other algorithms on OpenAI scoreboard. However, NEAT has failed to learn the complex problem, SpaceInvaders.

Hypothesis why NEAT fails on complex problems

One possible hypothesis why NEAT performs well on simple task but not on complex task is how NEAT evolves its Topologies.

Consider a neural network where 3 inputs and 2 outputs exist. When mutating, there are total 11 possible connections to add. If you add node mutation, there are 17 mutations to consider. Even this simple neural network has so many possibilities of mutation. What if the neural network has hundreds of inputs and tens of outputs? The possible mutation combinations grows exponentially, and it is simply not possible to check all of them. Even worse, since NEAT is tuning its weight by GA, you have to consider weight mutations, and that increases the search space of NEAT even more.



<complex mutation of NEAT>

When NEAT is relatively small, the possible combination is also small, so NEAT can find its optimal solution effectively. However, when it comes to hundreds of inputs, there are too many topologies and weight combinations, so NEAT fails to find the solution.

To justify this hypothesis, further investigation is needed.

Drawback of NEAT

The advantage of NEAT is that it finds the optimal topology of neural network by itself. However, to use NEAT effectively, one should carefully choose parameters of NEAT, and that is solely done by try and error. When I tested NEAT on the three tests on this paper, I had to run many trials to find the best parameters for each problem. When the parameters were not correctly set, NEAT could not find the solution easily, so I had to rerun the test. Even the original NEAT paper writes that it chooses its test parameter by many trials. This means that for each problems, one should run many trials to find best parameters, and that task is time consuming. This is a big disadvantage of NEAT.

Possible improvements

Reduce the search space

When solving complex problems, one should guide NEAT by reducing the possible topologies. For example, by only allowing forward connection, one can significantly reduce the possible topologies. Also, choosing crossover by some logic can be used to guide the topology.

Auto parameter setting

There are too many parameters to consider, and the try and errors to find parameters are time consuming. To reduce the time cost, auto tuning logic can be implemented by some simple logic. For example, a user inputs an initial parameters. Then NEAT runs automatically to see if the parameters are suited for the problem. When the parameters are not suited for the problem, NEAT stops itself and auto tune its parameter.

Final thoughts

I have implemented NEAT and tested using three test problems. Because of lack of time, the NEAT implementation is rather inefficient and has lots of rooms for improvement. For future work, I would like to improve the speed of my NEAT implementation. Also, I'd like to implement DQN and compare it to NEAT.

References

Kenneth O. Stanley, et al, (2002), Evolving Neural Networks through Augmenting Topologies, http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

David Beasley, et al, (1993), An Overview of Genetic Algorithms :Part 1, Fundamentals, University Computing, 1993, 15(2) 58-69.

Matthew Hausknecht, et al, (2014) A Neuroevolution Approach to General Atari Game Playing