Neuroevolution of a Pac-Man Player: An attempt

Alessio Quercia

Abstract—This paper is intended to show the results obtained by applying neuroevolution on Pac-Man video game. The main idea was to model a Pac-Man player using a neural network and to train this model to succeed in playing the video game by itself by using genetic algorithms. Some constraints had to be applied on the original problem (the video game rules) to simplify it and to reach some reasonable results. Several tests have been made by using different initial neural network topologies as well as different fitness functions, error functions and parameters for the adopted genetic algorithm.

1 Introduction

Video games resulted to be interesting challenges for artificial intelligence algorithms, in particular for reinforcement learning algorithms. Indeed, many approaches in modelling players of different games, such as Pac-Man, have been pursued using genetic algorithms as learning algorithms. In general, reinforcement learning algorithms try to evaluate the performance of a given model on a given task, by considering some factors specified in the fitness function (or value function, or objective function) [1], [2]. Step by step, the model behavior on the task tend to improve to maximize the fitness function, and to have a better performance on the task.

In this paper, a neuro-evolutionary approach (NEAT) has been used to allow Pacman players modelled with a neural networks to learn how to play and score as many points as possible playing Pac-Man video game. NEAT is a genetic algorithm, which not only aims at optimizing the neural networks connection weights, but also at minimizing their topologies (or structures). In fact, starting from a uniform population of fully connected neural networks with no hidden units (only input and output units), the algorithm keeps the neural networks structures minimized at each epoch, by adding a new node or a new connection only if it improves the network performance on the given task (that is, its fitness).

In the following sections, Pac-Man video game and some artificial intelligence approaches applied to it are described, before analyzing a general neuro-evolutionary algorithm. Then, the model used in this project is explained, together with the adopted apporach (NEAT) and its main features and parameters. Before analyzing the evolution process, the adopted initial population and fitness function are described. In the last sections, the simulations and experiments made during this project are explained, giving some details about the system architecture and the implementation itself. Finally, the results are discussed and some conclusions, as well as some possible future work, are drawn on the obtained results.

Alessio Quercia, Intelligent Systems, A/A 2017-2018, Università degli Studi di Milano, via Celoria 18, Milano, Italy
E-mail: alessio.quercia@studenti.unimi.it

2 Background

The following subsections briefly describe Pac-Man video game, some of the most recent artificial intelligence approaches using it as a test-bed and neuro-evolutionary approach, the one adopted for this project.

2.1 Pac-Man video game

Pac-Man is an arcade video game produced by Namco and created by Toru Iwatani and Shigeo Fukani in 1980. It is considered as an icon both for the video game industry and for the popular culture [3].

Pac-Man's goal for the player is to navigate Pacman inside a maze (Figure 4 and to collect all the pellets and power-pills to progress to the next level. At the beginning of each level three opposing ghosts spawn in the hideout (which Pacman is unable to enter), positioned in the centre of the maze, and one spawns on the top of it, and they try to catch Pacman during the game, by following and eating him on collision. Pacman start the game with three lives and loses one each time he is eaten by a ghost. When Pacman has no more lives, the player loses the game. Typically, the ghosts chase Pacman during the game, but when he eats a power-pill, the situation is reversed for a limited amount of time, because he gains the power to eat them on collision and they cannot harm him, so they start escaping from him. When a ghost is eaten, he is restored inside the hideout, where he can restart moving after some time and he cannot be eaten again.

The player can improve his score by collecting (eating) pellets (10 points), power-pills (50 points) and bonus items that sometimes appears in the maze, which all give a specific amount of points. The player is also rewarded exponentially for each ghost Pacman eats while still under the effect of the same power-pill, starting from 200 (for eating a single ghost) up to 1600 (for eating all the four ghosts with a single power-pill) [4].

2.2 Artificial intelligence approaches to Pac-Man

Despite its simplicity, Pac-Man, as many other games and real-time video games, has been used to test and experiment artificial intelligence algorithms. Indeed, video games pose many interesting and complex challenges, such as strict time constraints and simultaneous moves.

Many different approaches to Pac-Man video game has been done during the last years concerning two main topics: an artificial intelligence modelling a Pacman player and an artificial intelligence modelling each of the ghosts. In the following, some of them are briefly described:

- Modelling a Pac-Man player as a simple finite state machine and set of ruleset and allowing it to learn through the application of population-based incremental learning (PBIL) [5].
- Modelling Pacman agents through neural networks and allowing them to learn from few on screen information by evolving them using a $(\mu + \lambda)$-Evolution Strategy [6].
- Modelling both Pacman and the ghosts with different sets of fuzzy classes and applying specific policies to the agents in terms of fuzzy actions that are defuzzified to obtain their moves. Q-Learning algorithm is used to optimize the agents value functions [7].
- Modelling a Pac-Man player using neural networks and evolving them using a neuro-evolutionary approach (NEAT) [3].
- Modelling the four ghosts as neural networks and evolving them using a neuro-evolutionary approach (NEAT) in order to allow them to learn to teamwork and to develop strategies to win against a real Pac-Man player [4], [8].

Moreover, there is a competition (MS Pac-Man versus Ghost Team) that allows people to use their own artificial intelligence for MS Pac-Man (a Pac-Man variation) or for the ghost team, and to compete against each others [9].

2.3 Neuroevolution

Neuroevolution is the artificial evolution of neural networks using genetic algorithms. It searches for a neural network that performs well at a given task in the space of behaviors. This approach represents a faster and more efficient alternative to those methods estimating a utility function (or a value function), such as Q-Learning. Neuroevolution is effective in problems with continuous and high-dimensional state spaces and for non-Markovian tasks. Moreover, neural networks allow to map from genotype to phenotype in an efficient way, making neuroevolution a suitable approach to reinforcement learning tasks, given that no supervision is required.

In traditional neurovolution apporoaches, a fixed neural network topology (or structure) is chosen at the beginning (before starting the evolution itself) and a population composed by neural networks having that topology is created. Usually a fully connected neural network with a single hidden layer (plus the input and output layers) is used. Indeed, a fully connected neural network can approximate any continuous function [10]. The goal of a fixed-topology neuro-evolutionary approach is to optimize the neural networks connection weights by searching in the weights space through the evolution of the population, that is neural networks selection, reproduction and mutation.

However, the connections weights are not the only aspect of neural networks that contribute to their behavior. Indeed, also their topology affects their functionality. Evolving neural networks structure along with their connection weights can significantly improve the performance of a neuro-evolutionary approach. This approach allows to minimize the dimensionality of the search space of the connection weights, having neural networks with minimal structures at each epoch. An example of this approach is NeuroEvolution of Augmenting Topologies (NEAT), in which topologies are minimized throughout the evolution, providing an efficiency improvement [11], [12].

3 Theoretical model

The following subsections describe the simplifications made on Pac-Man game during this project, the adopted method (NEAT), the different parameters, initial populations (neural network topologies) and fitness functions used and the evolution process itself.

3.1 Pac-Man simplifications

Given the Pac-Man rules described above, some simplification, heuristics and constraints have been applied to have a simplified version of the game on which to be able to apply the neuro-evolutionary approach. They are described in the following list:

- the bonus items have been removed from the game;
- Pacman has only one life per game (instead of three). This simplification has been applied to reduce time need to perform experiments;
- two ghosts chase Pacman actively, while the other two random move;
- the moves generated as output by the neural networks for the Pacman agents are then constrained to avoid Pacman from selecting forbidden or useless moves (such as moving against a wall, while already moving in a corridor) and penalizing moves that brings to loops (such as going forward and backward forever or going forever towards a wall). Even though, the possibilit not to select a move has been granted, allowing the agent to keep moving towards a direction (even towards a wall), because real players can decide not to move in certain situations, to let the ghosts reach them, and then move to a power-pill to eat them all.

3.2 NEAT

NeuroEvolution of Augmenting Topologies (NEAT), created by K. O. Stanley and R. Miikkulainen, is a neuro-evolutionary method which, as already mentioned above, is designed to both optimize the neural networks connection weights and minimize their topologies throughout the evolution. However, evolving the structures incrementally presents several technical challenges:

1) the need for an appropriate genetic representation that allows disparate topologies to cross over in a meaningful way;
2) the need to protect topological innovation from premature disappearance from the population, so as to give it some time to be optimized;
3) the need for a way that allows to minimize the topologies throughout the evolution.

NEAT approach solve the listed challenges applying the techniques, that are the main features of the approach itself. They are described in the following subsections [11], [12].

3.2.1 Genetic encoding

NEAT represents each genome as a list of connection genes, each of which refers to two node genes being connected. Each node gene can be an input, hidden or output node. Each connection gene specifies the in-node, the out-node, the connection weight, whether or not the connection gene is enabled, and an innovation number, which allows to find corresponding genes when two genomes cross over during mating.

Connection weights mutate as in any genetic algorithm, whereas structural mutations occur in two ways:

- add connection mutation, that is a new connection gene with random weight is added between two unconnected nodes;
- add node mutation, that is a new node gene is added and placed between two already connected nodes, by disabling their connection gene and by adding two new connection genes to connect the two nodes to the new node. The connection leading into the new node receives weight equal to 1, whereas the connection leading out receives the same weight as the old connection.

3.2.2 Tracking genes through historical markings

The historical origin of a gene allows to know which genes have to be matched during the cross over. Indeed, two genes with the same historical origin represents the same structure since they are both derived from the same gene. To keep track of the historical origin of the genes, a global innovation number is incremented each time a new gene is created and assigned to it. During the cross over, genes with the same innovation number are aligned and called matching genes. The others are called disjoint, if they occur within the range of the other parent's innovation number, or excess, if they occur outside it. In composing the offspring, genes are randomly chosen from either parent at matching genes, whereas all excess or disjoint genes are always included from the more fit parent.

3.2.3 Protecting innovation through speciation

NEAT protects the topological innovation by speciating the population, allowing the organisms to compete primarly within their own niches instead of with the entire population. Indeed, each species has organisms that have similar structure and that share the fitness of their niche. This prevents species to become too big, even if many of its organisms perform well.

3.2.4 Minimizing dimensionality through incremental growth from minimal structure

NEAT suggests starting from a uniform population of networks with zero hidden nodes, because, as already mentioned above, the method gradually increment the networks topologies through the mutations, so as to keep them minimal at each epoch.

3.3 Parameters

NEAT method has several parameters controlling the evolution. The main ones are reported below:

- pop_size, corresponding to the number of organisms (neural networks) inside the population;
- epoch_number, corresponding to the number of epochs the algorithm should perform, that is the amount of times the population should be brought from a generation to the next one, through selection, reproduction and mutation (including structure mutation);
- weight_mutation_power, that is the power with which a connection weight is changed during a mutation;
- mutate_add_node_prob, that is the probability of a new node to be added to a neural network structure.
- mutate_add_link_prob, that is the probability of a new link to be added to a neural network structure.
- dropoff_age, corresponding to the number of epochs in which no fitness improvement for a species is tolerated, that is, if a species reaches a certain number (dropoff_age) of generations without improving, it is culled from the population along with all of its speculative structures.

3.4 Initial Population

As already mentioned above, NEAT focuses on structure minimization during throughout the evolution process. Thus, the suggested initial population is a uniform population of fully connected neural networks with no hidden nodes, that is just with an input and an output layers. Indeed, the hidden nodes are added during the evolution process.

Different initial neural networks topologies have been tested during this project. The most performing one is a

fully connected neural network with 22 input units (plus a bias unit), 4 hidden units and 5 output units. Its input and output units are described in the following scheme:

- Input layer:
  - pacman_row: Pacman's row in the matrix;
  - pacman_col: Pacman's column in the matrix;
  - pacman_left: Pacman's left cell content;
  - pacman_right: Pacman's right cell content;
  - pacman_up: content of the cell positioned above Pacman's one;
  - pacman_down: content of the cell positioned below Pacman's one;
  - for each ghost:
    - ghost_row: ghost row in the matrix;
    - ghost_col: ghost col in the matrix;
    - ghost_mode: ghost mode (normal, vulnerable, died, cage);
  - nearest_food_row: row of food nearest to Pacman;
  - nearest_food_col: column of the food nearest to Pacman;
  - nearest_powerup_row: row of the power-pill nearest to Pacman;
  - nearest_powerup_col: column of the power-pill nearest to Pacman;
- Output layer:
  - move_left, decision to move left;
  - move_right, decision to move right;
  - move_up, decision to move up;
  - move_down, decision to move down;
  - no_action, decision not to "press" any directional key;

3.5 Fitness function

The fitness function is used to evaluate an organism performance on a specific task, which in this case corresponds a neural network performance in playing Pac-Man videogame.

To measure the fitness, many factors can be considered, such as the total score and the elapsed time since the game start.

Two different fitness functions have been tested during this project (both trying to maximize the score):

$$f_1 = performed\_score \tag{1}$$

$$f_2 = \frac{1}{error + k} \tag{2}$$

where

$$error = max\_score - performed\_score \tag{3}$$

and k is a small normalization value (set to 0.00001), max_score is the maximum score achievable in a single level with the simplifications listed above (that is 12454) and $performed\_score$ is the score performed by the neural network.

3.6 Evolution process

The evolution process has the following scheme:

1) population initialization;
2) n networks (depending on the number of available threads) run separated simulations in parallel, that is each network starts a new Pac-Man game and plays it step-by-step (corresponding to the game time). At each step:
   a) each neural network have the current state of the game as input and producing a move as output;
   b) each move produced as output by a neural network is constrained, not to have unfeasible or useless moves (as described above);
   c) each neural network performance is evaluated at the end of its game, using a fitness function;
3) when all the organisms inside the population have been evaluated for their performances in playing their Pac-Man game, the most performing organisms are selected to reproduce. The best ones are directly cloned in the new population (elitism);
4) the reproduction occurs among the selected organisms, until the new population is filled;
5) the different mutations (connection weights and structure) occur with their respective probability;
6) the new population is ready, if the total number of epochs is reached, then the process stops; otherwise, it continues from point 2.

4 Simulations and Experiments

In the following subsections the main simulations and experiments made during the project are described, together with the adopted system architecture and some details about the code implementation.

4.1 System architecture

I ran the simulations on my laptop, characterized by an Intel Core i7-6700HQ CPU, a Nvidia Geforce GTX 950M GPU and 8 GB RAM. The implemented algorithm uses every CPU core.

4.2 Implementation details

The Pac-Man video game implementation was taken from GitHub user leonardo-ono [13] and re-adapted to the project needings. Indeed, a replay system was required and many classes had to be re-adapted to the project code.

As already mentioned before, the evolution process is fully distributed on multiple cores, allowing to have a nice speed-up of the performance with respect to the serialized version. Indeed, depending on the number of the available cores on the computer running the application, more or less organisms are evaluated in parallel.

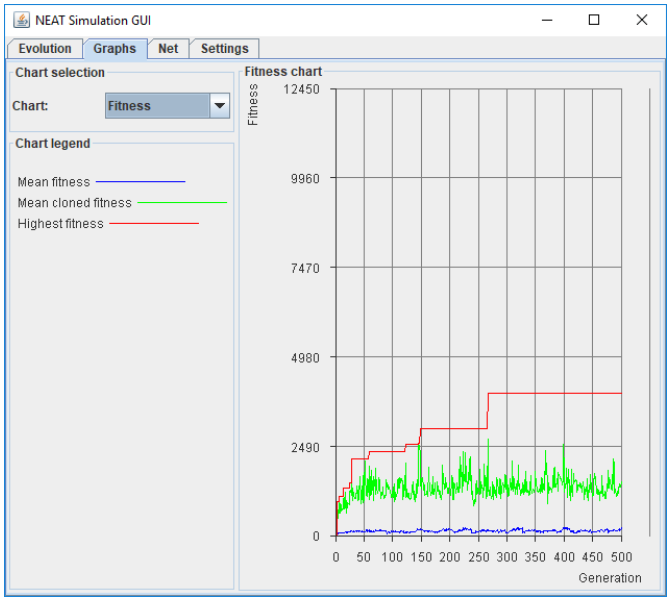

Fig. 1: Best result obtained using fitness 1, mutate_add_node_prob = 0.03 and mutate_add_link_prob = 0.08.

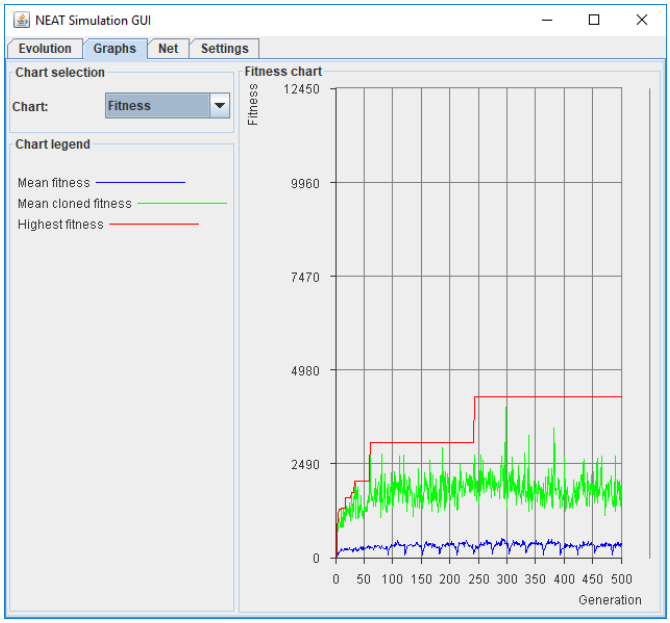

Fig. 2: Best result obtained using fitness 1, mutate_add_node_prob = 0.3 and mutate_add_link_prob = 0.8.

5 Results

Several tests have been made, using different parameters and fitness functions. The population size was fixed to 30 neural networks, with the topology described above, the weight_mutation_power and the dropoff_age have been respectiveley fixed to 2.5 and 50, and each test was run for 500 epochs, due to the low performing system architecture requiring a lot of time to run such experiments. Indeed, not to have long experiments, each neural network played a single Pac-Man game in each epoch. In general, in just 500 epochs the most performing neural network reached a score of about 4000, as shown in Figures 1, 2, 3, 4. The trend of the fitness functions suggest that giving the neural networks some more time, that is more epochs, better results can be obtained, that is neural networks scoring more than 4000 points in a single game can be obtained.

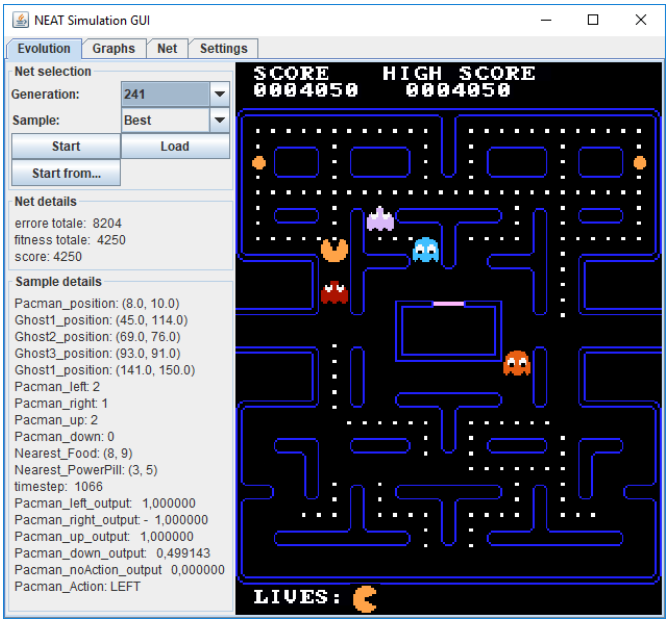

Fig. 3: Best neural network using fitness 1, mutate_add_node_prob = 0.3 and mutate_add_link_prob = 0.8 made a score of 4250.

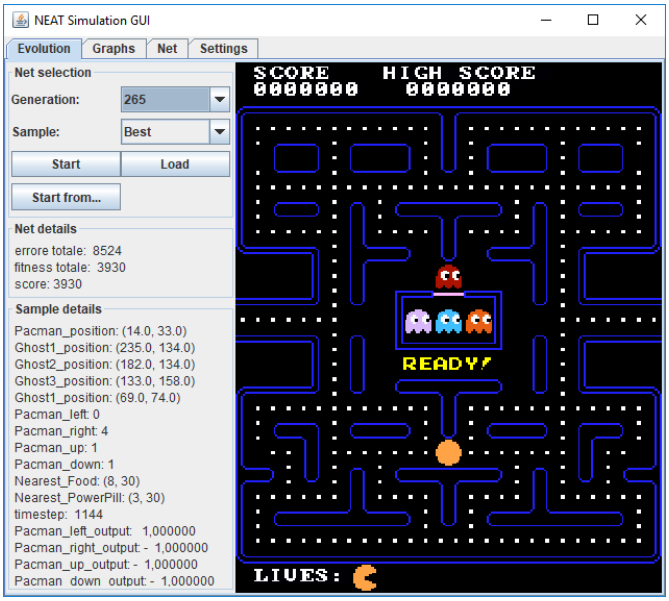

Fig. 4: Best neural network using fitness 1, mutate_add_node_prob = 0.03 and mutate_add_link_prob = 0.08 made a score of 3930.

6 Conclusions

The aim of this paper, and of the project itself, was modelling Pacman players through neural networks and making them learn to play Pac-Man games through neuroevolution. It was indeed possible to achieve good results, even though, with high probability, better results can be obtained by considering other factors (such as time spent for a game) or penalizing ”bad” moves in the fitness function and by letting the evolution process run for more epochs, which requires an highly performing system architecture.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, p. 436, 2015.

[2] C. Bishop, *Pattern recognition and machine learning*. Springer New York, 2006.

[3] M. Miranda, A. A. Sánchez-Ruiz, and F. Peinado, “A neuroevolution approach to imitating human-like play in ms. pac-man video game.” in *CoSECivi*, 2016, pp. 113–124.

[4] M. Wittkamp, L. Barone, and P. Hingston, “Using neat for continuous adaptation and teamwork formation in pacman,” in *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*. IEEE, 2008, pp. 234–242.

[5] M. Gallagher and A. Ryan, “Learning to play pac-man: An evolutionary, rule-based approach,” in *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 4. IEEE, 2003, pp. 2462–2469.

[6] M. Gallagher and M. Ledwich, “Evolving pac-man players: Can we learn from raw input?” in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*. IEEE, 2007, pp. 282–287.

[7] N. A. Rossini, C. Quadri, and N. A. Borghese, “Clever pac-man.” in *WIRN*, 2011, pp. 11–19.

[8] G. N. Yannakakis and J. Hallam, “Evolving opponents for interesting interactive computer games,” *From animals to animats*, vol. 8, pp. 499–508, 2004.

[9] P. Rohlfshagen and S. M. Lucas, “Ms pac-man versus ghost team cec 2011 competition,” in *Evolutionary Computation (CEC), 2011 IEEE Congress on*. IEEE, 2011, pp. 70–77.

[10] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

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

[12] ——, “Efficient evolution of neural network topologies,” in *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, vol. 2. IEEE, 2002, pp. 1757–1762.

[13] leonardo ono, “Java2dpacmangame,” https://github.com/leonardo-ono/Java2DPacmanGame, 2017.