Annals of the University of Craiova, Mathematics and Computer Science Series
Volume 37(1), 2010, Pages 76–85
ISSN: 1223-6934

Evolving a Minimum Input Neural Network Based Controller for the Pac-Man Agent

Dragoş Nicolescu

Abstract. In this paper we present the development and implementation of a neural network-based controller for the Pac-Man agent which is fed with minimal information about the environment. The implementation of the game we used differs from the original. In this regard some key aspects were changed in order to provide new challenges for the agent and thus better test our controller. A neural network is used to compute a desirability value for the locations to which the agent can move. The non-deterministic nature of the game does not allow for fast and accurate feedback, thus a different method for training was developed. Accordingly, we considered a genetic algorithm to evolve weights for the neural network. To conclude, the controller adapts very well to the environment, resulting in well-trained agents that can complete several consecutive levels of the game.

2010 Mathematics Subject Classification. 68T42; 68T20.
Key words and phrases. neural network, Pac-Man agent, genetic algorithm, minimum input.

1. Introduction

Implementing Artificial Intelligence (AI) methods in games has been a great way to test new developments in the field for quite some time. This is due to the capacity of games to simulate conditions that occur in the real world without complicating the way in which these are represented. One of the games that is most used as an AI testing platform is Pac-Man, an arcade game that proved very popular amongst gamers and researchers alike.

Game controllers based on neural networks are nothing new, starting with their use in classic board games [1] and continuing with more dynamic environments [2] which are characteristic of new-age games. In terms of agent control a neural network offers adaptability and a fast learning process with minimum supervision. The weak point of such a network is the need for feedback, the learning process being dependent on receiving the results of actions after they are taken.

In this paper we try to overcome the disadvantage a neural network faces when no immediate feedback is available by using a genetic algorithm for training. The evolutionary process produces an adaptable agent that learns by itself rather than relying on hardcoded instructions. There is a need for such agents in new games [3] as the game industry is trying to leave behind the era of hard-coded enemies and to replace them with new ones that adapt to the actions of a player. The resulting agent proves to be competitive by the game standards, surprising us by sometimes clearing several levels of the game.

Received May 5, 2012.

2. Pac-Man in AI

As we mentioned before, Pac-Man is one of the most used games in testing AI methods. The game offers a challenge that is simple to understand, yet requires practice and strategy in order to overcome [4]. For this reason researchers have focused on developing a controller that can understand and adapt to the certain situations that the agent encounters during the game [5], [6].

In order to have a clear understanding of what those situations might be we will first present the game mechanics and the differences between the original and our implementation.

2.1. The Pac-Man Game. Pac-Man was launched in Japan on May 22, 1980 by the Namco Company. Typical of the 80s, it is an arcade game that achieved great success, being popular even 30 years after it appeared. Proof of this came when the search engine Google honoured the game's 30 year anniversary in 2010 by allowing visitors to play a version of the game on the Google homepage. Due to the time spent playing the game by employees the US economy suffered a 120 million dollar loss on that day.

The game environment consists of a maze populated by 3 types of objects: 2 reward types and an obstacle (or "enemy") type. The rewards are *pellets* and *power-pills* and the obstacles are *ghosts*.

The agent must navigate through the maze and gather all rewards (for which he gets points) while avoiding the enemies. Pellets are the most numerous objects in the maze and they are worth the smallest amount of points. Power pills are rare and spread across the maze. They bring the agent more points than the pellets and also allow him to "eat" the ghosts for a limited time. Eating a ghost brings the agent even more points and also resets the ghost to the starting position from which it cannot escape until the effect of the power pill is over. Ghosts are enemies that move through the maze at different speeds. Usually there are 5 of them and all have different behaviours when it comes to chasing the agent (from random movement to aggressive following). When the agent eats a power pill the ghosts become *scared* which means that they move slower and try to run away from the agent.

The agent has a limited number of lives (usually 3). When he is touched by a ghost he loses a life and both he and the ghosts are reset to their starting positions. If all 3 lives are lost so is the game. If the agent manages to collect all rewards in the maze he receives points and moves to the next level which consists of a different maze (keeping all remaining lives).

Considering the game mechanics and the objective of the game (gather as many points as possible) there are many strategies for playing. For example, a player can avoid ghosts and try to complete as many levels as possible, while another can chose to chase ghosts and eat them, gathering points but also risking losing lives if ghosts quickly change their scared state.

2.2. Changes to the Game. When implementing the mechanics of the game we changed a few of the rules in order to simplify the representation of the game and take out elements that only had entertainment value for human players. We also wanted to provide additional challenges for our agent to overcome. The most important change is the behaviour of the ghosts: all our ghosts chase the agent aggressively, except for when they are scared. The exact way the ghosts work is the following: when not in a scared state they try to minimize the distance to the agent; when in a scared state they try to maximize this distance. However, this behaviour makes the ghosts behave deterministically, which is not realistic. A random element is added to ghost movement by introducing, let us say, a 0.2 chance that a ghost will move in a random direction excluding the one that minimizes/maximizes the distance to the agent. This change, combined with the fact that ghosts are not allowed to turn back the way they came (except when changing states) produces the desired behaviour, the ghosts being a lot more aggressive than the ones in the original game.

In the classic implementation, when the player moves to a new level the map changes to a new maze. Completing a level in our game will just reset the same map (all entities are moved to their original positions and the score and number of agent lives are maintained). We opted for this change because maze shape and size has no influence on the way our agent learns, as long as all entities in the maze maintain the same characteristics.

Another important change we made is removing "out time" for ghosts. As we mentioned before, in the original game, when a ghost is eaten, it is transported to a "crypt" in the middle of the maze and it cannot escape until the effect of the power pill is over. This mechanic takes ghosts out of the game, allowing players to collect rewards uninhibited for a short period. In our emulation of the game, once a scared ghost is eaten, it immediately spawns in the centre of the maze in the normal, not scared state, and it starts chasing the agent again. Similar changes were made in [7] in order to make ghosts more dangerous.

The original mazes sometimes contained tunnels through which the agent and ghosts could pass to get to the opposite side of the map (effectively providing a point at which the map wrapped around). Our maze contains no tunnel, as we considered this would have minimal impact on evolved agent strategies.

Some other elements of the original game were left out of our emulation because it was considered they had minimal impact on training the agent. These include: Pac-Man does not slow down when collecting a reward; no fruit (another rare type of reward) and no additional life at 10 000 points.

3. Methodology

Implementation of the game and controller was done in .NET using C# and the Windows Forms interface. All entities were implemented in modular, object-oriented style. For example, the map object contains a list of map square (location) objects and the neural network is made up of several layer objects which, in turn, contain several perceptron objects.

Each location object represents a square of the map grid and can contain walls (left, right, upper, lower), a pellet or a power pill and is either accessible or inaccessible. All these features are stored in a map array which can be quickly changed to represent a different maze. Distances between such locations are useful when feeding input to the neural network. In order to save time when training, these distances are calculated recursively when the map is loaded and stored for later use. This method is described in [7] and proves to be a very efficient approach.

When the application starts, the map and other entities in the game are initialized and map distances are calculated. Upon initiating a new training session, a neural network is created according to a specified architecture. Next a new genetic algorithm is instanced using the selected evolution parameters. This algorithm adjusts the weights of the neural network according to the evolutionary paradigm. When training is over, the best evolved individual is saved along with other information which reflects the progression of the training session for later analysis.

3.1. Neural Network Approach. Before the agent moves to a new location all possible locations are analysed (between 1 and 4 map locations adjacent to the current location). For each of these locations several parameters are fed into the neural network and the output represents the desirability of that certain location. The agent then chooses the location with the highest desirability value and moves to it, repeating this process until he dies.

The neural network that we used is a fully-connected multilayer perceptron (MLP) that uses a hyperbolic tangent activation function. The input layer contains 6 weightless input units, each for a characteristic of the analysed location, as described in Table 1.

Input Perceptron	**Input Characteristic**	**Input Range**	**Input Significance**
1	Dist_Gh1	{0,...,28}	Distance to nearest ghost.
2	State_Gh1	{-1,1}	State of nearest ghost (-1 for normal, 1 for scared).
3	Dist_Gh2	{0,...,28}	Distance to second nearest ghost.
4	State_Gh2	{-1,1}	State of second nearest ghost.
5	Dist_Pellet	{0,...,28}	Distance to nearest pellet.
6	Dist_Pill	{0,...,28}	Distance to nearest power pill.

TABLE 1. Input to the neural network.

The hidden layer consists of 10 computing units and is fully connected to the input layer. This gives the hidden layer a total of 60 weights. Fully connected to the hidden layer is the output layer which only contains 1 unit that produces the network output. The output layer has 10 weights, bringing the total length of the weight vector of the network to 70.

As mentioned above, all neurons use a hyperbolic tangent activation functions, meaning that the activation function of the i-th neuron for v_i, as weighted sum of inputs, and y_i as output, is given by:

$$\phi(y_i) = tanh(v_i/4)$$

Note. Division of the weighted sum of inputs by 4 was added in order to determine a smooth activation in the output layer for high input values.

3.2. The Hybrid Neural Network-Genetic Algorithm. Evolving weights for the neural network is accomplished using a standard genetic algorithm. As we mentioned before, the total number of weights that control the output of the neural network is 70. A vector consisting of these 70 weights (genes) is considered as a chromosome in the evolutionary paradigm.

When the genetic algorithm assumes control of tuning the neural network weights, it generates a population of individuals. The individuals (chromosomes) are generated using a random generator that produces uniformly distributed double values between -1 and 1.

After the population is generated, each individual takes turns in controlling the agent (via the neural network) for a set number of games. The total score achieved in these games represents the fitness of the individual.

Once all individuals in the population have an updated fitness value (each has assumed control of the agent once), the process of creating a new generation starts. First, all individuals are ordered from worst to best according to their fitness value. The worst performing n individuals (where n is the *death rate*) are eliminated from the population.

The best performing n individuals are selected as parents for the children that will replace the eliminated individuals. Every pair of parents produces 2 offspring, by combining each half of the first vector with the corresponding half from the second vector.

As soon as a new child is created he goes through the mutation phase. For each gene (weight) a random double between 0 and 1 is generated. If the generated number is less than the *mutation chance* the gene will be mutated. This consists of generating another random number between 0 and 1 and adding or subtracting it to/from the gene. Both addition and subtraction operations have equal chances to mutate the gene, but only one is selected at random.

After mutation occurs for all new individuals, these are added to the new generation that goes through the same process as the generation before it.

4. Results

An agent was evolved using the purposed method and his results were compared with the results of different controllers [5], [7]. The main focus was on minimizing the size of the input to the neural network as much as possible while still evolving a well-playing agent.

4.1. Task. An efficient location evaluator was developed in [7] by providing the neural network with the distances to all normal and scared ghosts, the distances to the closest pellet and power pill and the distance to the closest junction. Such info allows for a fine-tuned agent that adapts to specific situations in the environment.

By feeding the network with less information we aim to simplify the evolutionary process by means of reducing the individual length, thus better covering the search space using a relatively small population. Our goal is to shorten the evolution process and the size of the input to the network as much as possible in order to still evolve a competitive, adaptive agent.

4.2. Setup. The maze that we used is a simplified version of one of the classic Pac-Man levels, consisting of a 15x15 grid and no tunnels (Fig. 1).

Technically, inside the maze there are 131 pellets and 4 power pills. The enemy agents are 5 ghosts with speeds equal to that of the agent (when not scared). When the agent collects a power pill the ghost speed drops to half and the ghosts enter scared state for 20 moves (a move represents a transition from a grid square to another adjacent one). The duration of the ghost scared state is cumulative.

Another important aspect of the testing setup is the amount of reward points the agent receives for each action. These numbers are presented in Table 2.

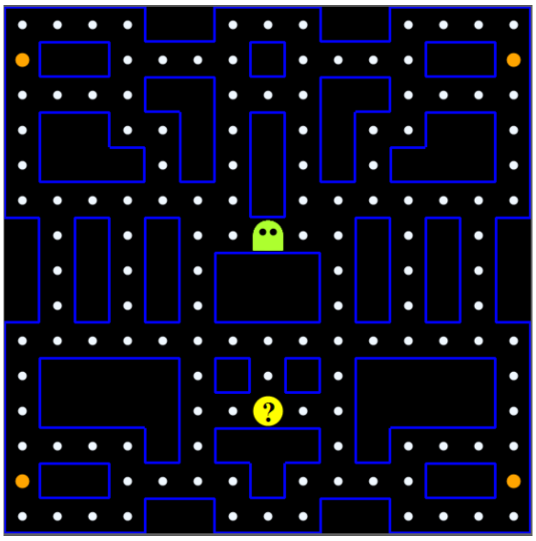

FIGURE 1. The maze in the initial state with all entities in the starting positions.

Action	**Reward (points)**
Eating a pellet	10
Eating a power pill	50
Eating a scared ghost	100
Completing a level	500

TABLE 2. Rewards for specific actions.

The evolution process is controlled by the parameters represented in Table 3 along with the values we heuristically set for them when evolving our best agent.

Parameter Name	**Value**
Population Size	100
Number of Generations	50
Mutation Chance	0.1
Life Span	10
Death Rate	20

TABLE 3. Parameters values controlling evolution.

Life span is the number of games for which each individual assumes control of the agent, and *death rate* represents the percent of the old population that is replaced when creating a new generation.

4.3. Experimental Results. The evolution process that produced our strongest controller is presented in Fig. 2 (the evolution parameter values are the ones in Table 3). For each individual in each generation a total fitness is determined by summing all scores he obtained while controlling the agent. The total fitness is then divided by

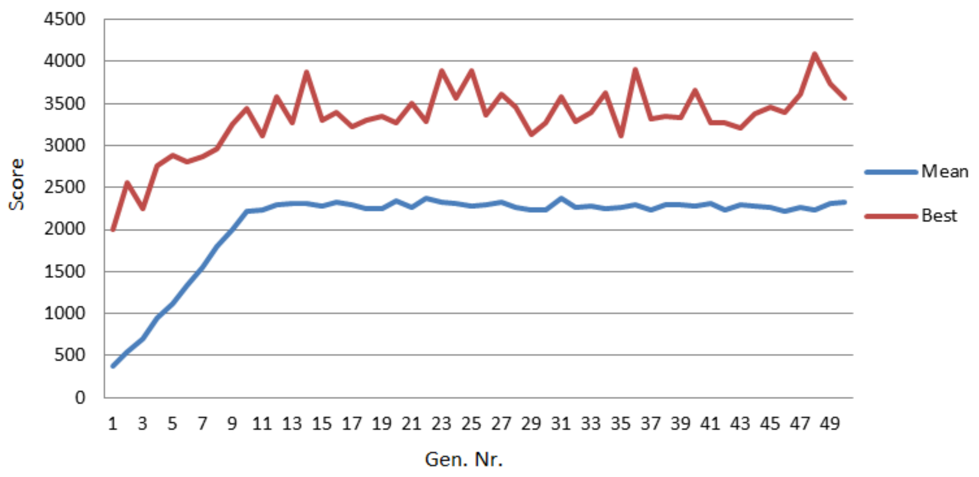

FIGURE 2. The progression of individual best and mean scores during evolution.

the *life span* and a mean fitness value is obtained (for the sake of simplicity, we will refer to this value as *individual fitness*).

In Fig. 2 the *Mean* line describes the progression of the mean of all individual fitness values in a generation. Values along the *Best* line represent the fitness of the best individual in a generation. *Mean* starts at 382.32 (gen. nr. 1) and ends with 2327.84 (gen. nr. 50), while *Best* ranges from 2004 for the first generation to 3569 for the last one, with a peak value of 4083 for generation number 48.

After completing the training phase the best individual in the last generation was chosen to enter the testing phase. In order to make for an easier comparison we adopted the testing method used in [5]. The agent plays 100 games and the results are analysed taking into consideration the mean over 100 for the following values: the number of pellets and power pills collected, the number of eaten ghosts, the number of wins and the score. For our agent these values are:

- Ghosts eaten (mean over 100 games): **2.28**
- Games won (mean over 100 games): **0.35**
- Power pills (mean over 100 games): **5.31**
- Score (mean over 100 games): **2307.7**

4.4. Observations and Discussion. A normal round of the game played by the neural network based agent proceeds as follows:

(1) At the start of the game the agent focuses on the closest power pill, going straight for it. Once the pill is collected ghosts are scared and start avoiding the agent, trying to maximize the distance to his location.
(2) The agent moves to the next power pill, eating any ghosts he finds on the way. However, he does not actively chase scared ghosts for more than a few moves if this takes him further from the targeted power pill.
(3) Step 2 is repeated until there are no more power pills left. By this time the agent has cleared the outskirts of the maze and the ghosts are still scared for a good amount of time due to the cumulative effect of the pills. By this point the agent would usually have eaten an average of 2 ghosts.

(4) Once there are no more pills in the maze the agent takes advantage of the scared time left to collect the remaining outermost pellets. At this stage he actively avoids scared ghosts, being aware that the absence of any more power pills would mean that a scared ghost, once eaten, would immediately return as an aggressive enemy.
(5) During an average game the scared time ends when there are few pellets left in the centre of the maze. Once this happens, the agent switches to avoiding the aggressive ghosts at all costs, even if this takes him further away from the remaining pellets. He tries to get all ghosts to follow him and then head for the last rewards. The non-deterministic path of the ghosts (0.2 chance to make a random move, 0.8 chance to close in on Pac-Man) means that often the agent is ambushed inside a corridor and he loses a life (in which case he moves to step 6). In 10% of the cases the agent manages to complete the level without losing any lives.
(6) If the agent has lost a life trying to collect the last remaining pellets he and the ghosts are reset and his behaviour is the one described at step 5.

In order to obtain an objective evaluation of the minimum input neural network based controller (MINNC) efficiency a comparison with other Pac-Man controllers is required. It was decided to first compare the MINN with two Influence Map based controllers [5]: LIMA (Limited Influence Map Agent) and IMA (Influence Map Agent). IMA uses global knowledge about the environment, similarly to MINNC, while LIMA uses minimal local knowledge. A second comparison is done between MINNC and the neural network based controller (MLP-20) developed in [7]. This controller is similar to MINNC, the difference being that MLP-20 feeds the neural network with more information about the environment.

4.4.1. *Neural Network vs. Influence Maps.* A direct comparison between the results of the neural network based controller and the influence map based controllers (LIMA and IMA) implemented in [5] can be observed in Table 4.

Result	**NN Controller**	**LIMA**	**IMA**
Ghosts	2.28	0.91	1.55
Wins	0.35	0.08	0
Pills	5.31	3.67	3.68
Score	2307.7	1515	1532

TABLE 4. Results of neural network, LIMA and IMA based controllers.

It is important to note that the environment in which the neural network based controller was tested has a greater degree of difficulty due to the extremely aggressive ghosts. Despite this fact, the controller described in this paper clearly performs better when considering all analysed aspects. Due to the dangerous ghost behaviour, it was expected that the controller would evolve to avoid ghosts at all times and rather focus on pellets and pills. However, the agent proved to be very versatile, managing to find the right time to eat ghosts and avoiding them for the rest of the game.

An important aspect of the comparison between the neural networks based agent and the IMA and LIMA agents is the number of levels completed. For IMA the number is 0 while LIMA manages to improve the influence map method and obtain 8 wins in 100 games. The neural network controller wins 35 games out of 100, which is a remarkable number, considering the added difficulty of the environment. The number of wins also explains why the agent collects an average of 5.31 pills per game when there are only 4 pills in a level. Every 3 levels the agent completes an additional level, which means he collects (3+1)×4 pills every 3 games. This results in 16 collected pills every 3 games, an average of 5.3 pills per game (close to the 5.31 value that the test produced).

4.4.2. *Minimum Input vs. Complete Input.* In [7] the author develops 5 different neural network based controllers for Pac-Man, testing them in deterministic and non-deterministic versions of the game. The non-deterministic environment is very similar to the one used in this paper. However, there are few important differences: the environment used in [7] contains 220 pellets (89 more than our environment) and the reward for eating a ghost starts at 200 and doubles for every other ghost ate in quick succession (our reward has a constant value of 100). The scoring difference might not seem great, but it means a difference of 10890 points for completing a level in ideal conditions.

The best controller developed in [7] (MLP-20) is similar to MINNC, the difference being that MLP-20 uses a larger input set. This additional information consists of the distance to and state of all ghosts (as opposed to only the two closest ghosts as is the case of MINNC) and the distance to the closest junction.

After 100 games MLP-20 obtained a mean score of 4781, which is almost double the score of MINNC. However, MLP-20 was evolved over 1000 generations, while MINCC is the result of only 50 generations. According to [7] MLP-20 did not score more than 2000 points before generation 200. This means that after just 50 generations MINNC performs better than MLP-20 does after 200 generations. This happens despite the difference in scoring which grants MLP-20 an advantage.

The fast evolution of MINNC is attributed to the smaller input that allows for covering a more diverse population set while using less generations and individuals. Given enough generations it is probable that both controllers will evolve to obtain similar scores, but when considering a short evolution time MINNC is sure to produce a better agent.

5. Conclusions and Further Research

This paper aimed to present a way to simplify the process of evolving a neural network based controller for an agent in a non-deterministic environment. This objective is accomplished by reducing the set of possible solutions through decreasing the size of the input vector, resulting in a Minimum Input Neural Network based Controller (MINNC).

When compared to other methods of controlling an agent, neural networks prove to be well suited for the task, managing to learn fast and adapt easily to the environment.

In order to test if reducing the size of the input to the neural network can speed up the learning process while producing the same results we opted for comparing MINNC to a complete input neural network based controller. The MINNC produces results similar to those of the complete input controller after a 5 times shorter evolution process. Another effect of reducing the input size is the increased simplicity of the neural network and the reduction in computational resources needed to implement the controller.

The only downside to using a MINNC can be noticed in more complex environments where the state of the environment cannot be fully captured in small feature vector. In addition to this, a less complex neural network is not capable of learning very subtle behaviours, the kind that would be needed in order to accomplish complicated tasks. This issue can be fixed by pre-processing the input in such a way that the vector size is reduced, while still encapsulating rich information about the environment.

While reducing input size proves to be very efficient way of reducing the time and resources needed to evolve a well-playing agent the optimization can be taken further by adjusting the parameters of the evolution process. This direction can provide great results and is sure to be the focus of further research.

Another interesting approach to using a neural network to control an agent is based on real-time training of the network, eliminating the genetic algorithm. Such an approach can prove to have a very fast response time and high adaptability to sudden changes in the environment, being well suited for the non-deterministic nature of the game.

Although Pac-Man came out more than 30 years ago, it still provides a challenging environment for research into agent control and behaviour. Apart from the Pac-Man agent the team of ghosts has also received attention from the scientific community, resulting in interesting developments in cooperative agent behaviour. A lot of the recent advances in Non-Player Character (NPC) AI which are implemented in modern games with immersive gameplay are based on methods first tested using Pac-Man, earning the game a special status both amongst players and researchers.

References

[1] Alex Lubberts, Risto Miikkulainen, Co-Evolving a Go-Playing Neural Network, Coevolution: Turning Adaptive Algorithms Upon Themselves, Birds-of-a-Feather Workshop, *Genetic and Evolutionary Computation Conference* **6** (2001), 14–19.

[2] John Reeder, Roberto Miguez, Jessica Sparks, Michael Georgiopoulos, Georgios Anagnostopoulos, Interactively Evolved Modular Neural Networks for Game Agent Control, *IEEE Symposium on Computational Intelligence and Games* (2008), 167–174.

[3] Ross Graham, Hugh McCabe and Stephen Sheridan, Neural Networks for Real-time Pathfinding in Computer Games, *Proceedings of ITB Research Conference 2004* (2004), 223–231.

[4] N. Beume, H. Danielsiek, C. Eichhorn, B. Naujoks, M. Preuss, K. Stiller, S. Wessing, Measuring Flow as Concept for Detecting Game Fun in the Pac-Man Game, *Proceedings of the 2008 IEEE Congress on Evolutionary Computation* (2008), 2447–3454.

[5] D. Nicolescu and C. Stoean, An Efficient Influence Map Based Controller for the Pac-Man Agent, *Journal of Knowledge, Communications and Computing Technologies* **3**, no. 1 (2011), 53–67.

[6] N. Wirth, M. Gallagher, An Influence Map Model for Playing Ms. Pac-Man, *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games*, IEEE Press, Perth, Australia (2008), 228–233.

[7] Simon M. Lucas, Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man, *IEEE Symposium on Computational Intelligence and Games*, (2005), 203–210.

(Dragoş Nicolescu) Faculty of Exact Sciences, Department of Informatics, University of Craiova, 13 A.I. Cuza Street, Craiova, 200585, Romania
E-mail address: dragosm.nicolescu@yahoo.com