Solving games with AI

A comparison between Reinforcement Learning and Genetic Algorithms

Giovanni De Toni, Andrei Catalin Coman, Seyed Mahed Mousavi
Department of Information Engineering and Computer Science
University of Trento, Italy
{giovanni.detoni, andreicatalin.coman, seyedmahed.mousavi}@studenti.unitn.it

***Abstract*—One of the objectives of Artificial Intelligence (AI) is to perform tasks like a human being. Well-known games are one of the domains that have been exploited for this purpose. Experts intention is to design systems capable of playing games in order to perform as well as or even outperform a human counterpart. In this study, we tend to introduce Bio-Inspired solutions for playing 2D games. We designed a Feedforward Neural Network whose weights are evolved by means of the Differential Evolution Algorithm. Then we employed the NeuroEvolution of Augmenting Topologies Genetic Algorithm to evolve a Neural Network together with its weights. As baseline, we designed a Reinforcement Learning algorithm for some games in order to be able to obtain performance comparison between different approaches. We mainly focused on Flappy Bird and Pac-Man as case studies. However, we tried few other experiments within the OpenAI's Gym toolkit to observe our solutions' performance.**

***Index Terms*—Differential Evolution, NeuroEvolution of Augmenting Topologies, Feedforward Neural Network, Reinforcement Learning, OpenAI, Gym**

I. Introduction

WITH the development of Artificial Intelligence, inspired by natural biology, experts tried to create systems which are able to perform a task just like a human being. As one of the main interests of all people is playing games, they attracted a huge amount of attention of experts to consider them as their experimental environment. Classic video games of first generation consoles have been exploited for this purpose due to their simple environment and action space.

Artificial Neural Networks (ANNs) are computer systems inspired by **Biological Neural Networks (BNNs)** which form the brain in living individuals. Currently, they are being deployed in a variety of tasks including but not limited to Computer Vision, Speech Recognition or Natural Language Understanding/Generation. ANNs are also applied to playing video games which is of interest in this study.

Each ANN has certain properties which contribute to the performance and the behaviour of the network. Firstly, the structure of the network is an important quality which the designer should take care of. The structure of each architecture consists of the number of hidden layers and the number of nodes (neurons) within each layer of the network. The task of each neuron in a layer, whether it is an input layer, hidden layer or output layer, is to take a group of weighted inputs (features), apply an activation function and return an output.

The other property of the NNs is the weights the neurons have to deal with. Weights are hyper-parameters which define the behaviour (output) of each node. These hyper-parameters have a considerable contribution to the behaviour of the network and must be well tuned and optimized.

There have been many studies in order to efficiently optimize the weights and even the structure of the NN itself. Differential Evolution (DE) [1] is one of the mentioned methods, which takes the weights of the network as a genome, and evolves it iteratively using biologically inspired operators in order to improve the aptness of the candidate.

While DE is widely used to optimize the weights of the network, **NeuroEvolution of Augmenting Topologies (NEAT)** [2] is an example of **Topology (structure) and Weight Evolving Artificial Neural Network (TWEANN)**. It attempts to simultaneously learn the appropriate weights and topology of the **Feedforward Neural Network (FFNN)**.

The other method used for playing games by artificial intelligence is **Reinforcement Learning (RL)** [3]. In RL we have an agent which "acts" by performing certain actions which are dependent on the stimuli of the environment and on the behavioural policy of the agent itself. After an action, the agent will experience a reward or a penalty. RL enables us to learn an optimal policy for the agent by making it performing its task several times.

II. Approach

In this study, we tried to apply bio-inspired methods to play 2D video games. Firstly, we designed an ANN whose weights were optimized using DE. Secondly, we used NEAT to evolve from scratch a complete ANN. Finally, we used RL in order to be able to compare the performance of the mentioned methods with one another.

A. Bio-Neural Network

Our Bio-Neural Network uses the Differential Evolution algorithm to evolve its own weights. It is an iterative process that proceeds for a defined number of generations. Despite its simple nature, this algorithm has proved to be very effective in many optimization problems. Essentially, each individual in the population is compared with its own evolved version, preserving the most effective version with regard to a given fitness measure. An individual's evolution occurs through the combination with three other individuals randomly picked from the same population. The combination may occur on the whole **dimensionality** of the **problem** or, as in our case,

on a lower-dimension fraction of the problem (portions of the neural network). DE is governed by two parameters, the **crossover rate** and the **differential weight**.

B. NEAT

NEAT efficiently evolves the optimal neural network thanks to three key techniques. The first is called **complexification**, it starts searching for the optimal topology from low-dimensional spaces (perceptron architectures) and then it moves to higher-dimensional space. The second is called **historical markings**, it allows to match networks of different topologies and it enables the crossover. Finally, the **speciation** divides the population into species in order to keep incompatible network separated and to ensure the mating between the same species.

C. Reinforcement Learning

The main technique used is called **Q-Learning**. The agent will estimate a function $Q : S \times A \rightarrow \mathcal{R}$, which returns the expected reward ($r \in \mathcal{R}$) for the state/action pair ($(s, a) \in S \times A$). The agent will then choose the next actions based on the ones that maximize Q. The core of the algorithm is a **value-iteration update** which is used to estimate Q:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t)+\alpha(r_t+\gamma \max_a(Q(s_{t+1}, a)) \quad (1)$$

where α is the learning rate, r_t is the reward at time t for have taken action a_t when at state s_t. λ is a discount factor ($\lambda < 1$) used to enable convergence to the optimal Q.

III. Case Study Environments

We tried to experiment with our proposed solutions on famous instances of 2D games. We decided to consider **Flappy Bird** and **Pac-Man** as our case studies since they present simple game mechanics which reduces the analysis complexity to a manageable size.

A. Flappy Bird

Flappy Bird[1] is a side-scroller game in which a bird tries to fly between columns of Super Mario[2]-like green pipes held from the top and the bottom with an empty space in between. The objective is not to hit the pipes and navigate forward through the environment. The player has only two possible moves: doing nothing, which makes the bird fall due to gravity, or fly, which makes the bird flap upwards. In this way, the player is supposed to keep the bird between the pipes and not to hit them. We used an open-source implementation for our study which is freely available on GitHub[3].

[1]**Flappy Bird**: https://en.wikipedia.org/wiki/Flappy_Bird
[2]**Super Mario**: https://en.wikipedia.org/wiki/Super_Mario
[3]**GitHub Flappy Bird**: https://github.com/sourabhv/FlapPyBird

B. Pac-Man

Pac-Man[4] is a classic game with a yellow character which the player controls. The player navigates through a maze which contains some dot points (the "food") and four ghosts. The four ghosts chase Pac-Man and if Pac-Man comes into contact with one of them, he loses. The goal of the game is to collect all the dots. There are also four power pellets that provide Pac-Man with the temporary ability to eat the ghosts and earn bonus points. For Pac-Man, we tried two implementations of its environment:

- **Gym OpenAI**: the first implementation[5] of Pac-Man is taken from Gym library which is a toolkit from OpenAI[6] community. Gym provides open source implementations of many 2D Atari games and robotic simulations. Each game can either be represented by an RGB image as an array of shape (201,160,3) or the RAM allocation in the simulated Atari machine with a size of 128 bytes;
- **UC Berkeley**: The other implementation of Pac-Man is done by a professor in the University of California Berkeley for the assignments of his course "Introduction to Artificial Intelligence"[7]. The implementation provides three sizes for the maze: small 7×20, medium 12×20 and the original size 27×27.

IV. Experiments

A. Flappy Bird

The features used during our experiments were three: the **vertical distance from the lower pipe**, the **horizontal distance from the next pair of pipes** and the **player velocity on the y-axis**. Instead of the number of pipes successfully passed, we decided to use the number of **collision checks** per second as a score function to **maximize** (basically, how much time Flappy stays alive). This allows a faster convergence especially in the initial phase of the game. Moreover, we experimented by varying the **game's difficulty** (which can be easy, normal or hard) by changing the "gap" between the pipes and by also varying the **number of generations** and the **size of the populations**. We also tried several specific configurations for the DE and the NEAT algorithms. Mutation rates and crossover rate were kept **fixed** since we noticed that they do not influence the final performances.

Bio-Neural Network

The creation of the version based on the Differential Evolution algorithm was done by combining several Python libraries, including DEAP[8], PyTorch[9] and multiprocessing[10]. The generation of individuals, i.e. the definition of new sets of weights of the neural network, took place through a **Xavier-like values initialization**. We tested our DE algorithm by employing two combination strategies. The first called **shallow**

[4]**Pac-Man**: https://en.wikipedia.org/wiki/Pac-Man
[5]**Gym MsPacman**: https://gym.openai.com/envs/MsPacman-ram-v0/
[6]**OpenAI**: https://openai.com/
[7]**Intro to AI**: https://inst.eecs.berkeley.edu/~cs188/fa18/
[8]**DEAP**: https://deap.readthedocs.io/en/master/
[9]**PyTorch**: https://pytorch.org/
[10]**multiprocessing**: https://docs.python.org/3.4/library/multiprocessing.html

randomly picks one of the matrices of weights between two layers of the neural network and applies the crossover. The second one, defined as **normal**, considers each weight matrix and randomly picks the index of a single row to which the crossover is to be applied.

We designed four feed-forward neural networks for our purpose. All those network architectures share the same input and output layer:

- **shallow**: $input(3) \rightarrow hidden(4) \rightarrow output(2)$;
- **deep**: $input(3) \rightarrow hidden(8) \rightarrow hidden(4) \rightarrow output(2)$;
- **wide**: $input(3) \rightarrow hidden(16) \rightarrow output$;
- **wider**: $input(3) \rightarrow hidden(32) \rightarrow output(2)$.

Hidden layers use **ReLU** as activation function while the output layer uses **LogSoftmax**. The individual selection procedure used was the **Random Selection** of $k = 3$ individuals.

NEAT

The NEAT model was evolved by testing several possible combinations of parameters. We trained it with and without **elitism** and also by varying the **starting number of hidden nodes** (from 4 to 16). The initialization weights and bias were sampled from a **standard normal distribution** (zero mean, unit variance) in the range $[-100, 100]$. The FFNN used only ReLU as the activation function.

Reinforcement Learning

The reinforcement learning experiments were done by employing an existing implementation of Q-Learning for Flappy Bird[11], which was trained for 5000 games.

B. Flappy Bird Experimental Results

DE and NEAT evolved FFNN proved to be successful in both the "easy" and "normal" version of Flappy Bird. They were able to avoid completely the collisions with the pipes. However, when we tried with the "hard" difficulty, things changed. Figure 2 shows the performance of each EA employed. We discovered that a wider network was performing better than a deep one. On the contrary, NEAT evolved FFNN showed that smaller network can also give better performances (see Figure 1 for an example of NEAT network). Moreover, elitism does not provide any sensible improvement to the fitness. Table I shows the fitness obtained by the best individuals from both DE and NEAT compared with the RL trained agent. Despite our efforts, the RL version still outperforms the other approaches (see Table III for all the results).

C. Pac-Man

Since Pac-Man is more complex than Flappy Bird, we spent some time trying to devise the most representative features in order to better train the agents. When using Berkeley's implementation, the final features selected were: if **Pac-Man can eat ghosts**, the **distance from the closest ghost**, the **distance from the closest power pellet**, the **distance from the closest food**, the **number of food pills remaining** and the **current game score**. When using OpenAI implementation, we simply used the **entire RAM signature** of the current game state as features.

We tried to **maximize** the total score obtained by Pac-Man which is equal to the total number of food eaten during a game.

[11] **Q-Learning Flappy Bird**: https://github.com/chncyhn/flappybird-qlearning-bot

TABLE I
BEST OVERALL FITNESS OBSERVED WHILE PLAYING FLAPPY BIRD (AT HARD DIFFICULTY).

Algorithm	Best Fitness
DE	3205
NEAT	2556
RL	**5659**

Bio-Neural Network

We experimented using Berkeley's **small environment** with one ghost and the complete default OpenAI environment. For both environments we used a three layer neural network. The Berkeley version consists of $input(35) \rightarrow hidden(140) \rightarrow output(5)$ while the OpenAI version consists of $input(128) \rightarrow hidden(512) \rightarrow output(9)$. Hidden layers use **ReLU** as activation function while the output layer uses **LogSoftmax**. The individual selection procedure used was the **Random Selection** of $k = 3$ individuals.

NEAT

Unfortunately, during the experiments on Berkeley's Pac-Man with NEAT, we noticed that our Python implementation was heavily taxing the RAM memory of the machine (over 40GB). It was therefore impossible to test extensively the NEAT implementation since it required too many computational resources. For the OpenAI version, we tried several parameter combinations (with/without elitism, etc.) and several numbers of starting hidden nodes (4, 16).

Reinforcement Learning

Since the possible states of a Pac-Man game are too many to be represented explicitly, **Approximate Q-Learning** was used in order to represent the Q function. Basically, each $Q(s, a)$ is represented by a linear combination of some state's features and some weights $Q(s_t, a_t) = \sum_{i=0}^{N} f_i(s_t, a_t)w_i$. Therefore, we are not learning Q directly anymore, but we are learning the weights w_i instead. We used three slightly different features for this approach: the **number of ghosts one step away** from the Pac-Man, the **distance from the closest food** and **if the Pac-Man can eat food**. Experiments were performed using only Berkeley's version of Pac-Man.

D. Pac-Man Experimental Results

The Pac-Man game presented a real challenge for the EAs. As we can see from Figure 3, the performance of both DE and NEAT were not satisfactory. They were unable to produce

TABLE II
BEST OVERALL FITNESS OBSERVED WHILE PLAYING PAC-MAN (WITH THE SMALL ENVIRONMENT AND ONE GHOST).

Algorithm	Best Fitness
DE	-684
NEAT	$\sim$
RL	**1154**

a valid agent for playing Pac-Man (this happened both for the Berkeley and OpenAI's implementations) while the RL implementation clearly outperformed both (see Table II for the fitness results). We have some hypothesis regarding the causes of this issue:

- Pac-Man has a **dynamic environment** (ghost moves, food depletes, etc.) so it is much more difficult to evolve something;
- Pac-Man is a **multi-objective optimization problem** since we need to catch all the food in the fastest way possible and avoid all ghosts. Therefore, our fitness function and algorithms may be unfit for it;
- **Wrong parameter initialization**: it may be the case that, if we let DE and NEAT run for many generations, we would have gotten better individuals. This approach happens to have worked in some previous works [4];
- **Wrong features**: the features we designed and the RAM signature maybe are not enough to create an agent using EA algorithms since they may provide too little information to play correctly.

E. Other Environments

In order to test our proposed solutions on other environments different from the case studies, we used two of the Gym library simulations, so that we could observe their generalization capabilities.

- **MountainCar**[12] This experiment consists of a car positioned in an one-dimensional track between two mountains. The strength of the car is not enough to scale the mountain in a single pass. Therefore momentum must be exploited in order to reach the goal. The input features consist of the position **velocity of the car**, while the possible actions range between pushing left, staying still and pushing right. We used $input(2) \rightarrow hidden(8) \rightarrow output(3)$ as neural network architecture;
- **CartPole**[13] This experiment consists of a pole attached by an un-actuated joint to a cart, which moves along a frictionless track. By moving the cart on left or right, the final goal is to keep the pole in equilibrium. The input features consist of **cart's position** and **velocity** together with **pole's angle** and **velocity at tip**. We used $input(4) \rightarrow hidden(8) \rightarrow output(2)$ as neural network architecture.

We optimized both FFNN using NEAT and DE. The crossover rate was set to 0.25. The differential weight parameter was set to 1. Each population had 50 individuals and we made them run for 500 generations. Our approaches produced solutions which clearly were optimal for both the MountainCar and CartPole environments.

[12]**Gym MountainCar**: https://gym.openai.com/envs/MountainCar-v0/
[13]**Gym CartPole**: https://gym.openai.com/envs/CartPole-v1/

V. CONCLUSION & FUTURE WORK

Evolutionary techniques proved themselves to be robust and efficient methods which can be employed to solve tasks like playing games. Moreover, even if they are intrinsically stochastic with respect to Reinforcement Learning algorithms (e.g. Q-Learning), they are still able to provide optimal results if correctly used. However, they are not plug-and-play methods. They still need some careful investigation and planning before to be used on certain types of tasks, especially regarding parameters tuning. This was evident during our experiments with Pac-Man. The EA methods did not provide any meaningful result, while their RL counterpart proved itself to be reliable and precise. That happened because Pac-Man provides a dynamic game environment and it can also be considered as a multi-objective optimization problem (catch all the food in the quickest way possible without being caught by the ghosts). In order to evolve correctly an agent for Pac-Man, other solutions must be pursued. For instance, NSGA-2 algorithm could be tried in order to find the best tradeoff between the various objectives. In conclusion, DE and NEAT are really efficient and they do not need complex initialization on games which provides a "static" environment with a single objective to be maximized/minimized. However, when the game complexity starts to rise, more careful thought is needed in order to make them successful.

VI. CONTRIBUTIONS

The work has been carried out as a group task, and therefore it has been made possible by cooperation and communication between the team's members. However, each of us has focused on a specific task:

- Giovanni implemented Reinforcement Learning and NEAT components of the study and the related parameter tunings needed for each game and environment;
- Mahed implemented the Neural Networks required for the project and looked for appropriate environments and implementations of the games to be used;
- Andrei took the mentioned Neural Networks and integrated them with a Differential Evolution component for the weight optimization.

The experiments and the report writing were done equally by all members of the group. All the supporting code can be found at this link `gth.com/bioai`.

REFERENCES

[1] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, Dec 1997. [Online]. Available: https://doi.org/10.1023/A:1008202821328

[2] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[3] L. Busoniu, R. Babuska, B. D. Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators (Automation and Control Engineering)*. CRC Press, 2010.

[4] M. Gallagher and M. Ledwich, "Evolving pac-man players: Can we learn from raw input?" 05 2007, pp. 282 – 287.

APPENDIX A
FIGURES AND TABLES

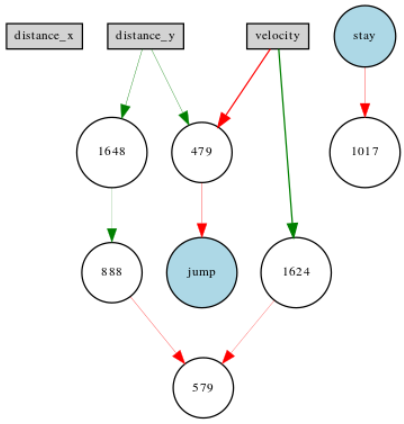

Fig. 1. FFNN evolved by NEAT to play Flappy Bird. Blue nodes are the outputs while grey nodes represent the inputs. Apparently, only the vertical distance from the pipes and the player velocity matters for deciding which action to take. Red links specify that the weights on those connections are negative, while the green links specify that they are positive. Thicker lines specify the magnitude of the weights.

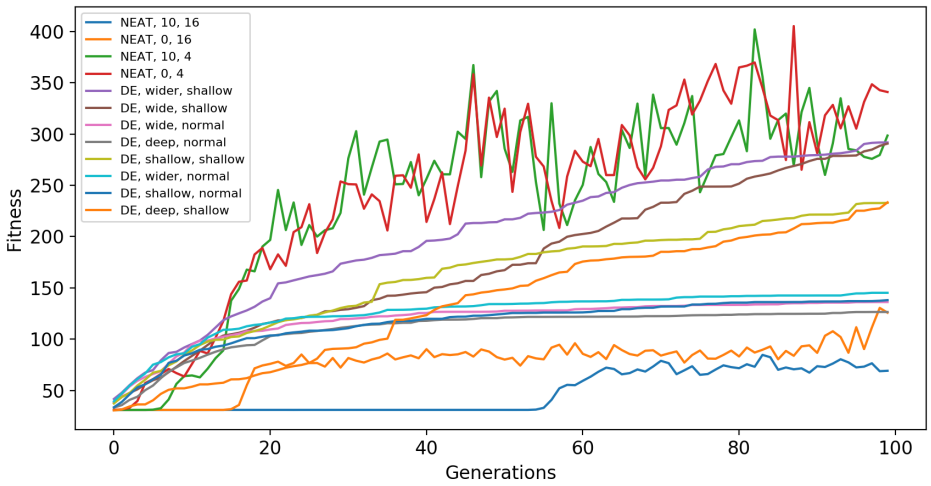

Fig. 2. Flappy Bird. Mean fitness of populations trained by employing DE and NEAT with the difficulty set to hard. We can immediately notice how NEAT performs better than DE despite having a higher variance.

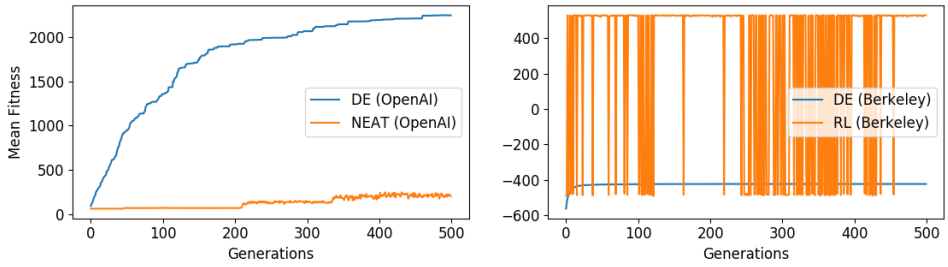

Fig. 3. Pac-Man. Mean fitness of the population trained with NEAT and DE. On the left, we can see the results for the OpenAI's implementation. On the right, the result for Berkeley's implementation. The fluctuations of the fitness for the RL implementation are caused by the fact that sometimes, at the starting state of the game, the ghost was generated too close to the Pac-Man, making it die instantly.

TABLE III
ALL EXPERIMENTAL RESULTS FOR FLAPPY BIRD.

Mean Fitness (Last Generation)	EA	Difficulty	Total Generations	Population Size	Architecture/Elitism Individuals	Weight Update/N. of Hidden Nodes
734.055	DE	hard	500	200	deep	shallow
3084.32	DE	easy	300	100	shallow	normal
797.41	DE	normal	100	100	shallow	normal
1515.84	DE	normal	100	100	deep	shallow
1115.88	DE	normal	300	100	shallow	normal
1234.66	DE	hard	300	100	shallow	shallow
3811.79	DE	easy	100	100	deep	shallow
138.07	DE	hard	100	100	shallow	normal
1381.39	DE	hard	1000	200	wide	shallow
1299.35	DE	easy	300	100	deep	normal
725.51	DE	easy	100	100	deep	normal
232.93	DE	hard	100	100	shallow	shallow
233.44	DE	hard	100	100	deep	shallow
474.51	DE	hard	300	100	deep	shallow
4271.88	DE	normal	100	100	shallow	shallow
402.03	DE	normal	100	100	deep	normal
217.01	DE	hard	300	100	shallow	normal
126.5	DE	hard	100	100	deep	normal
3643.02	DE	hard	2000	200	wide	shallow
5659.0	DE	normal	300	100	shallow	shallow
5659.0	DE	easy	300	100	shallow	shallow
5252.75	DE	normal	300	100	deep	shallow
3412.78	DE	easy	100	100	shallow	shallow
213.77	DE	hard	300	100	deep	normal
1075.95	DE	normal	300	100	deep	normal
972.33	DE	easy	100	100	shallow	normal
5406.93	DE	easy	300	100	deep	shallow
1464.4	NEAT	normal	100	100	0	4
1799.49	NEAT	easy	100	100	10	4
3927.53	NEAT	easy	100	100	0	16
2176.68	NEAT	normal	100	100	10	16
2477.1	NEAT	normal	300	100	0	4
2968.98	NEAT	easy	100	100	10	16
138.67	NEAT	normal	300	100	10	16
1991.61	NEAT	normal	300	100	0	16
341.06	NEAT	hard	100	100	0	4
1873.27	NEAT	easy	300	100	0	4
125.81	NEAT	hard	100	100	0	16
192.65	NEAT	hard	300	100	0	16
253.36	NEAT	hard	300	100	0	4
2501.02	NEAT	easy	100	100	0	4
2488.31	NEAT	normal	100	100	0	16
319.31	NEAT	hard	300	100	10	4
3265.37	NEAT	easy	300	100	0	16
2811.96	NEAT	normal	300	100	10	4
2311.09	NEAT	easy	300	100	10	16
69.16	NEAT	hard	100	100	10	16
3210.34	NEAT	normal	100	100	10	4
367.68	NEAT	hard	300	100	10	16
3361.53	NEAT	easy	300	100	10	4
298.57	NEAT	hard	100	100	10	4